Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

May 27, 2015

Abstract

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

1 Introduction

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

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

2 HRB Slicing guarantees IFC Noninterference

theory NonInterferenceInter
imports ../HRB-Slicing/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 cannot be altered during a program run. Thus, we have to extend the prerequisites of the slicing framework in [9] accordingly in a new locale:

```plaintext
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 :: ('node ('(.'Entry'.') )) and get-proc :: 'node ⇒ 'pname  
and get-return-edges :: 'edge ⇒ 'edge set  
and procs :: ('pname × 'var list × 'var list) list and Main :: 'pname  
and Exit::'node ('(.'Exit'.') )  
and Def :: 'node ⇒ 'var set and Use :: 'node ⇒ 'var set  
and ParamDefs :: 'node ⇒ 'var set and ParamUses :: 'node ⇒ 'var set list +  
fixes H :: 'var set  
fixes L :: 'var set  
fixes High :: 'node ('(.'High'.') )  
fixes Low :: 'node ('(.'Low'.') )  
assumes Entry-edge-Exit-or-High:  
[valid-edge a; sourcenode a = (-Entry-)]  
⇒ targetnode a = (-Exit-) ∨ targetnode a = (-High-)  
and High-target-Entry-edge:  
∃ a. valid-edge a ∧ sourcenode a = (-Entry-) ∧ targetnode a = (-High-) ∧  
kind a = (λs. True),  
and Entry-predecessor-of-High:  
[valid-edge a; targetnode a = (-High-)] ⇒ sourcenode a = (-Entry-)  
and Exit-edge-Entry-or-Low: [valid-edge a; targetnode a = (-Exit-)]  
⇒ sourcenode a = (-Entry-) ∨ sourcenode a = (-Low-)  
and Low-source-Exit-edge:  
∃ a. valid-edge a ∧ sourcenode a = (-Low-) ∧ targetnode a = (-Exit-) ∧  
kind a = (λs. True),  
and Exit-successor-of-Low:  
[valid-edge a; sourcenode a = (-Low-)] ⇒ targetnode a = (-Exit-)
```
and DefHigh: Def (-High-) = H
and UseHigh: Use (-High-) = H
and UseLow: Use (-Low-) = L
and HighLowDistinct: H \cap L = \{\}
and HighLowUNIV: H \cup L = UNIV

begin

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

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

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

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

lemma get-proc-High:
  get-proc (-High-) = Main
proof –
  from High-target-Entry-edge obtain a where valid-edge a 
      and sourcenode a = (-Entry-) and targetnode a = (-High-) 
      and intra-kind (kind a) by (fastforce simp:intra-kind-def)
  from \langle valid-edge a \rangle \langle intra-kind (kind a) \rangle
  have get-proc (sourcenode a) = get-proc (targetnode a) by (rule get-proc-intra)
  with \langle sourcenode a = (-Entry-) \rangle \langle targetnode a = (-High-) \rangle
  get-proc-Entry
  show \?thesis by simp
qed
lemma Entry-path-High-path:
assumes (-(Entry-)) -as→* n and inner-node n
obtains a' as' where as = a'#as' and (-(High-)) -as'→* n
and kind a' = (λs. True)✓
proof (atomize-elem)
from (-(Entry-)) -as→* n (inner-node n)
show ∃a' as'. as = a'#as' ∧ (-(High-)) -as'→* n ∧ kind a' = (λs. True)✓
proof (induct n' ≡ -(Entry-) as n rule: path-induct)
case (Cons-path n'' as n' a)
from (n'' -as→* n') (inner-node n') have n'' ≠ -(Exit-)
by (fastforce simp: inner-node-def)
with valid-edge a (sourcenode a = (-(Entry-)) (targetnode a = n'')
have n'' = -(High-) by -(drule Entry-edge-Exit-or-High, auto)
from High-target-Entry-edge
obtain a' where valid-edge a' and sourcenode a' = -(Entry-)
and targetnode a' = -(High-) and kind a' = (λs. True)✓
by blast
with valid-edge a (sourcenode a = -(Entry-) (targetnode a = n'')
⟨n'' = -(High-)⟩ have a = a' by (auto dest: edge-det)
with (n'' -as→* n') (n'' = -(High-)) ⟨kind a' = (λs. True)✓⟩ show ?case by blast
qed fastforce

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

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

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

2.2 Low Equivalence

In classical noninterference, an external observer can only see public values, in our case the L-variables. If two states agree in the values of all L-variables, these states are indistinguishable for him. Low equivalence groups those states in an equivalence class using the relation ≈_L:
definition lowEquivalence :: ('var → 'val) list ⇒ ('var → 'val) list ⇒ bool
(infixl ≈_L 50)
where s ≈_L s' ≡ ∀ V ∈ L. hd s V = hd s' V

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

lemma relevant-vars-Entry:
assumes V ∈ rv S (CFG-node (-Entry-)) and (-High-) /∈ [HRB-slice S] CFG
shows V ∈ L
proof –
from (V ∈ rv S (CFG-node (-Entry-))) obtain as n'
where (-Entry-) →∗ 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-) →∗ 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 $\langle V \in \text{Use}_{SDG} n' \rangle$ have False
  by $(\text{drule SDG-Use-parent-Use}, \text{simp add: Exit-empty})$
thus $?\text{thesis}$ by simp
next
case inner
with $\langle (-\text{Entry}) - as \rightarrow r, * \text{ parent-node } n' \rangle$ obtain $a' \text{ as'}$ where $as = a' \# as'$
  and $(-\text{High}) - as' \rightarrow r, * \text{ parent-node } n'$
by $(\text{fastforce elim: Entry-path-High-path}, \text{simp: intra-path-def})$
from $\langle (-\text{Entry}) - as \rightarrow r, * \text{ parent-node } n' \rangle \langle as = a' \# as' \rangle$
have source-node $a' = (-\text{Entry})$ by $(\text{fastforce elim: path}, \text{cases simp: intra-path-def})$
show $?\text{thesis}$
proof (cases $as' = []$)
case True
with $\langle (-\text{High}) - as' \rightarrow r, * \text{ parent-node } n' \rangle$ have parent-node $n' = (-\text{High})$
  by $(\text{fastforce simp: intra-path-def})$
with $\langle n' \in \text{HRB-slice } S \rangle \langle (-\text{High}) \notin \lfloor \text{HRB-slice } S \rfloor_{CFG} \rangle$
have False
  by $(\text{fastforce dest: valid-SDG-node-in-slice-parent-node-in-slice})$
  simp: SDG-to-CFG-set-def
thus $?\text{thesis}$ by simp
next
case False
with $\langle (-\text{High}) - as' \rightarrow r, * \text{ parent-node } n' \rangle$ have hd (source-nodes $as'$) = $(-\text{High})$
  by $(\text{fastforce intro: path-source-node simp: intra-path-def})$
from $\langle \text{False} \rangle$ have hd (source-nodes $as'$) $\in \text{set (source-nodes as')}$
  by $(\text{fastforce intro: hd-in-set simp: source-nodes-def})$
with $\langle as = a' \# as' \rangle$ have $\text{hd (source-nodes as')} \in \text{set (source-nodes as')}$
  by (simp add: source-nodes-def)
from $\langle \text{hd (source-nodes as')} = (-\text{High}) \rangle$
have $\text{valid-node (hd (source-nodes as'))}$ by simp
have $\text{valid-SDG-node (CFG-node (-\text{High}))}$ by simp
with $\langle \text{hd (source-nodes as')} = (-\text{High}) \rangle$
  $\langle \text{hd (source-nodes as')} \in \text{set (source-nodes as')} \rangle$
  $\forall n''. \text{ valid-SDG-node } n'' \land \text{ parent-node } n'' \in \text{set (source-nodes as')}$
  $\rightarrow V \notin \text{ Def}_{SDG} n''$
have $V \notin \text{ Def } (-\text{High})$
by $(\text{fastforce dest: CFG-Def-SDG-Def}[OF \text{ valid-node (hd (source-nodes as'))}])$
hence $V \notin \text{ H}$ by (simp add: DefHigh)
thus $?\text{thesis}$ by (rule not-High-Low)
qed
qed

lemma lowEquivalence-relevant-nodes-Entry:
assumes $s \approx_L s' \text{ and } (-\text{High}) \notin \lfloor \text{HRB-slice } S \rfloor_{CFG}$
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 \((-\text{rule relevant-vars-Entry})\)

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

lemma slpa-rv-Low-Use-Low:

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

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

\( \forall c \in \text{set} \ cs. \ \text{valid-edge} \ c; \ m \ \rightarrow \ast (-\text{Low-}); \ m \ \rightarrow \ast \ast (-\text{Low-}); \)

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

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

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

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

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

\[ \implies \forall V \in \text{Use} \ (-\text{Low-}). \ \text{state-val} \ (\text{transfers}(\text{slice-kinds} \ S \ as) \ s) \ V = \text{state-val} \ (\text{transfers}(\text{slice-kinds} \ S \ as') \ s') \ V \]

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

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

moreover

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

by (fastforce intro: empty-path simp: intra-path-def)

moreover

from \( (\text{valid-node} \ m) \ \langle m = (-\text{Low-}) \rangle \) have \( (-\text{Low-}) \in \text{HRB-slice} S \)

by (fastforce intro: HRB-slice-refl)

ultimately have \( V \in \text{rv} S \ (\text{CFG-node} \ m) \)

using \((m = (-\text{Low-}))\)

by (auto intro!: red 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 UseLow show \( ?\text{thesis} \) by simp

next
case False
from \(m \rightarrow as' \Leftarrow \rightarrow (-\text{Low-})\) \(\langle m = (-\text{Low-}) \rangle\) have as' = []
proof (induct m as' m'\equiv(-\text{Low-}) \ rule:path.induct)
case (Cons-path m'' as a m)
  from \(\langle \text{valid-edge a} \rangle \langle \text{sourcenode a} = m \rangle \langle m = (-\text{Low-}) \rangle\)
  have targetnode a = (-\text{Exit-}) by -(rule Exit-successor-of-Low,simp+)
  with \(\langle \text{targetnode a} = m'' \rangle \langle m'' \rightarrow as \rightarrow (-\text{Low-}) \rangle\)
  have (-\text{Low-}) = (-\text{Exit-}) by -(drule path-Exit-source,auto)
  with False have False by -(drule Low-neq-Exit,simp)
  thus ?case by simp
qed simp
with \(\forall V \in \text{Use} (-\text{Low-})\). \(V \in \text{rv S} (\text{CFG-node m})\)
  \(\forall V \in \text{rv S} (\text{CFG-node m}). \text{state-val s} V = \text{state-val s'} V\) Nil
show ?thesis by(auto simp:slice-kinds-def)
qed

next
case (slpa-intra cs a as)
note IH = \(\forall m \rightarrow as \rightarrow s s'. \ [\text{upd-cs cs as as'}]; \ \text{same-level-path-aux cs as as'}\);
  \(\forall a \in \text{set cs}. \text{valid-edge a}; m \rightarrow as \rightarrow (-\text{Low-}); m \rightarrow as' \rightarrow (-\text{Low-})\);
  \(\forall i \less length cs. \forall V \in \text{rv S} (\text{CFG-node (sourcenode (cs ! i)))}\).\)
  \(\text{fst} (s \! Succ i) V = \text{fst} (s'! Succ i) V\);
  \(\forall i \less\! Succ (length cs), \text{snd} (s \! i) = \text{snd} (s'! i)\);
  \(\forall V \in \text{rv S} (\text{CFG-node m}). \text{state-val s} V = \text{state-val s'} V\);
  \(\text{preds (slice-kinds S as) s}; \text{preds (slice-kinds S as') s'}\);
  \(\text{length s} = \text{Suc (length cs)}; \text{length s'} = \text{Suc (length cs)}\)
  \(\equiv\ \forall V \in \text{Use} (-\text{Low-}). \text{state-val (transfers(slice-kinds S as) s) V = state-val (transfers(slice-kinds S as') s') V}\)
note riv = \(\forall i \less length cs. \forall V \in \text{rv S} (\text{CFG-node (sourcenode (cs ! i)))}\).\)
  \(\text{fst} (s \! Succ i) V = \text{fst} (s'! Succ i) V\)
from \(\langle m \rightarrow a \# as \rightarrow as' \rightarrow (-\text{Low-})\rangle\) have sourcenode a = m and valid-edge a
  and targetnode a = as \rightarrow as' \rightarrow (-\text{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 \(\langle m \rightarrow as' \rightarrow as'' \rightarrow (-\text{Low-})\rangle\) have m = (-\text{Low-}) by fastforce
  with \(\langle \text{valid-edge a} \rangle \langle \text{sourcenode a} = m \rangle\) have targetnode a = (-\text{Exit-})
    by -(rule Exit-successor-of-Low,simp+)
  from \(\text{Low-source-Exit-edge obtain a' where valid-edge a'}\)
    and sourcenode a' = (-\text{Low-}) and targetnode a' = (-\text{Exit-})
    and kind a' = (\text{as, True}) by blast
  from \(\langle \text{valid-edge a} \rangle \langle \text{sourcenode a} = m \rangle \langle m = (-\text{Low-})\rangle\)
    \(\langle \text{targetnode a} = (-\text{Exit-}) \rangle \langle \text{valid-edge a'b} \rangle \langle \text{sourcenode a'} = (-\text{Low-})\rangle\)
    \(\langle \text{targetnode a'} = (-\text{Exit-}) \rangle\)
  have a = a' by(fastforce dest:edge-det)
with \(\text{kind } a' = (\lambda s. \text{True})\) have \(\text{kind } a = (\lambda s. \text{True})\) by simp

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

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

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

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

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

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

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

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

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

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

obtain \(\text{cfs'} \text{ where } [\text{simp}] : s' = cf' \# cf s'\)

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

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

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

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

have \(\text{False}\)

proof \((\text{cases } \text{sourcecenode } a \in [\text{HRB-slice } S]_{\text{CFG}})\)

case \(\text{True}\)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

next
case \(\text{False}\)

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

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

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

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

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

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

qed

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

next
case \((\text{Return } Q p f)\)
from $\langle \text{valid-edge ax} \rangle$ : $\langle \text{kind ax} = Q \langle \text{valid-edge a} \rangle \ (\text{intra-kind (kind a)}) \rangle$
$\langle \text{source-node a} = m \rangle$ $\langle \text{source-node ax} = m \rangle$

have $\text{False by \ -(drule return-edges-only,auto simp\:intra-kind-def)}$
thus $\langle \text{thesis} \rangle$ by simp

qed simp

with $\langle \text{same-level-path-aux cs as} \rangle$ $\langle as' = ax \# asx \rangle$

have $\langle \text{same-level-path-aux cs asx by}(fastforce \ simp\:intra-kind-def) \rangle$
show $\langle \text{thesis} \rangle$

proof
$\langle \text{cases target-node a = target-node ax} \rangle$
case True
with $\langle \text{valid-edge a} \rangle$ $\langle \text{valid-edge ax} \rangle$ $\langle \text{source-node a} = m \rangle$ $\langle \text{source-node ax} = m \rangle$

have $\langle a = ax by(fastforce \ intro\:edge-def) \rangle$

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

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

have $\langle \text{rv\:\forall V} \in \text{rv} \ (\text{CFG-node \ (target-node a)})$. $\text{state-val} \ (\text{transfer} \ \text{slice-kind S a}) \ s' \ V = \text{state-val} \ (\text{transfer} \ \text{slice-kind S a}) \ s' \ V \rangle$
by $\langle \text{-\ (rule rv-edge-slice-kinds,auto)} \rangle$
from $\langle \text{upd-cs cs \ (a \neq \ as) = [[] by(intra-kind \ (kind a)) \rangle} $

have $\langle \text{upd-cs cs \ as = [] by(fastforce \ simp\:intra-kind-def)} \rangle$

from $\langle \text{target-node ax \ -\asx\rightarrow* \ (-Low-)} \rangle$ $\langle a = ax \rangle$
have $\text{target-node a = -ax\rightarrow* \ (-Low-)} \ \langle \text{by simp} \rangle$
from $\langle \text{valid-edge a} \rangle$ $\langle \text{intra-kind \ (kind a)} \rangle$

obtain $\text{cfx}$
where $\text{cfx:transfer \ (slice-kind S a) \ s = cfx\#cfs} \land \text{snd} \ cfx = \text{snd} \ cf$

apply $\langle \text{cases cf} \rangle$
apply $\langle \text{cases \ source-node a} \in [HRB-slice S] \ \text{CFG} \ \text{apply} \ \text{auto} \rangle$
apply $\langle \text{fastforce \ dest: slice-intra-kind-in-slice \ simp\:intra-kind-def} \rangle$
apply $\langle \text{auto \ simp\:intra-kind-def} \rangle$
apply $\langle \text{drule \ slice-kind-Upd} \ \text{apply} \ \text{auto} \rangle$
by $\langle \text{erule \ kind-Predicate-notin-slice-slice-kind-Predicate \ auto} \rangle$

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

obtain $\text{cfx'}$
where $\text{cfx':transfer \ (slice-kind S a) \ s' = cfx'\#cfs'} \land \text{snd} \ cfx' = \text{snd} \ cf'$

apply $\langle \text{cases cf'} \rangle$
apply $\langle \text{cases \ source-node a} \in [HRB-slice S] \ \text{CFG} \ \text{apply} \ \text{auto} \rangle$
apply $\langle \text{fastforce \ dest: slice-intra-kind-in-slice \ simp\:intra-kind-def} \rangle$
apply $\langle \text{auto \ simp\:intra-kind-def} \rangle$
apply $\langle \text{drule \ slice-kind-Upd} \ \text{apply} \ \text{auto} \rangle$
by $\langle \text{erule \ kind-Predicate-notin-slice-slice-kind-Predicate \ auto} \rangle$

with $\text{cfx} \ \forall \ i < \text{Suc} \ \text{(length cs)}, \ \text{snd} \ (s!i) = \text{snd} \ (s^i!i)$

have $\text{snds:}\forall \ i < \text{Suc} \ \text{(length cs)}$.
$snd (\text{transfer} \ (\text{slice-kind S a}) \ s \! i) =$
$snd (\text{transfer} \ (\text{slice-kind S a}) \ s' \! i)$
by $\langle \text{auto \ case-tac} \ i, \text{auto} \rangle$
from $\text{rvs cfx cfx'}$ have $\langle \text{rvs' \forall \ i < \text{length cs}}$.
$\forall V \in \text{rv} \ (\text{CFG-node \ (source-node \ (cs \ ! \ i))}) \rangle$.
\[
\text{fst (transfer (slice-kind S a) s ! Suc i) V =}
\]
\[
\text{fst (transfer (slice-kind S a) s' ! Suc i) V}
\]
\text{by fastforce}

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

qed
qed

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

note IH = \(i \land m \land as \land s \land s'

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

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

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

from \(m \Rightarrow a \Rightarrow as \Rightarrow (-Low-)) \text{ have} source-node a = m \text{ and valid-edge a}
\(\text{and target-node a} \Rightarrow as \Rightarrow (-Low-) \text{ by(auto elim:path-split-Cons)}

from \(\forall c \in set cs. valid-edge c \Rightarrow valid-edge a)
\(\text{have} \forall c \in set (a # cs). valid-edge c \text{ by simp}

show ?case

proof(cases L = {})

case True with UseLow show ?thesis by simp

next
case False

show ?thesis

proof(cases as')

case Nil

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

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

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

from \(\text{valid-edge a} \Rightarrow (source-node a = m) \Rightarrow (m = (-Low-))
\(\Rightarrow (target-node a = (-Exit-)) \Rightarrow (valid-edge a') \Rightarrow (source-node a' = (-Low-))
\(\Rightarrow (target-node a' = (-Exit-))

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

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

with \(\text{(target-node a = (-Exit-)) (target-node a \Rightarrow as \Rightarrow (-Low-))}
\(\text{have} (-Low-) = (-Exit-) \text{ by} (-\text{drule path-Exit-source,auto})

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

\text{thus ?thesis by simp}

next
case (Cons ax asx)

with \(m \Rightarrow as' \Rightarrow (-Low-) \text{ have} source-node ax = m \text{ and valid-edge ax}
\(\text{and target-node ax \Rightarrow as' \Rightarrow (-Low-)} \text{ by(auto elim:path-split-Cons)}

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

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

auto simp:slice-kinds-def)
from ⟨preds (slice-kinds S as') s'⟩ (as' = ax # asx)

obtain cf' cf's' where [simp]:s' = cf'#cf's'
by (cases s')(auto simp: slice-kinds-def)

have ∃ Q r p fs. kind ax = Q:r→pfs

proof (cases kind ax rule: edge-kind-cases)
case Intra
have False
proof (cases sourcenode ax ∈ ⌊HRB-slice S⌋)
case True
have slice-kind S ax = kind ax
by ¬ (rule slice-intra-kind-in-slice)
from (valid-edge a) (kind a = Q:r→pfs)
have unique:∃ a'. valid-edge a' ∧ sourcenode a' = sourcenode a ∧
intra-kind (kind a') by (rule call-only-one-intra-edge)
from (valid-edge a) (kind a = Q:r→pfs)
obtain x
where x ∈ get-return-edges a by (fastforce dest: get-return-edge-call)
with (valid-edge a) obtain a' where valid-edge a'
and sourcenode a' = sourcenode a and kind a' = (λ cf. False)
by (fastforce dest: call-return-node-edge)
with (valid-edge ax) (sourcenode ax = m) (sourcenode a = m)
intra-kind (kind ax); unique
have a' = ax by (fastforce simp: intra-kind-def)
with (kind a' = (λ cf. False),)
(slice-kind S ax = kind ax) (as' = ax # asx)
⟨preds (slice-kinds S as') s'⟩
have False by (simp add: slice-kinds-def)
thus ?thesis by simp

next

case False
with (kind a = Q:r→pfs) (sourcenode ax = m) (sourcenode a = m)
have slice-kind S a = (λ cf. False):r→pfs
by (fastforce intro: slice-kind-Call)
with ⟨preds (slice-kinds S (a # as)) s⟩
have False by (simp add: slice-kinds-def)
thus ?thesis by simp

next

case (Return Q' p' f')
from ⟨valid-edge ax⟩ (kind ax = Q'←p,f') (valid-edge a) (kind a = Q:r→pfs)
⟨sourcenode a = m⟩ (sourcenode ax = m)
have False by ¬ (drule return-edges-only, auto)
thus ?thesis by simp

qed simp

have sourcenode a ∈ ⌊HRB-slice S⌋

proof (rule ccontr)
assume sourcenode a /∈ ⌊HRB-slice S⌋
from this (kind a = Q:r→pfs)

next
have slice-kind $S$ $a$ = (\lambda f. \text{False}): r \to pf$
  by (rule slice-kind-Call)
with \langle \text{preds} \ (\text{slice-kinds} \ S \ (a \ # \ as)) \ s \rangle
show False by (simp add: slice-kinds-def)
qed
with \langle \text{preds} \ (\text{slice-kinds} \ S \ (a \ # \ as)) \ s \rangle \ (\text{kind} \ a = Q : r \to pf)
have pred (kind a) s
  by (fastforce dest: slice-kind-Call-in-slice simp: slice-kinds-def)
from (sourcenode a \in \{HRB-slice \ S \}_\text{CFG})
  (sourcenode a = m) (sourcenode ax = m)
have sourcenode ax \in \{HRB-slice \ S \}_\text{CFG} by simp
with \langle as' = ax \ # \ asx \rangle \ \langle \text{preds} \ (\text{slice-kinds} \ S \ as') \ s' \rangle
\exists Q \ r \ p \ fs. \ \text{kind} \ ax = Q : r \to pf$
have pred (kind ax) s'
  by (fastforce dest: slice-kind-Call-in-slice simp: slice-kinds-def)
\{ fix V assume V \in \text{Use} (\text{sourcenode} a)
  from (\text{valid-edge} \ a ) \ \langle \text{sourcenode} \ a \in \[ \to _1 * \] sourcenode a
  by (fastforce intro: empty-path simp: intra-path-def)
with (\text{sourcenode} a \in \{HRB-slice \ S \}_\text{CFG})
  (\text{valid-edge} \ a ) \ \langle V \in \text{Use} (\text{sourcenode} a) \rangle
  have V \in rv S \ (\text{CFG-node} \ (\text{sourcenode} a))
by (auto intro!: rvl CFG-Use-SDG-Use simp: SDG-to-CFG-set-def sourcenodes-def)
\}
with \forall V \in rv S \ (\text{CFG-node} \ m). \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V
  \langle \text{sourcenode} \ a = m \rangle
have Use \forall V \in \text{Use} (\text{sourcenode} a). \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V
by simp
from \forall i < Suc (\text{length} \ cs). \ \text{snd} (s \ # i) = \text{snd} (s' \ # i)
have \text{snd} (hd s) = \text{snd} (hd s') by fastforce
with (\text{valid-edge} \ a ) \ \langle \text{kind} \ a = Q : r \to pf \rangle \ (\text{valid-edge} \ ax)
\exists Q \ r \ p \ fs. \ \text{kind} \ ax = Q : r \to pf \ (\text{sourcenode} a = m) \ (\text{sourcenode} ax = m)
\langle \text{pred} (\text{kind a}) \ s \rangle \ \langle \text{pred} (\text{kind ax}) \ s' \rangle \Us v \text{Use} \langle \text{length} \ s = \text{Suc} \ (\text{length} cs) \rangle
\langle \text{length} s' = \text{Suc} \ (\text{length} cs) \rangle
have [simp]: ax = a by (fastforce intro!: CFG-equal-Use-equal-call)
from (\text{same-level-path-aux} \ cs \ as') \ \langle as' = ax \# asx \rangle \ (\text{kind} \ a = Q : r \to pf)
\exists Q \ r \ p \ fs. \ \text{kind} \ ax = Q : r \to pf$
have same-level-path-aux (a \ # \ cs) \ axx by simp
from (\text{targetnode} \ ax \ - \ axx \to \ (- \text{Low}-)) have targetnode \ a \ - \ axx \to \ (- \text{Low}-)
by simp
from (\text{kind} \ a = Q : r \to pf) \ \langle \text{upd-cs} \ cs \ (a \ # \ as) = [] \rangle
have upd-cs (a \ # \ cs) as = [] by simp
from (\text{sourcenode} a \in \{HRB-slice \ S \}_\text{CFG}) \ (\text{kind} \ a = Q : r \to pf)
have slice-kind: slice-kind $S$ $a$ = $Q : r \to pf$(\text{cpsp} \ (\text{targetnode} \ a) \ \{HRB-slice \ S \} \ fs)
by (rule slice-kind-Call-in-slice)
from \forall i < Suc (\text{length} \ cs). \ \text{snd} (s \ # i) = \text{snd} (s' \ # i)
have snds: \forall i < Suc (\text{length} \ (a \ # \ cs))
  \text{snd} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s \ # i) =
  \text{snd} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s' \ # i)
by auto(case-tac i, auto)
from (valid-edge a) (kind a = Q:r→pfs) obtain ins outs
  where (p,ins,outs) ∈ set proc by (fastforce dest!: callee-in-procs)
with (valid-edge a) (kind a = Q:r→pfs)
have length (ParamUses (sourcenode a)) = length ins
  by (fastforce intro: ParamUses-call-source-length)
with (valid-edge a)
  have ∀ i < length ins. ∀ V ∈ (ParamUses (sourcenode a))!i. V ∈ Use (sourcenode a)
  by (fastforce intro: ParamUses-in-Use)
with ∀ V ∈ Use (sourcenode a). state-val s V = state-val s’ V
have ∀ i < length ins. ∀ V ∈ (ParamUses (sourcenode a))!i.
  state-val s V = state-val s’ V
  by fastforce
with (valid-edge a) (kind a = Q:r→pfs) (p,ins,outs) ∈ set proc
  (pred (kind a) s) (pred (kind ax) s’)
have ∀ i < length ins. (params fs (fst (hd s)))!i = (params fs (fst (hd s’)))!i
  by (fastforce intro!: CFG-call-edge-params)
from (valid-edge a) (kind a = Q:r→pfs) (p,ins,outs) ∈ set proc
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 }
hence ∀ i < length fs. (fs ! i) (fst cf) = (fs ! i) (fst cf’) by simp
{ fix i assume i < length fs
  with ∀ i < length fs. (fs ! i) (fst cf) = (fs ! i) (fst cf’)
  have (fs ! i) (fst cf) = (fs ! i) (fst cf’) by simp
  have ((csppa (targetnode a) (HRB-slice S) 0 fs)!i)(fst cf) =
    ((csppa (targetnode a) (HRB-slice S) 0 fs)!i)(fst cf’)
  proof (cases Formal-in (targetnode a, i + 0) ∈ HRB-slice S)
  case True
    with (i < length fs)
    have (csppa (targetnode a) (HRB-slice S) 0 fs)!i = fs!i
      by (rule csppa-Formal-in-in-slice)
    with (fs ! i) (fst cf) = (fs ! i) (fst cf’): show ⋆thesis by simp
  next
  case False
    with (i < length fs)
    have (csppa (targetnode a) (HRB-slice S) 0 fs)!i = empty
      by (rule csppa-Formal-in-notin-slice)
    thus ⋆thesis by simp
  qed }
hence $\forall i < \text{length } fs.
((\text{cspp} \text{ (targetnode a)} \text{ (HRB-slice S) } fs)!i)!\text{fst cf} =
((\text{cspp} \text{ (targetnode a)} \text{ (HRB-slice S) } fs)!i)!\text{fst cf}')
by(simp add:cspp-def)

\{ fix i assume i < \text{length } fs
hence (\text{params} \text{ (cspp (targetnode a)} \text{ (HRB-slice S) } fs)
(\text{fst cf}))!i =
((\text{cspp} \text{ (targetnode a)} \text{ (HRB-slice S) } fs)!i)(\text{fst cf}')
by(\text{fastforce intro:params-nth})
\}

moreover
from (i < \text{length } fs)
have (\text{params} \text{ (cspp (targetnode a)} \text{ (HRB-slice S) } fs)
(\text{fst cf}')!i =
((\text{cspp} \text{ (targetnode a)} \text{ (HRB-slice S) } fs)!i)(\text{fst cf}')
by(\text{fastforce intro:params-nth})

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

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

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

obtain cfz where [simp]:
  \text{transfer} \text{ (slice-kind S a)} \text{ (cf # cfs) = cfz # cf # cfs}
  \text{transfer} \text{ (slice-kind S a)} \text{ (cf' # cfs') = cfz # cf' # cfs'}
  \text{by \text{auto}}

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

from \text{res} \text{ (i < \text{length } fs) \text{ (source} \text{node a = m)}

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

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

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

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

16
moreover
from\langle\text{length } s' = \text{Suc (length } cs)\rangle
have\langle\text{length (transfer (slice-kind } S \ a) \ s') = \text{Suc (length } (a \# \ cs))\rangle\text{ by simp}
moreover
from \text{IH} \langle\forall \langle\text{OF \ \langle\text{upd-cs } (a \# \ cs) \ \text{as = } []\rangle \ \langle\text{same-level-path-aux } (a \# \ cs) \ \text{asx}\rangle \ \forall \langle\text{c \ \text{set } cs' \ \\text{valid-edge } c \ \langle\text{targetnode } a - as - \rightarrow \ (<\text{Low}-)\rangle \ \langle\text{targetnode } a - as - \rightarrow \ (<\text{Low}-)\rangle \ \text{rvs' } \\text{snds rv calculation}\rangle \ \langle\text{as'} = ax\#axs\rangle\rangle\rangle\text{ show } ?\text{thesis by (simp add: slice-kinds-def) qed qed
next
case (\text{slpa-Return } cs \ a \ \text{as } Q \ p \ \text{f } c' \ \text{cs'})
note \text{IH = }\langle\forall m \ as' \ s' \ [\text{upd-cs } cs' \ \text{as = } []\rangle \ \langle\text{same-level-path-aux } cs' \ \text{as'}\rangle\rangle
\forall c \ \text{set } cs', \ \text{valid-edge } c; m - as - \rightarrow \ (<\text{Low}-); m - as' - \rightarrow \ (<\text{Low}-);\ \forall i < \text{length } cs', \ \forall V \ \in \text{rv } S \ \langle\text{CFG-node } (\text{source-node } (cs \ i) i)\rangle.
\text{fst} (s \ ! \ Suc i) V = \text{fst} (s' \ ! \ Suc i) V;
\forall i < \text{Suc (length } cs') \ \text{snd} (s \ i) = \text{snd} (s' \ i);
\forall V \ \in \text{rv } S \ \langle\text{CFG-node } m\rangle, \ \text{state-val } s V = \text{state-val } s' V;
\text{preds (slice-kinds } S \ \text{as} \ s) \ \langle\text{preds (slice-kinds } S \ \text{as'} \ \text{s')}\rangle \ \langle\text{length } s = \text{Suc (length } cs'); \ \text{length } s' = \text{Suc (length } cs')\rangle \ \Rightarrow \ \forall V \ \in \text{Use} \ (<\text{Low}-), \ \text{state-val (transfers (slice-kinds } S \ \text{as}) \ s) V = \text{state-val (transfers (slice-kinds } S \ \text{as'}) \ s') V'\rangle
note \text{rvs = }\langle\forall i < \text{length } cs, \ \forall V \ \in \text{rv } S \ \langle\text{CFG-node } (\text{source-node } (cs \ i) i)\rangle\rangle.
\text{fst} (s \ ! \ Suc i) V = \text{fst} (s' \ ! \ Suc i) V
from (m - a \# \ as - \rightarrow \ (<\text{Low}-)) \ \text{have source-node } a = m \ \text{and valid-edge } a \ \text{and target-node } a - as - \rightarrow \ (<\text{Low}-) \ \text{by (auto elim: path-split-Cons)}
from \\langle\forall c \ \text{set } cs, \ \text{valid-edge } c \ \langle\text{cs = c' \# cs'}\rangle\rangle
\text{have valid-edge } c' \ \text{and } \ \forall c \ \text{set } cs'. \ \text{valid-edge } c \ \text{by simp-all show } ?\text{case proof (cases } L = \{\}\rangle
\text{case True with UseLow show } ?\text{thesis by simp}
next
case False
show ?\text{thesis proof (cases } as'\rangle
\text{case Nil with } (m - as' - \rightarrow \ (<\text{Low}-)) \ \text{have } m = (<\text{Low}-) \ \text{by fastforce with } \langle\text{valid-edge } a \ \langle\text{source-node } a = m\rangle \ \text{have target-node } a = (<\text{Exit}-)\rangle \ \text{by -(rule Exit-successor-of-Low,simp+)}
from Low-source-Exit-edge obtain a' \ \text{where valid-edge } a' \ \text{and source-node } a' = (<\text{Low}-) \ \text{and target-node } a' = (<\text{Exit}-)\rangle \ \text{and kind } a' = (\lambda s. \ True) \ \text{by blast from Low-source-Exit-edge obtain a' \ where valid-edge } a' \ \text{and source-node } a' = (<\text{Low}-) \ \text{and target-node } a' = (<\text{Exit}-)\rangle \ \text{and kind } a' = (\lambda s. \ True) \ \text{by blast from } (\langle\text{valid-edge } a \ \langle\text{source-node } a = m\rangle \ \langle\text{m = (<Low>)}\rangle \ \langle\text{target-node } a = (<\text{Exit}-)\rangle \ \langle\text{valid-edge } a' \ \langle\text{source-node } a' = (<\text{Low}-)\rangle \ \langle\text{target-node } a' = (<\text{Exit}-)\rangle \ \text{have } a = a' \ \text{by (fastforce dest: edge-det)}\rangle
\text{with } (\langle\text{kind } a' = (\lambda s. \ True)\rangle \ \text{have kind } a = (\lambda s. \ True) \ \text{by simp with } (\langle\text{target-node } a = (<\text{Exit}-)\rangle \ \langle\text{target-node } a - as - \rightarrow \ (<\text{Low}-)\rangle\rangle}
have \((-\text{Low})\) = \((-\text{Exit})\) by \(-(\text{drule path-Exit-source,auto})\)

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

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

next
case \(\langle\text{Cons ax asx}\rangle\)
  with \(\langle m \sim as'\to as\rangle\) have sourcenode ax = m and valid-edge ax
  and targetnode ax = asx \(-\text{asx}\to as\) \(\langle\text{-Low}\rangle\) by \(\text{(auto elim:path-split-Cons)}\)
  from \(\langle\text{valid-edge a \#valid-edge ax}\rangle\) \(\langle\text{kind a = Q}\to pf\rangle\)
  \(\langle\text{sourcenode a = m}\rangle\) \(\langle\text{sourcenode ax = m}\rangle\)
  have \(\exists J f. \text{kind ax = Q}\to pf\) by \(\text{(auto dest:return-edges-only)}\)
  with \(\langle\text{same-level-path-axv cs asx}\rangle\) \(\langle\text{asx = ax}\#asx\rangle\) \(\langle\text{cs = c'}\# cs'\rangle\)
  have ax \(\in\) get-return-edges c' and same-level-path-axv cs' asx by auto
  from \(\langle\text{valid-edge c'}\rangle\) \(\langle\text{ax \in get-return-edges c'}\rangle\)
  \(\langle\text{a \in get-return-edges c'}\rangle\)
  have \(\langle\text{simp}\rangle ax = a by (\text{rule get-return-edges-unique} )\)
  from \(\langle\text{targetnode ax \sim asx}\to as\rangle\) \(\langle\text{-Low}\rangle\) have targetnode a = asx \(-\text{asx}\to as\) \(\langle\text{-Low}\rangle\)
by simp
  from \(\langle\text{upd-cs cs a \# as}\rangle = []\) \(\langle\text{kind a = Q}\to pf\rangle\)
  \(\langle\text{cs = c'}\# cs'\rangle\)
  \(\langle\text{a \in get-return-edges c'}\rangle\)
  have \(\langle\text{upd-cs cs a \# as}\rangle = []\) by simp
  from \(\langle\text{length s = Succ (length cs)}\rangle\)
  \(\langle\text{cs = c'}\# cs'\rangle\)
  obtain \(\langle\text{cf cf}x\rangle\) \(\langle\text{cfs where s = cf}\# cf\# cf\# cf\}
    by \(\langle\text{cases s,auto,case-tac list fastforce+}\rangle\)
  from \(\langle\text{length s'} = Succ (length cs)\rangle\)
  \(\langle\text{cs = c'}\# cs'\rangle\)
  obtain \(\langle\text{cf cf}x'\rangle\) \(\langle\text{cfs' where s' = cf'}\# cf'\# cf'\# cf'\}
    by \(\langle\text{cases s',auto,case-tac list fastforce+}\rangle\)
  from \(\langle\text{res1: } \forall i < \text{length cs} \} .\)
  \(\forall V \in \text{rv S (CFG-node (sourcenode (cs' ! i)))}\).
  \(\text{fst } ((\text{cf}x\# cf\# cf\# cf) \sim \text{Succ i}) \sim \text{V} = \text{fst} ((\text{cf}x'\# cf'\# cf'\# cf') \sim \text{Succ i}) \sim \text{V}\)
  and \(\forall V \in \text{rv S (CFG-node (sourcenode c'))}\).
  \(\text{fst cf}x\} \sim \text{V} = \text{fst cf}x'\} \sim \text{V}\)
  by auto
  from \(\langle\text{valid-edge c'}\rangle\) \(\langle\text{a \in get-return-edges c'}\rangle\)
  obtain \(Q x px fsx where\) kind \(c' = Q x: rv\to px: fsx\)
    by \(\text{(fastforce dest!-only-call-get-return-edges)}\)
  have \(\forall V \in \text{rv S (CFG-node (targetnode a))}\).
  \(\text{V} \in \text{rv S (CFG-node (sourcenode c'))}\)
proof
  fix \(V\) assume \(V \in \text{rv S (CFG-node (targetnode a))}\)
  from \(\langle\text{valid-edge c'}\rangle\) \(\langle\text{a \in get-return-edges c'}\rangle\)
  obtain \(a'\ where\) edge:valid-edge \(a'\ sourcenode \(a' = sourcenode\ c'\)
    targetnode \(a' = targetnode\ a\)\ intra-kind\ (kind \(a')\)
    by \(\sim (\text{drule call-return-node-edge,auto simp:intra-kind-def})\)
  from \(\langle\text{V} \in \text{rv S (CFG-node (targetnode a))}\rangle\)
  obtain \(a'\ where\) targetnode\ \(a\ to asx\)\ parent-node\ \(n'\)
    and \(n' \in \text{HRB-slice S and} V \in \text{Use}_{\text{SDG}} n'\)
    and \(\forall n''\ \text{valid-SDG-node} n''\ and\ \text{parent-node} n''\ \in\ \text{set (sourcenodes as)}\)
    \(\rightarrow V \notin \text{Def}_{\text{SDG}} n''\) by \(\text{fastforce elim:rvE}\)
from (targetnode a, \mathbin{-\rightarrow,^*} parent-node n') edge
have source-node c' = a'\mathbin{\#-\rightarrow,^*} parent-node n'
  by (fastforce intro: Cons-path simp: intra-path-def)
from (valid-edge c') (kind c' = \text{Q\_expression}) have Def (source-node c') = 
  
  \{ 
  \}
  by (rule call-source-Def-empty)
hence \forall n''. valid-SDG-node n'' \wedge parent-node n'' = source-node c'
  \rightarrow V \notin \text{Def}_{SDG} n'' \text{ by (fastforce dest: SDG-Def-parent-Def)}
with all (source-node a' = source-node c')
have \forall n''. valid-SDG-node n'' \wedge parent-node n'' \in \text{set} (source-nodes (a'\#as))
  
  \rightarrow V \notin \text{Def}_{SDG} n'' \text{ by (fastforce simp: source-nodes-def)}
with (source-node c' = a'\#as \mathbin{-\rightarrow,^*} parent-node n')
(n' \in \text{HRB-slice } S \setminus V \in \text{Use}_{SDG} n')
show V \in rv S (CFG-node (source-node c'))
  by (fastforce intro: rvI)
qed
show \theta
proof (cases source-node a \in [HRB-slice S]_{CFG})
case True
from (valid-edge c') (a \in \text{get-return-edges } c')
have get-proc (targetnode c') = get-proc (source-node a)
  by -(drule intra-proc-additional-edge,
  auto dest: get-proc-intra simp: intra-kind-def)
moreover
from (valid-edge c') (kind c' = Q\_expression \mathbin{\rightarrow,^*} px fsx)
have get-proc (targetnode c') = px \text{ by (rule get-proc-call)}
moreover
from (valid-edge a; (kind a = \text{Q\_expression}))
have get-proc (source-node a) = p \text{ by (rule get-proc-return)}
ultimately have \text{[simp]} px = p \text{ by simp}
from (valid-edge c') (kind c' = Q\_expression \mathbin{\rightarrow,^*} px fsx)
obtain ins outs where (p, ins, outs) \in \text{set proces}
  \text{by (fastforce dest!: callee-in-procs)}
with (source-node a \in [HRB-slice S]_{CFG})
  (valid-edge a; (kind a = \text{Q\_expression}))
have slice-kind: slice-kind S a =
  Q \leftarrow p (\lambda cf c' , rspp (targetnode 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 (targetnode a) (HRB-slice S) outs (fst cfx) (fst cf),
  snd cfx) \# cfs
  and sx': transfer (slice-kind S a) s' =
  (rspp (targetnode a) (HRB-slice S) outs (fst cfx') (fst cf'),
  snd cfx') \# cfs'
  by simp-all
with \text{rvI} have \text{rvI} ! \forall i < \text{length } cs'.
  \forall V \in rv S (CFG-node (source-node (cs' ! i))).
\[
\text{fst } ((\text{transfer } (\text{slice-kind } S \ a) \ s) ! \ Suc \ i) \ V = \\
\text{fst } ((\text{transfer } (\text{slice-kind } S \ a) \ s') ! \ Suc \ i) \ V \\
\text{by fastforce}
\]

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

\[
\langle s = cf#cfx#cfs \ association \rangle \langle s' = cf'#cfx'#cfs' \ association \rangle \\
\text{have} \ \text{snds} : \forall i < \text{Suc (length } cs'). \\
\text{snd} \ (\text{transfer } (\text{slice-kind } S \ a) \ s ! i) = \\
\text{snd} \ (\text{transfer } (\text{slice-kind } S \ a) \ s' ! i) \\
\text{apply} \ \text{auto} \ \text{apply} (\text{case-tac } i) \ \text{apply} \ \text{auto} \\
\text{by} (\text{erule-tac } x = \text{Suc (Suc nat) in allIE} \ \text{auto})
\]

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

\text{proof}

\text{fix} \ V \ \text{assume} \ V \in rv S \ (\text{CFG-node } (\text{targetnode } a)) \\
\text{show} \ (\text{rspp } (\text{targetnode } a) \ (\text{HRB-slice } S) \ \text{outs} \\
(\text{fst cfx}) \ (\text{fst cf})) \ V = \\
(\text{rspp } (\text{targetnode } a) \ (\text{HRB-slice } S) \ \text{outs} \\
(\text{fst cfx}') \ (\text{fst cf}') \ V)
\]

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

\text{case True}

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

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

\text{show } \ ? \text{thesis}
\]

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

\text{case True}

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

\text{have} \ (\text{rspp-eq }): (\text{rspp } (\text{targetnode } a) \\
(\text{HRB-slice } S) \ \text{outs} \ (\text{fst cfx}) \ (\text{fst cf})) \ V = \\
(\text{fst cf})(\text{outs} i) \\
(\text{rspp } (\text{targetnode } a) \\
(\text{HRB-slice } S) \ \text{outs} \ (\text{fst cfx}') \ (\text{fst cf}') \ V = \\
(\text{fst cf}')(\text{outs} i)) \\
\text{by (auto intro: rspp-Actual-out-in-slice)}

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

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

\text{proof}

\text{fix} \ V \ \text{assume} \ V \in \text{Use } (\text{source node } a) \\
\text{from} \ (\text{valid-edge } a) \ (\text{source node } a = m)
have parent-node (CFG-node m) \to (-\to,*, parent-node (CFG-node m))
  by (fastforce intro:empty-path simp:intra-path-def)
with \langle source-node a \in [HRB-slice S]_{CFG} \rangle
\langle V \in \text{Use (source-node a)} \rangle \langle \text{source-node} a = m \rangle \langle \text{valid-edge} a \rangle
show V \in rv S (CFG-node m)
  by -(rule ref, 
  auto intro!:CFG-Use-SDG-Use simp:SDG-to-CFG-set-def
sourcenodes-def)
qed
with \langle \forall V \in set outs. V \in \text{Use (source-node} a) \rangle
have \langle \forall V \in set outs. V \in rv S (CFG-node m) \rangle by simp
with \langle \forall V \in rv S (CFG-node m). state-val s V = state-val s' V \rangle
\langle s = cf\#cfx\#cfs \rangle \langle s' = cf'\#cfx'\#cfs' \rangle
have \langle \forall V \in set outs. (fst cf) V = (fst cf') V \rangle by simp
with \langle i < length (ParamDefs (target-node a)) \rangle
\langle length(ParamDefs (target-node a)) = length outs \rangle
have \langle (fst cf)(outs!i) = (fst cf')(outs!i) \rangle by fastforce
with rspp-eq show ?thesis by simp
next
case False
with \langle i < length (ParamDefs (target-node a)) \rangle \langle valid-edge a \rangle
\langle length(ParamDefs (target-node a)) = length outs \rangle
\langle (ParamDefs (target-node a))!i = V :[THEN sym] \rangle
have rspp-eq:(rspp (target-node a))
\langle (HRB-slice S) outs (fst cfx)(fst cf) V = \rangle
\langle (fst cfx)(ParamDefs (target-node a))!i \rangle
\langle rspp (target-node a) \rangle
\langle (HRB-slice S) outs (fst cfx')(fst cf') V = \rangle
\langle (fst cfx')(ParamDefs (target-node a))!i \rangle
by (auto intro!:rspp-Actual-out-notin-slice)
from \langle \forall V \in rv S (CFG-node (source-node c')) \rangle.
\langle (fst cfx) V = (fst cfx') V \rangle
\langle V \in rv S (CFG-node (target-node a)) \rangle
\langle \forall V \in rv S (CFG-node (target-node a)) \rangle
\langle V \in rv S (CFG-node (source-node c')) \rangle
\langle (ParamDefs (target-node a))!i = V :[THEN sym] \rangle
have \langle (fst cf\#cfx)(ParamDefs (target-node a) ! i) = \rangle
\langle (fst cf')(ParamDefs (target-node a) ! i) \rangle by fastforce
with rspp-eq show ?thesis by fastforce
qed
next
case False
with \langle \forall V \in rv S (CFG-node (source-node c')) \rangle.
\langle (fst cfx) V = (fst cfx') V \rangle
\langle V \in rv S (CFG-node (target-node a)) \rangle
\langle \forall V \in rv S (CFG-node (target-node a)) \rangle
\langle V \in rv S (CFG-node (source-node c')) \rangle
show ?thesis by (fastforce simp:rspp-def map-merge-def)
qed
qed
with sx.sx'
have \( rv \vdash V \in rv S \) (CFG-node (targetnode a)),
  state-val (transfer (slice-kind S a) s) V =
  state-val (transfer (slice-kind S a) s') V
  by fastforce
from (preds (slice-kinds S (a \# as)) s)
have preds (slice-kinds S as)
  (transfer (slice-kind S a) s)
  by (simp add: slice-kinds-def)
moreover
from (preds (slice-kinds S as') s' \( \langle as' = ax#as' \rangle \))
have preds (slice-kinds S asx)
  (transfer (slice-kind S a) s')
  by (simp add: slice-kinds-def)
moreover
from \( \langle \text{length } s = \text{Suc } \langle \text{length } cs \rangle \rangle \langle cs = c' \# cs' \rangle \) sx
have length (transfer (slice-kind S a) s) = \( \text{Suc } \langle \text{length } cs' \rangle \)
  by (simp simp add: \( s = cf#cfx#cfs \))
moreover
from \( \langle \text{length } s' = \text{Suc } \langle \text{length } cs \rangle \rangle \langle cs = c' \# cs' \rangle \) sx'
have length (transfer (slice-kind S a) s') = \( \text{Suc } \langle \text{length } cs' \rangle \)
  by (simp simp add: \( s' = cf'#cfx'#cfs' \))
moreover
from IH [OF \( \langle \text{upd-cs } cs' \rangle \langle \text{same-level-path-aux } cs' \ asx' \rangle \)
  \( \forall c \in \text{set } cs' \), valid-edge c] (targetnode a \( \to as \to* \) (-Low-))
  (targetnode a \( \to -as \to* \) (-Low-)) res' snds rv' calculation \( \langle as' = ax#as' \rangle \)
show \( \text{thesis by (simp add: slice-kinds-def)} \)
next
case False
from this \( \langle \text{kind } a = Q \in p \rangle \)
have slice-kind: slice-kind S a = (\( \lambda cf. \text{True} \))\( \in p \langle \lambda cf cf'. cf' \rangle \)
  by (rule slice-kind-Return)
with \( \langle s = cf#cfx#cfs \rangle \langle s' = cf'#cfx'#cfs' \rangle \)
have [simp]: transfer (slice-kind S a) s = cfx#cfs
  transfer (slice-kind S a) s' = cfx'#cfs' by simp-all
from slice-kind \( \forall i < \text{Suc } \langle \text{length } cs \rangle \).
  snd \( \langle s ! i \rangle \) = snd \( \langle s' ! i \rangle \)
  \( \langle cs = c' \# cs' \rangle \)
  \( \langle s = cf#cfx#cfs \rangle \)
  \( \langle s' = cf'#cfx'#cfs' \rangle \)
have snds: \( \forall i < \text{Suc } \langle \text{length } cs \rangle \).
  snd (transfer (slice-kind S a) s ! i) =
  snd (transfer (slice-kind S a) s' ! i) by fastforce
from rv1 have rv2 \( \forall i < \text{length } cs' \).
  \( \forall V \in rv S \) (CFG-node (source-node (cs' ! i))).
  \( \text{fst } \langle \text{transfer } \langle \text{slice-kind } S a \rangle s ! \text{Suc } i \rangle V =
  \text{fst } \langle \text{transfer } \langle \text{slice-kind } S a \rangle s' ! \text{Suc } i \rangle V \)
  by fastforce
from \( \forall V \in rv S \) (CFG-node (targetnode a))
  \( V \in rv S \) (CFG-node (source-node c'))
\( \forall V \in rv S \) (CFG-node (source-node c')).
22
\[(\text{fst } cf) \ V = (\text{fst } cf') \ V\]

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

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

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

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

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

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

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

**moreover**

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

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

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

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

**moreover**

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

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

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

**moreover**

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

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

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

**moreover**

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

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

**qed**

**qed**

**lemma** rv-Low-Use-Low:

**assumes** \(m \rightarrow as \rightarrow \# \rightarrow \langle -\text{Low-}\rangle \text{ and} \ m \rightarrow as' \rightarrow \# \rightarrow \langle -\text{Low-}\rangle \text{ and} \text{ get-proc} \ m = \text{Main} \)

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

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

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

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

**shows** \(\forall V \in \text{Use} \ (-\text{Low-})\).

\(\text{state-val} \ (\text{transfers} \ (\text{slice-kinds} \ S \ as) \ [(cf', \text{undefined})]) \ V = \)

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

**proof** (cases as)

**case** Nil

**with** \(\langle m \rightarrow as \rightarrow \# \rightarrow \langle -\text{Low-}\rangle \rangle\) **have** \(\text{valid-node} \ m \text{ and} \ m = \langle -\text{Low-}\rangle\)

**by** (auto intro:path-valid-node simp:vp-def)

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

**moreover**

**from** \(\langle \text{valid-node} \ m \rangle \langle m = \langle -\text{Low-}\rangle \rangle\) **have** \(\langle -\text{Low-} \rangle \rightarrow [] \rightarrow \langle -\text{Low-}\rangle\)

**by** (fastforce intro:empty-path simp:intra-path-def)

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

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

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

proof (induct \(m\) \(\text{as'}\) \(\text{m} '\equiv (\text{-Low})\) rule: path.induct)
  case (Cons-path \(m''\) as \(m\))
  from \(\langle \text{valid-edge} \ \alpha \rangle \ \langle \text{sourcenode} \ a = m \rangle \ (m = (\text{-Low}))\)
  have \(\text{targetnode} \ a = (\text{-Exit})\) by (rule Exit-successor-of-Low, simp+)
  with \(\langle \text{targetnode} \ a = m'' \rangle \ (m'' - \text{as'} \rightarrow_{\gamma}^{\star} (\text{-Low}))\)
  have \((\text{-Low}) = (\text{-Exit})\) by (drule path-Exit-source, auto)
  with \(\text{False}\) have \(\text{False}\) by (drule Low-neq-Exit, simp)
  thus \(?case\) by simp
qed

next
  case (Cons \(\alpha\) \(\text{as'}\))
  have \(\langle m - \text{as'} \rightarrow_{\gamma}^{\star} (\text{-Low}) \rangle\) \(\text{sourcenode} \ \alpha = m\) \(\text{and valid-edge} \ \alpha\)
  and \(\text{targetnode} \ \alpha - \text{as'} \rightarrow_{\gamma}^{\star} (\text{-Low})\)
  by (auto elim: path-split-Cons simp: vp-def)
show \(?thesis\)
proof (cases \(L = \{\}\))
  case True with UseLow show \(?thesis\) by simp
next
  case False
show \(?thesis\)
proof (cases \(\text{as'}\))
  case Nil
  have \(\langle m - \text{as'} \rightarrow_{\gamma}^{\star} (\text{-Low}) \rangle\) \(m = (\text{-Low})\) by (fastforce simp: vp-def)
  have \(\langle \text{valid-edge} \ \alpha \rangle\) \(\langle \text{sourcenode} \ \alpha = m \rangle\) \(\text{and targetnode} \ \alpha = (-\text{Exit})\)
  by (rule Exit-successor-of-Low, simp+)
  from \(\text{Low-source-Exit-edge}\) obtain \(\alpha'\) where \(\text{valid-edge} \ \alpha'\)
    and \(\text{sourcenode} \ \alpha' = (\text{-Low})\) \(\text{and targetnode} \ \alpha' = (\text{-Exit})\)
    and \(\text{kind} \ \alpha' = (\lambda S. \text{True})\) by blast
  from \(\langle \text{valid-edge} \ \alpha \rangle\) \(\langle \text{sourcenode} \ \alpha = m \rangle\) \(m = (\text{-Low})\)
  \(\langle \text{targetnode} \ \alpha = (\text{-Exit})\rangle\) \(\langle \text{valid-edge} \ \alpha' \rangle\) \(\langle \text{sourcenode} \ \alpha' = (\text{-Low})\rangle\)
  \(\langle \text{targetnode} \ \alpha' = (\text{-Exit})\rangle\)
have \( ax = a' \) by (fastforce dest:edge-det)

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

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

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

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

thus ?thesis by simp

next

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

from \((m \rightarrow as\rightarrow \rightarrow\) \(\vdash \) have \(\text{valid-path-aux} \[] \text{ as and } m \rightarrow as\rightarrow\) \(\vdash \) (Low-)

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

from this \(\vdash \) (as = ax#'asx') (get-proc \( m \) = Main)

have \(\text{same-level-path-aux} \[] \text{ as \& upd-cs \[] as \=} \[

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

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

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

from \((m \rightarrow as\rightarrow \rightarrow\) \(\vdash \) have \(\text{valid-path-aux} \[] \text{ as and } m \rightarrow as\rightarrow\) \(\vdash \) (Low-)

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

from this \(\vdash \) (as' = ax#'asx') (get-proc \( m \) = Main)

have \(\text{same-level-path-aux} \[] \text{ as' and upd-cs \[] as' = \[]}

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

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

hence \(\text{same-level-path-aux} \[] \text{ as' by simp}

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

\(\vdash \) (same-level-path-aux \[] \text{ as'}) \((m \rightarrow as\rightarrow\) \(\vdash \) (Low-), \((m \rightarrow as\rightarrow\) \(\vdash \) (Low-))

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

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

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

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

qed

qed

lemmas \(\text{nonInterference-path-to-Low}:

\text{assumes } \text{[cf]} \approx L \text{[cf] and } \text{[High]} \notin \text{[HRB} \text{slice } S \text{]} \text{CFG}

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

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

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

\text{shows } \text{map } \text{fst } \text{(transfers } \text{(kinds } as\text{)} \[(\text{cf}, \text{undefined})]\) \approx L

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

proof

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

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

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

\(\text{and } \forall V \in \text{Use } \text{[Low-].}

\text{state-val } \text{(transfers } \text{(slice-kinds } S \text{ ax}) \[(\text{cf}, \text{undefined})]\) \text{ V = state-val } \text{(transfers } \text{(kinds } as\text{)} \[(\text{cf}, \text{undefined})]\) \text{ V}

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

theorem nonInterference-path:
assumes \[\cf] \cong_L [\cf'] \notin [HRB-slice S]_{CFG}
and \CFG-node (-Low-) ∈ S
and (-Entry-) → as' → * (-Exit-) \preds (kinds as) [(cf, undefined)]
and (-Entry-) → as' → * (-Exit-) \preds (kinds as') [(cf', undefined)]
shows map \fst (transfers (kinds as) [(cf, undefined)]) \cong_L
map \fst (transfers (kinds as') [(cf', undefined)])
proof –
from (-Entry-) → as' → * (-Exit-) obtain x xs where as = x#xs
and (-Entry-) = sourcenode x and valid-edge x
and targetnode x = x→* (-Exit-)
apply (cases as = [])
apply (clarsimp simp:up-def, drule empty-path-nodes, drule Entry-noteq-Exit, simp)
by (fastforce elim: path-split-Cons simp vp-def)
from (- Entry) - as → √* (- Exit) have valid-path as by (simp add: vp-def)
from (valid-edge x) have valid-node (targetnode x) by simp
hence inner-node (targetnode x)
proof (cases rule: valid-node-cases)
case Entry
with (valid-edge x) have False by (rule Entry-target)
thus ?thesis by simp
next

case Exit
with (targetnode x - xs →* (- Exit)) have xs = []
by -(drule path-Exit-source_auto)
from Entry-Exit-edge obtain z where valid-edge z
and sourcenode z = (- Entry) and targetnode z = (- Exit)
and kind z = (∀ s. False) ∨ by blast
from (valid-edge x) (valid-edge z) (- Entry) = sourcenode x
(sourcenode z = (- Entry)) Exit (targetnode z = (- Exit))
have x = z by (fastforce intro: edge-det)
with (preds (kinds as) \{[cf, undefined]\}) \{as = x # xs\} \{xs = []\}
-kind z = (∀ s. False) ∨
have False by (simp add: kinds-def)
thus ?thesis by simp
qed simp

with (targetnode x - xs →* (- Exit)) obtain x' xs' where xs = xs'@[x']
and targetnode x - xs' →* (- Low) and kind x' = (∀ s. True) ∨
by (fastforce elim: path-Exit-Low-path)
with (- Entry) = sourcenode x (valid-edge x)
have (- Entry) - x # xs' →* (- Low) by (fastforce intro: Cons-path)
from (valid-path as) \{as = x # xs\} \{xs = xs'@[x']\}
have valid-path (x # xs')
by (simp add: valid-path-def del: valid-path-aux.simps)
(rule valid-path-aux-split, simp)
with (- Entry) - x # xs' →* (- Low) have (- Entry) - x # xs' → √* (- Low)
by (simp add: vp-def)
from \{as = x # xs\} \{xs = xs'@[x']\} have as = (x # xs')@[x'] by simp
with \{preds (kinds as) \{[cf, undefined]\}\}
have \{preds (kinds (x # xs')) \{[cf, undefined]\}\}
-by (simp add: kinds-def preds-split)
from (- Entry) - as' → √* (- Exit) obtain y ys where as' = y # ys
and (- Entry) = sourcenode y and valid-edge y
and targetnode y - ys →* (- Exit)
apply \{cases as' = []\}
apply (clarsimp simp: vp-def, drule empty-path-nodes, drule Entry-noteq-Exit-simp)
by (fastforce elim: path-split-Cons simp: vp-def)
from (- Entry) - as' → √* (- Exit) have valid-path as' by (simp add: vp-def)
from (valid-edge y) have valid-node (targetnode y) by simp
hence inner-node (targetnode y)
proof (cases rule: valid-node-cases)
case Entry
with \(\text{valid-edge } y\) have False by (rule Entry-target)
thus \(\forall \text{thesis by simp}\)

next

-case Exit
  with (targetnode y - ys \rightarrow^* \text{(-Exit-)} ) have ys = []
  by (drule path-Exit-source, auto)
from Entry-Exit-edge obtain z where valid-edge z
  and sourcenode z = (\text{-Entry-}) and targetnode z = (\text{-Exit-})
  and kind z = (\text{\(\lambda s.\) False}) by blast
from (valid-edge y) (valid-edge z) (\text{-Entry-}) = sourcenode y
(sourcenode z = (\text{-Entry-}) ) Exit (targetnode z = (\text{-Exit-}))
  have y = z by (fastforce intro:edge-det)
with \(\text{preds (kinds } \text{as}') \quad [(\text{cf}', undefined)]\) \(\text{as}' = y \# y\text{s} \quad \text{ys} = []\)
  have False by (simp add: kinds-def)
thus \(\forall \text{thesis by simp}\)
qed

simp

with (targetnode y - ys \rightarrow^* \text{(-Exit-)} ) obtain y' y's where \(\text{ys} = \text{ys}' @ [y']\)
  and targetnode y - ys' \rightarrow^* \text{(-Low-)} and kind y' = (\text{\(\lambda s.\) True})
  by (fastforce elim: Exit-path-Low-path)

with \text{(-Entry-)} = sourcenode y \quad \text{valid-edge } y
  have (\text{-Entry-}) - y \# y\text{s}' \rightarrow^* \text{(-Low-)} by (fastforce intro: Cons-path)
from \text{valid-path } \text{as} \quad \text{as}' = y \# y\text{s}' \quad \text{ys} = \text{ys}' @ [y']
  have valid-path (y \# y\text{s}')
  by (simp add: valid-path-def del: valid-path-aux.simps)
     (rule valid-path-aux-split, simp)

with \text{(-Entry-)} - y \# y\text{s}' \rightarrow^* \text{(-Low-)} have \text{(-Entry-)} - y \# y\text{s}' \rightarrow^* \text{(-Low-)}
  by (simp add: vp-def)
from \text{as}' = y \# y\text{s}' \quad \text{ys} = \text{ys}' @ [y'] have as' = (y \# y\text{s}') @ [y'] by simp
with \text{preds (kinds } \text{as}') \quad [(\text{cf}', undefined)]
  have \text{preds (kinds } \text{y} \# y\text{s}') \quad [(\text{cf}', undefined)]
  by (simp add: kinds-def preds-split)
from [(\text{cf}) \approx_L (\text{cf}')] \quad \text{[HRB-slice } S \text{]} \quad \text{CFG'} / \quad \text{CFG-node (\text{\text{(-Low-)} } )} \in S
  \text{(-Entry-)} - x \# x\text{s}' \rightarrow^* \text{(-Low-)} \text{preds (kinds } x \# x\text{s}') \quad [(\text{cf}, undefined)]
  \text{(-Entry-)} - y \# y\text{s}' \rightarrow^* \text{(-Low-)} \text{preds (kinds } y \# y\text{s}') \quad [(\text{cf}', undefined)]
  map fst (transfers (kinds } x \# x\text{s}') \quad [(\text{cf}, undefined)] \approx_L
  map fst (transfers (kinds } y \# y\text{s}') \quad [(\text{cf}', undefined)]
by (rule nonInterference-path-to-Low)

with \text{as' = x \# x\text{s}' } \quad \text{xs = xs' @ [x']} \quad \text{kind } x' = (\text{\(\lambda s.\) True})
  \text{as' = y \# y\text{s} } \quad \text{ys = ys' @ [y']} \quad \text{kind } y' = (\text{\(\lambda s.\) True})
show \(\forall \text{thesis}\)
  apply (cases transfers (map kind xs') (transfer (kind } x) \quad [(\text{cf}, undefined)]))
  apply (auto simp add: kinds-def transfers-split)
  by ((cases transfers (map kind y' ) (transfer (kind } y) \quad [(\text{cf}', undefined)])), (auto simp add: kinds-def transfers-split)+
qed

end
locale NonInterferenceInter =
  NonInterferenceInterGraph source node target node kind valid-edge Entry
  get-proc get-return-edges pros Main Exit Def Use ParamDefs ParamUses
  H L High Low +
SemanticsProperty source node target node kind valid-edge Entry get-proc
  get-return-edges pros Main Exit Def Use ParamDefs ParamUses sem identifies
for source node :: 'edge ⇒ 'node and target node :: 'edge ⇒ 'node
and kind :: 'edge ⇒ ('var, 'val, 'ret, 'pname) edge-kind
and valid-edge :: 'edge ⇒ bool
and Entry :: 'node (''Entry'') and get-proc :: 'node ⇒ 'pname
and get-return-edges :: 'edge ⇒ 'edge set
and pros :: ('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] S1
and identifies :: 'node ⇒ 'com ⇒ bool (- ≡ - [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: [final c; n ≡ c]
⇒ ∃ a. valid-edge a ∧ source node a = n ∧ target node a = (-Low-) ∧ kind a =
↑id
begin

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

theorem nonInterference:
  assumes [cf₁] ≡ L [cf₂] and (-High-) ∉ | HRB-slice S | CFG
  and CFG-node (-Low-) ∈ S
  and valid-edge a and source node a = (-High-) and target node a = n
  and kind a = (As. True), and n ≡ c and final c'
  and (c₁[cf₁]) ⇒ ⟨c₁', s₁⟩ and (c₁[cf₂]) ⇒ ⟨c₁', s₂⟩
  shows s₁ ≡ L s₂
proof –
  from High-target-Entry-edge obtain az where valid-edge ax
  and source node ax = (-Entry-) and target node ax = (-High-)
  and kind ax = (As. True) by blast

29
\[
\begin{align*}
&\text{from } (n \triangleq c) \langle c, [cf_1] \rangle \Rightarrow \langle c', s_1 \rangle \\
&\text{obtain } n_1 \ as_1 \ cf_1 \ where \ n - as_1 \rightarrow^* n_1 \ \text{and} \ n_1 \triangleq c' \\
&\quad \text{and } \text{preds } (\text{kinds } as_1) \ [[cf_1, \text{undefined}]] \\
&\quad \text{and } \text{transfers } (\text{kinds } as_1) \ [[cf_1, \text{undefined}]] = \text{cf}_1 \ \text{and } \text{map fst cf}_1 = s_1 \\
&\quad \text{by} (\text{fastforce dest fundamental-property}) \\
&\text{from } (n - as_1 \rightarrow^* n_1) \ (\text{valid-edge } a) \ (\text{source-node } a = (-\text{High} -)) \ (\text{target-node } a = n) \\
&\quad \langle \text{kind } a = (\lambda s, \text{True}) \rangle \\
&\quad \text{have } (-\text{High} -) - a \# as_1 \rightarrow^* n_1 \ \text{by} (\text{fastforce intro: Cons-path simp: vp-def valid-path-def}) \\
&\quad \text{from } (\text{final } c') \ (n_1 \triangleq c') \\
&\quad \text{obtain } a_1 \ \text{where valid-edge } a_1 \ \text{and } \text{source-node } a_1 = n_1 \\
&\quad \text{and } \text{target-node } a_1 = (-\text{Low} -) \ \text{and } \text{kind } a_1 = \uparrow \text{id} \ \text{by} (\text{fastforce dest: final-edge-Low}) \\
&\quad \text{hence } n_1 - [a_1] \rightarrow^* (-\text{Low} -) \ \text{by} (\text{fastforce intro: path-edge}) \\
&\quad \text{with } (-\text{High} -) - a \# as_1 \rightarrow^* n_1 \ \text{have } (-\text{High} -) - (a \# as_1)@[a_1] \rightarrow^* (-\text{Low} -) \\
&\quad \quad \text{by} (\text{fastforce intro: path-Append simp: vp-def}) \\
&\quad \text{with } (\text{valid-edge } ax) \ (\text{source-node } ax = (-\text{Entry} -)) \ (\text{target-node } ax = (-\text{High} -)) \\
&\quad \text{have } (-\text{Entry} -) - ax\#((a \# as_1)@[a_1]) \rightarrow^* (-\text{Low} -) \ \text{by } (-\text{rule Cons-path}) \\
&\quad \text{moreover} \\
&\quad \text{from } (-\text{High} -) - a \# as_1 \rightarrow^* n_1 \ \text{have } \text{valid-path-aux } [] \ (a \# as_1) \\
&\quad \quad \text{by} (\text{simp add: vp-def valid-path-def}) \\
&\quad \quad \text{with } \langle \text{kind } a_1 = \uparrow \text{id} \rangle \ \text{have } \text{valid-path-aux } [] \ ((a \# as_1)@[a_1]) \\
&\quad \quad \quad \text{by} (\text{fastforce intro: valid-path-aux-Append}) \\
&\quad \quad \text{with } \langle \text{kind } ax = (\lambda s, \text{True}) \rangle \ \text{have } \text{valid-path-aux } [] \ (ax\#((a \# as_1)@[a_1])) \\
&\quad \quad \quad \text{by simp} \\
&\quad \quad \quad \text{ultimately have } (-\text{Entry} -) - ax\#((a \# as_1)@[a_1]) \rightarrow^* (-\text{Low} -) \\
&\quad \quad \quad \quad \text{by } (\text{simp add: vp-def valid-path-def}) \\
&\quad \text{from } (\text{valid-edge } a) \ (\text{kind } a = (\lambda s, \text{True}) \rangle \ (\text{source-node } a = (-\text{High} -)) \\
&\quad \quad \quad \langle \text{target-node } a = n \rangle \\
&\quad \text{have get-proc } n = \text{get-proc } (-\text{High} -) \\
&\quad \quad \quad \text{by } (\text{fastforce dest: get-proc-intra simp: intra-kind-def}) \\
&\quad \text{with get-proc-Low have get-proc } n = \text{Main} \ \text{by simp} \\
&\quad \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 = \uparrow \text{id}) \\
&\quad \text{have get-proc } n_1 = \text{get-proc } (-\text{Low} -) \\
&\quad \quad \text{by } (\text{fastforce dest: get-proc-intra simp: intra-kind-def}) \\
&\quad \text{with get-proc-Low have get-proc } n_1 = \text{Main} \ \text{by simp} \\
&\quad \text{from } (n - as_1 \rightarrow^* n_1) \ \text{have } n - as_1 \rightarrow^* n_1 \\
&\quad \quad \text{by } (\text{cases as}_1) \\
&\quad \quad \quad \langle \text{auto dest: vpa-Main slpa intro: get-proc } n_1 = \text{Main} \rangle \ (\text{get-proc } n = \text{Main}) \\
&\quad \quad \quad \quad \text{simp: vp-def valid-path-def valid-call-list-def slp-def} \\
&\quad \quad \quad \quad \quad \text{same-level-path-def simp del: valid-path-aux.simps} \\
&\quad \text{then obtain } cf \ r \ \text{where } cf \ r \text{ transfers } (\text{map kind } as_1) \ [[cf_1, \text{undefined}]] = [[cf, r]] \\
&\quad \quad \text{by } (\text{fastforce elim: slp-callback-length-equal simp: kinds-def}) \\
&\quad \text{from } (\text{kind } ax = (\lambda s, \text{True}) \rangle \ (\text{kind } a = (\lambda s, \text{True}) \rangle \\
&\quad \quad \text{preds } (\text{kinds } as_1) \ [[cf_1, \text{undefined}]] \ (\text{kind } a_1 = \uparrow \text{id}) \ cf \ r \\
&\quad \quad \text{have preds } (\text{kinds } ax\#((a \# as_1)@[a_1])) \ [[cf_1, \text{undefined}]] \\
&\quad \quad \quad \text{by } (\text{auto simp: kinds-def preds-split}) \\
&\quad \text{from } (n \triangleq c) \ (\text{cases } [cf_2]) \Rightarrow (c', s_2) \\
\end{align*}
\]
obtain \( n_2 \) \( a_2 \) \( cfs_2 \) where \( n \rightarrow a_2 \rightarrow \star \) \( n_2 \) and \( n_2 \triangleq c' \)
and \( \text{preds (kinds as2)} \) \(([cfs_2,\text{undefined}])\)
and \( \text{transfers (kinds as2)} \) \(([[cfs_2,\text{undefined}]) = cfs_2 \) and \( \text{map fst cfs}_2 = s_2 \)
by (fastforce dest: fundamental-property)
from \( n \rightarrow a_2 \rightarrow \star \) \( n_2 \) \( \text{valid-edge ax} \) \( \langle \text{source-node a} = (-\text{High-}) \rangle \) \( \langle \text{target-node a} = n \rangle \)
\( \langle \text{kind a} = (\lambda s. \text{True}) \rangle \)
\( \text{have (-High-) -a}\# a_2 \rightarrow \star \) \( n_2 \) by (fastforce intro: Cons-path simp: vp-def valid-path-def)
from \( \text{final c'} \) \( n_2 \triangleq c' \)
obtain \( a_2 \) where \( \text{valid-edge a} \) and \( \text{source-node a} = n_2 \)
and \( \text{target-node a} = (-\text{Low-}) \) and \( \text{kind a} = \uparrow \text{id} \) by (fastforce dest: final-edge-Low)
\( \text{hence n}_2 \rightarrow \text{a}_2 \rightarrow \star \) \( (-\text{Low-}) \) by (fastforce intro: path-edge)
\( \text{with (-High-) -a}\# a_2 \rightarrow \star \) \( n_2 \) have \( (-\text{High-}) -\text{(a}\# a_2)\circ \text{a}_2 \rightarrow \star \) \( (-\text{Low-}) \)
by (fastforce intro: path-Append simp: vp-def)
\( \text{with \( \langle \text{valid-edge ax} \rangle \) \( \langle \text{source-node ax} = (-\text{Entry-}) \rangle \) \( \langle \text{target-node ax} = (-\text{High-}) \rangle \) \) \( \text{have (-Entry-) -ax}\# ((a\# a_2)\circ ) \rightarrow \star \) \( (-\text{Low-}) \) by \( -\text{(rule Cons-path)} \)
moreover
from (-\text{High-) -a}\# a_2 \rightarrow \star \) \( n_2 \) have \( \text{valid-path-aux} [] \) \( (a\# a_2) \)
\( \text{by (simp add: vp-def valid-path-def)} \)
\( \text{with \( \langle \text{kind a} = \uparrow \text{id} \rangle \) \( \langle \text{valid-path-aux} [] \) \( (a\# (a\# a_2)\circ ) \rangle \) \) \( \text{by simp} \)
ultimately have \( (-\text{Entry-}) -ax\# ((a\# a_2)\circ ) \rightarrow \star \) \( (-\text{Low-}) \)
\( \text{by (simp add: vp-def valid-path-def)} \)
from \( \langle \text{valid-edge ax} \rangle \) \( \langle \text{kind a} = (\lambda s. \text{True}) \rangle \) \( \langle \text{source-node a} = (-\text{High-}) \rangle \) \( \langle \text{target-node a} = n \rangle \)
\( \text{have get-proc n} = \text{get-proc} (-\text{High-}) \)
\( \text{by (fastforce dest: get-proc-intra simp: intra-kind-def)} \)
\( \text{with get-proc-High have get-proc n} = \text{Main} \) by simp
from \( \langle \text{valid-edge a}_2 \rangle \) \( \langle \text{source-node a}_2 = n_2 \rangle \) \( \langle \text{target-node a}_2 = (-\text{Low-}) \rangle \) \( \langle \text{kind a}_2 = \uparrow \text{id} \rangle \)
\( \text{have get-proc a}_2 = \text{get-proc} (-\text{Low-}) \)
\( \text{by (fastforce dest: get-proc-intra simp: intra-kind-def)} \)
\( \text{with get-proc-Low have get-proc a}_2 = \text{Main} \) by simp
from \( n \rightarrow a_2 \rightarrow \star \) \( n_2 \) have \( n \rightarrow a_2 \rightarrow a_\star \) \( n_2 \)
\( \text{by (cases as2)} \)
(auto dest: \( \text{Main-slap intro: get-proc a}_2 = \text{Main} \) \( \langle \text{get-proc n} = \text{Main} \rangle \)
\( \text{simp: vp-def valid-path-def valid-call-list-def sfp-def same-level-path-def simp det: valid-path-aux-simps) \)
then obtain \( cfx' r' \)
\( \text{where cfx': transfers (map kind as2)} \) \(([[cfx_2,\text{undefined}]) = ([cfx_2', r']]) \)
\( \text{by (fastforce elim: sfp-callback-length-equal simp: kinds-def)} \)
\( \text{from \( \langle \text{kind ax} = (\lambda s. \text{True}) \rangle \) \( \langle \text{kind a} = (\lambda s. \text{True}) \rangle \)
\( \langle \text{preds (kinds as2)} \) \(([[cfx_2,\text{undefined}]) \) \) \( \langle \text{kind a}_2 = \uparrow \text{id} \rangle \) \( \text{cfx'} \)
\( \text{have preds (kinds (ax\# ((a\# a_2)\circ ) [a_2]))} \) \(([[cfx_2,\text{undefined}]) \)
\( \text{by (auto simp: kinds-def preds-split)} \)
\( \text{from \( \langle [cfx_2] \approx_L [cfx_2] \rangle \) \( \langle \text{(-High-)} \notin [\text{HRB-slice S}_{\text{CFG}}] \) \( \langle \text{CFG-node (-Low-)} \in S \rangle \)
\( \langle (-\text{Entry-}) -ax\# ((a\# a_1)\circ ) \rightarrow \star \) \( (-\text{Low-}) \).} \)
have \( \text{map \, fst \, (transfers \, (kinds \, (ax \#((a \# as_1)@a_1))) \, [(cf_1, \text{undefined})])} \approx_L \text{map \, fst \, (transfers \, (kinds \, (ax \#((a \# as_2)@a_2))) \, [(cf_2, \text{undefined})])} \)

by (rule nonInterference-path-to-Low)

with \( \text{kind \, ax = (\lambda s. \text{True})} \) \\
\( \text{kind \, a_1 = (\lambda s. \text{True})} \) \\
\( \text{kind \, a_2 = \uparrow \text{id}} \)

\( \text{transfers \, (kinds \, as_1) \, [(cf_1, \text{undefined})] = cfs_1} \) \\
\( \text{map \, fst \, cfs_1 = s_1} \)

\( \text{transfers \, (kinds \, as_2) \, [(cf_2, \text{undefined})] = cfs_2} \) \\
\( \text{map \, fst \, cfs_2 = s_2} \)

show \( ?\text{thesis} \) by (cases \( s_1 \))(cases \( s_2 \),(fastforce \, simp:kinds-def\, \text{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 fulfil 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 = 
'tnode LDCFG-node \times (('var,'val,'ret,'pname) \text{edge-kind}) \times 'node LDCFG-node

3.1.2 Lifting basic definitions using 'edge and 'node

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

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

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

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

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

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

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

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

3.1.3 Lifting the Def and Use sets

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

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

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

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

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

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

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

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

where

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

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

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

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

where

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

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

where

    lift-ParamUses ParamUses (Node n) = ParamUses n

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

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

where

    lift-ParamDefs ParamDefs (Node n) = ParamDefs n

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

34
3.2 The lifting lemmas

3.2.1 Lifting the CFG locales

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

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

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

lemma lift-CFG:
  assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
  and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
  shows CFG src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-get-proc get-proc Main)
  (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
  procs Main
proof –
  interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
    by (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 try 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 try a = try a'
  thus a = a'
proof (induct rule: lift-valid-edge.induct)
case lve-edge thus \{ case by \{ (erule lift-valid-edge.cases, auto dest: edge-det) \}
qed (auto elim: lift-valid-edge.cases)
next
  fix a Q r f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q: r \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 knd a = 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 knd a = Q: r \rightarrow p fs
  thus \exists \text{ins outs. } (p, \text{ins, outs}) \in \text{set procs}
  by (fastforce elim: lift-valid-edge.cases intro: callee-in-procs)
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q: r \rightarrow p fs
  thus lift-get-proc get-proc Main (src a) = lift-get-proc get-proc Main (try a)
  by (fastforce elim: lift-valid-edge.cases intro: get-proc-intra
  simp: get-proc-Entry get-proc-Exit)
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q: r \rightarrow p fs
  thus lift-get-proc get-proc Main (try a) = p
  by (fastforce elim: lift-valid-edge.cases intro: get-proc-call)
next
  fix a Q' p f'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q': p \leftarrow f'
  thus lift-get-proc get-proc Main (src a) = p
  by (fastforce elim: lift-valid-edge.cases intro: get-proc-return)
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q: r \rightarrow p fs
  then obtain ax where valid-edge ax and knd ax = Q: r \rightarrow p fs
  and sourcenode ax \neq \text{Entry} \lor \text{targetnode ax} \neq \text{Exit}
  and src a = \text{Node} (sourcenode ax) and try a = \text{Node} (targetnode ax)
  by (fastforce elim: lift-valid-edge.cases)
from valid-edge ax: (knd ax = Q: r \rightarrow p fs);
have all: \forall a'. valid-edge a' \land \text{targetnode a'} = \text{targetnode ax} \longrightarrow
  (\exists Qx r x fsx. \text{kind} a' = Qx: r x \rightarrow p fsx)
  by (auto dest: call-edges-only)
{ fix $a'$
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit $a'$
  and \( \text{try } a' = \text{try } a \)
  hence \( \exists Qx \; rx \; fsx. \; \text{knd} \; a' = Qx:rx\rightarrow pfsx \)
proof(induct rule:lift-valid-edge.induct)
  case (lve-edge $ax'\; e$)
    note [simp] = (e = (Node (sourcenode $ax'$), kind $ax'$, Node (targetnode $ax'$))):
      from \( \langle \text{try } e = \text{try } a; \text{try } a = \text{Node (targetnode } ax) \rangle \)
      have targetnode $ax' = \text{targetnode } ax$ by simp
      with \( \text{valid-edge } ax' \) all have \( \exists Qx \; rx \; fsx. \; \text{knd } ax' = Qx:rx\rightarrow pfsx \) by blast
      thus \(?\text{case by simp} 
    next
    case (lve-Entry-edge $e$)
      from \( \langle e = (\text{NewEntry}, (\lambda s. \text{True}) \_\_\_, \text{Node Entry})\rangle \) \( \langle \text{try } e = \text{try } a \rangle \)
      \( \langle \text{try } a = \text{Node (targetnode } ax) \rangle \)
      have targetnode $ax = \text{Entry}$ by simp
      with \( \text{valid-edge } ax \) have False by(rule Entry-target)
      thus \(?\text{case by simp} 
    next
    case (lve-Exit-edge $e$)
      from \( \langle e = (\text{Node Exit}, (\lambda s. \text{True}) \_\_\_, \text{NewExit})\rangle \) \( \langle \text{try } e = \text{try } a \rangle \)
      \( \langle \text{try } a = \text{Node (targetnode } ax) \rangle \) have False by simp
      thus \(?\text{case by simp} 
    next
    case (lve-Entry-Exit-edge $e$)
      from \( \langle e = (\text{NewEntry}, (\lambda s. \text{False}) \_\_\_, \text{NewExit})\rangle \) \( \langle \text{try } e = \text{try } a \rangle \)
      \( \langle \text{try } a = \text{Node (targetnode } ax) \rangle \) have False by simp
      thus \(?\text{case by simp} 
  qed }
thus \( \forall a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \land \text{try } a' = \text{try } a \rightarrow (\exists Qx \; rx \; fsx. \; \text{knd } a' = Qx:rx\rightarrow pfsx) \) by simp
next
fix $a \; Q' \; p \; f'$
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit $a$
and \( \text{knd } a = Q'\leftarrow p f' \)
then obtain $ax$ where valid-edge $ax$ and kind $ax = Q'\leftarrow pf'$
and sourcenode $ax \neq \text{Entry} \lor \text{targetnode } ax \neq \text{Exit}
and src $a = \text{Node (sourcenode } ax) \) and \( \text{try } a = \text{Node (targetnode } ax) \)
by (fastforce elim:lift-valid-edge.cases)
from \( \text{valid-edge } ax \) \( \langle \text{knd } ax = Q'\rightarrow p f' \rangle \)
have all:\( \forall a'. \text{valid-edge } a' \land \text{sourceode } a' = \text{sourceode } ax \rightarrow (\exists Qx \; fx. \; \text{knd } a' = Qx\rightarrow pfx) \)
by (auto dest:return-edges-only)
{ fix $a'$
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit $a'$
  and \( \text{src } a' = \text{src } a \)
  hence \( \exists Qx \; fx. \; \text{knd } a' = Qx\rightarrow pfx \)
proof(induct rule:lift-valid-edge.induct)
case (lve-edge ax' e)
  note [simp] = ⟨e = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))⟩
  from ⟨src e = src a⟩ ⟨src a = Node (sourcenode ax)⟩
  have sourcenode ax' = sourcenode ax by simp
  with ⟨valid-edge ax'⟩ all have ∃ Qx fx. kind ax' = Qx←pfx by blast
  thus ?case by simp
next
  case (lve-Entry-edge e)
  from ⟨e = (NewEntry, (λ s. True) ↪, Node Entry)⟩ ⟨src e = src a⟩
  ⟨src a = Node (sourcenode ax)⟩ have False by simp
  thus ?case by simp
next
  case (lve-Exit-edge e)
  from ⟨e = (Node Exit, (λ s. True) ↪, NewExit)⟩ ⟨src e = src a⟩
  ⟨src a = Node (sourcenode ax)⟩ have False by (rule Exit-source)
  thus ?case by simp
next
  case (lve-Entry-Exit-edge e)
  from ⟨e = (NewEntry, (λ s. False) ↪, NewExit)⟩ ⟨src e = src a⟩
  ⟨src a = Node (sourcenode ax)⟩ have False by simp
  thus ?case by simp
next
  qed }
thus ∀ a’. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a’ ∧
  src a’ = src a → (∃ Qx fx. kind ax’ = Qx←pfx) by simp
next
fix a Q r p fs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and kend a = Q:r→pfs
thus lift-get-return-edges get-return-edges valid-edge
  sourcenode targetnode kind a ≠ {}
proof (induct rule:lift-valid-edge.induct)
case (lve-edge ax e)
  from ⟨e = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩
  ⟨kind e = Q:r→pfs⟩
  have kind ax = Q:r→pfs by simp
  with ⟨valid-edge ax⟩ have get-return-edges ax ≠ {}
    by (rule get-return-edge-call)
  then obtain ax’ where ax’ ∈ get-return-edges ax by blast
  with ⟨e = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩ ⟨valid-edge ax⟩
  have ⟨Node (sourcenode ax’), kind ax’, Node (targetnode ax’)⟩ ∈
    lift-get-return-edges get-return-edges valid-edge
    sourcenode targetnode kind e
    by (fastforce intro:lift-get-return-edgesI)
  thus ?case by fastforce
qed simp-all
next
fix a a′
assume a′ ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
thus lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′
proof (induct rule::lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a′ e′)
  from ⟨valid-edge ax⟩ ⟨a′ ∈ get-return-edges ax⟩ have valid-edge a′
  by (rule get-return-edges-valid)
from ⟨valid-edge ax⟩ ⟨a′ ∈ get-return-edges ax⟩ obtain Q r p fs
  where kind ax = Q :r⇒p→fs by (fastforce dest!: only-call-get-return-edges)
with ⟨valid-edge ax⟩ ⟨a′ ∈ get-return-edges ax⟩ obtain Q′ f′
  where kind a′ = Q′ :f′⇒p fs by (fastforce dest!: call-return-edges)
from ⟨valid-edge a′⟩ ⟨kind a′ = Q′ :f′⇒p⟩ have get-proc(sourcenode a′) = p
  by (rule get-proc-return)
have sourcenode a′ ≠ Entry
proof
  assume sourcenode a′ = Entry
  with get-proc-Entry ⟨get-proc(sourcenode a′) = p⟩ have p = Main by simp
  with ⟨kind a′ = Q′ :f′⇒p⟩ have kind a′ = Q′ :f′⇒Main by simp
  with ⟨valid-edge a′⟩ show False by (rule Main-no-return-source)
qed
with ⟨e′ = (Node (sourcenode a′), kind a′, Node (targetnode a′))⟩
  ⟨valid-edge a′⟩
  show ?case by (fastforce intro::lve-edge)
qed
next
fix a a′
assume a′ ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
thus ∃ Q r p fs. knd a = Q :r⇒p→fs
proof (induct rule::lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a′ e′)
  from ⟨valid-edge ax⟩ ⟨a′ ∈ get-return-edges ax⟩ have ∃ Q r p fs. kind ax = Q :r⇒p→fs
  by (rule only-call-get-return-edges)
with ⟨a = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩
  ⟨valid-edge a⟩
  show ?case by simp
qed
next
fix a Q r p fs a′
assume a′ ∈ lift-get-return-edges get-return-edges
  valid-edge sourcenode targetnode kind a and knd a = Q :r⇒p→fs
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
thus ∃ Q′ f′. knd a′ = Q′ :f′⇒p fs
proof (induct rule::lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a′ e′)
  from ⟨a = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩

(knd $a = Q:r\rightarrow_p f s$)

have $\exists a' . \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} a'$

with (valid-edge $a$) ($a' \in \text{get-return-edges}$ $a$) have $\exists Q' f' , \text{kind } a' = Q' \hookrightarrow_p f'$

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

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

show $?e$ case by simp

qed

next

fix $a$ $Q'$ $p f'$

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

and $\text{knd } a = Q' \hookrightarrow_p f'$

thus $\exists! a' . \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a\' \land$

$(\exists Q r f s . \text{knd } a' = Q:r\rightarrow_p f s) \land a \in \text{get-return-edges}$ $\text{get-return-edges}$

valid-edge sourcenode targetnode kind $a'$

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

case (lve-edge $a$ $e$)

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

(knd $e = Q' \hookrightarrow_p f'$) have $\text{knd } a = Q' \hookrightarrow_p f'$ by simp

with (valid-edge $a$)

have $\exists! a' . \text{valid-edge } a' \land (\exists Q r f s . \text{kind } a' = Q:r\rightarrow_p f s) \land$

$a \in \text{get-return-edges}$ $a'$$'$

by (rule return-needs-call)

then obtain $a' Q r f s$ where valid-edge $a'$ and kind $a' = Q:r\rightarrow_p f s$

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

and $\forall x . \text{valid-edge } x \land (\exists Q r f s . \text{kind } x = Q:r\rightarrow_p f s) \land$

$a \in \text{get-return-edges}$ $x \rightarrow x = a'$$'$

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

let $?e' = \langle \text{Node } (\text{sourcenode } a'), \text{kind } a', \text{Node } (\text{targetnode } a') \rangle$

have sourcenode $a' \neq \text{Entry}$

proof

assume sourcenode $a' = \text{Entry}$

with (valid-edge $a'$) ($\text{kind } a' = Q:r\rightarrow_p f s$)

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

qed

with (valid-edge $a'$)

have $\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } ?e'$$'$

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

moreover

from ($\text{knd } a' = Q:r\rightarrow_p f s$) have $?e' = Q:r\rightarrow_p f s$ by simp

moreover

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

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

have $e \in \text{get-return-edges}$ $\text{get-return-edges}$ valid-edge

sourcenode targetnode kind $?e'$$'$ by (fastforce intro:$\text{lift-get-return-edgesI})$

moreover

{ fix $x$

assume $\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } x$

and $\exists Q r f s . \text{knd } x = Q:r\rightarrow_p f s$

and $e \in \text{get-return-edges get-return-edges}$ valid-edge

40
sourcenode targetnode kind \( x \)

\[ \text{from } \langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } x \rangle \]

\[ \exists Q \ r \ fs. \ \text{kind } x = Q_{:r \leftarrow pfs} \] obtain \( y \) where valid-edge \( y \)

and \( x = (\text{Node } (\text{sourcenode } y), \ \text{kind } y, \ \text{Node } (\text{targetnode } y)) \)

by (fastforce elim:lift-valid-edge.cases)

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

sourcenode targetnode kind \( x \) valid-edge \( a \)

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

have \( x = ?e' \)

proof (induct rule:lift-get-return-edges.induct)

\[ \text{case } (\text{lift-get-return-edgesI ax ax' e}) \]

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

by (rule get-return-edges-valid)

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

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

have sourcenode \( a = \text{sourcenode } ax' \)

and targetnode \( a = \text{targetnode } ax' \)

by simp-all

with \( \text{valid-edge } a \) valid-edge \( ax' \) have \([\text{simp}]a = ax' \) by (rule edge-det)

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

\[ \exists Q \ r \ fs. \ \text{kind } x = Q_{:r \leftarrow pfs} \] have \( \exists Q \ r \ fs. \ \text{kind } ax = Q_{:r \leftarrow pfs} \) by simp

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

have \( ax = ax' \) by fastforce

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

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

qed 

ultimately show \( ?\text{case} \) by (blast intro:exI1)

qed simp-all

next

fix \( a \) \( a' \)

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

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

thus \( \exists a'', \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a'' \land \)

src \( a'' = \text{try } a \land \text{try } a' = \text{src } a' \land \text{kind } a'' = (\lambda cf. \ False) \)

proof (induct rule:lift-get-return-edges.induct)

\[ \text{case } (\text{lift-get-return-edgesI ax ax' e}) \]

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

obtain \( ax' \) where valid-edge \( ax' \) and sourcenode \( ax' = \text{targetnode } ax \)

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

by (fastforce dest:intra-proc-additional-edge)

let \( ?\text{ex} = (\text{Node } (\text{sourcenode } ax'), \ \text{kind } ax', \ \text{Node } (\text{targetnode } ax')) \)

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

proof

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

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

qed

with \( \text{sourcenode } ax' = \text{targetnode } ax \) have sourcenode \( ax' \neq \text{Entry} \) by simp

with \( \text{valid-edge } ax' \)

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

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

next
case (lve-Exit-edge ex)
with (src x = src e)
⟨ 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

ultimately show ?case by (blast intro:exI)

qed simp-all

next
fix a Q’ p f’
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q’←p f’

thus ∃!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’←p f’ ⟩
have kind a = Q’←p f’ 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’←p f’ ⟩ 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)) ⟩
\(<\text{targetnode } a' = \text{targetnode } a>\)
have \(\text{trg } ?e' = \text{trg } e\) by simp
moreover
from \(<\text{intra-kind}(\text{kind } a')>\) have \(\text{intra-kind } (\text{kind } ?e')\) by simp
moreover
\{ fix \(x\) assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit \(a\) and \(\text{trg } x = \text{trg } e\) and \(\text{intra-kind } (\text{knd } x)\)
from \(<\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } e>\)
have \(x = ?e'\)
proof (induct rule: lift-valid-edge.cases)
case \(<\text{be-edge } ax ex>\)
from \(<\text{intra-kind } (\text{knd } x)>\) \(<x = ex>\) \(<\text{trg } x = \text{trg } e>\)
\(<e = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax))>\)
have \(\text{intra-kind } (\text{kind } ax)\) and \(\text{targetnode } ax = \text{targetnode } a\) by simp-all
with \(<\text{valid-edge } ax>\) \(\text{imp } \text{have } ax = a' \text{ by fastforce}\)
with \(<x = ex>\) \(<ex = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax))>\)
show ?case by simp
next
case \(<\text{be-Entry-edge } ex>\)
with \(<\text{trg } x = \text{trg } e>\)
\(<e = (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a))>\)
have \(\text{targetnode } a = \text{Entry } \text{by simp}\)
with \(<\text{valid-edge } ax>\) have False by (rule Entry-target)
thus ?case by simp
next
case \(<\text{be-Exit-edge } ex>\)
with \(<\text{trg } x = \text{trg } e>\)
\(<e = (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a))>\)
have False by simp
thus ?case by simp
next
case \(<\text{be-Entry-Exit-edge } ex>\)
with \(<\text{trg } x = \text{trg } e>\)
\(<e = (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a))>\)
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 valid-edge sourcenode targetnode kind Entry Exit \(a\)
and lift-valid-edge valid-edge sourcenode targetnode kind 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 valid-edge \(x\)
and \(a:a = (\text{Node } (\text{sourcenode } x), \text{kind } x, \text{Node } (\text{targetnode } x))\) and valid-edge
\[ x' \]

and \( \mathbf{a}'\mathbf{a}' = (\text{Node}(\text{sourcenode } x'), \text{kind } x', \text{Node}(\text{targetnode } x')) \]
by (auto elim!:lift-valid-edge.cases)
with \( \text{kind } \mathbf{a} = Q_1 : r_1 \rightarrow p s_1 \) and \( \text{kind } \mathbf{a}' = Q_2 : r_2 \rightarrow p s_2 \) by simp-all
have \( \text{valid-edge } x \) (valid-edge \( x' \)) have \( \text{targetnode } x = \text{targetnode } x' \)
by (rule same-proc-call-unique-target)
with \( \mathbf{a} \mathbf{a}' \) show \( \text{trg } a = \text{trg } a' \) by simp
next
from unique-callers show distinct-fst procs.
next
fix \( p \) ins outs
assume \( (p, \text{ins}, \text{outs}) \in \text{set procs} \)
from distinct-formal-ins[OF this] show distinct ins.
next
fix \( p \) ins outs
assume \( (p, \text{ins}, \text{outs}) \in \text{set procs} \)
from distinct-formal-outs[OF this] show distinct outs.
qed

lemma lift-CFG-wf:
assumes \( \text{wf} : \text{CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses} \)
and \( \text{pd} : \text{Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit} \)
shows \( \text{CFG-wf src trg knd (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry)} \)
\( \text{(lift-get-proc get-proc Main)} \)
\( \text{(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) procs Main (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)} \)
\( \text{(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)} \)
proof
interpret \( \text{CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses} \)
by (rule \( \text{wf} \))
interpret \( \text{Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit} \)
by (rule \( \text{pd} \))
interpret \( \text{CFG:CFG src trg knd} \)
\( \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry} \)
\( \text{lift-get-proc get-proc Main} \)
\( \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind} \)
\( \text{procs Main} \)
by (fastforce intro:lift-CFG \( \text{wf} \) \( \text{pd} \))
show \( ?\text{thesis} \)
proof
show \( \text{lift-Def Def Entry Exit H L NewEntry} = \{\} \wedge \)
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 (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 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 Q' r' p' fs' ins outs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q': r'→ p'fs' and (p', ins, outs) ∈ set procs
thus length (lift-ParamDefs ParamDefs (trg a)) = length outs
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from ⟨valid-edge a⟩ have distinct (ParamDefs (targetnode a))
    by (rule distinct-ParamDefs)
    with ⟨src e = Node (sourcenode a), kind a, Node (targetnode a)⟩ 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' r'→ p'fs' and (p', ins, outs) ∈ set procs
thus length (lift-ParamDefs ParamDefs (try a)) = length outs
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩ ⟨knd e = Q' r'→ p'fs'⟩
  have kind a = Q' r'→ p'fs' and trg e = Node (targetnode a) by simp-all

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

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

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

qed simp-all

next

fix \( n \) \( V \)

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

\( \langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} a \rangle \) \( n \)
and \( V \in \text{set (lift-ParamDefs ParamDefs } n) \)

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

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

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

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

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

next

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

next

assume \( \exists m. n = \text{Node } m \land \text{valid-node } m \)

then obtain \( m \) where \( n = \text{Node } m \land \text{valid-node } m \)
by blast
from \( \langle n = \text{Node } m \rangle \) \( \langle V \in \text{set (lift-ParamDefs ParamDefs } n) \rangle \)

have \( V \in \text{set (ParamDefs } m) \)

by simp

with \( \langle \text{valid-node } m \rangle \) have \( V \in \text{Def } m \)
by (rule ParamDefs-in-Def)

with \( \langle n = \text{Node } m \rangle \) show \(?\text{thesis by (fastforce intro:lift-Def-node)} \)

qed simp-all

next

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

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

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

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

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

have \( \text{knd } a = Q: r \xrightarrow{p} fs \) \( \langle \text{knd } e = Q: r \xrightarrow{p} fs \rangle \)

by simp

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

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

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

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

qed simp-all

next

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

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

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

\[
\begin{align*}
\text{proof (induct rule:lift-valid-edge.induct)} \\
\text{case (lev-edge \ a \ e)} \\
\text{show ?case} \\
\text{proof (rule ccontr)} \\
\text{assume lift-Def \ Def \ Entry \ Exit \ H \ L \ (\text{src} \ e) \neq \{\}} \\
\text{then obtain } x \text{ where } x \in \text{lift-Def} \ Def \ Entry \ Exit \ H \ L \ (\text{src} \ e) \text{ by blast} \\
\text{from } \langle e = (\text{Node} \ (\text{sourcenode} \ a), \text{kind} \ a, \text{Node} \ (\text{targetnode} \ a)) \rangle \langle \text{kind} \ e = Q:\,r \hookrightarrow_p \text{fs} \rangle \\
\text{have kind \ a = Q:\,r \hookrightarrow_p \text{fs} \text{ by simp}} \\
\text{with } \langle \text{valid-edge} \ a \rangle \langle \text{Def} \ (\text{sourcenode} \ a) = \{\} \rangle \\
\text{by (rule call-source-Def-empty)} \\
\text{have sourcenode \ a \neq Entry} \\
\text{proof} \\
\text{assume sourcenode \ a = Entry} \\
\text{with } \langle \text{valid-edge} \ a \rangle \langle \text{kind} \ a = Q:\,r \hookrightarrow_p \text{fs} \rangle \\
\text{show False by (rule Entry-no-call-source)} \\
\text{qed} \\
\text{from } \langle e = (\text{Node} \ (\text{sourcenode} \ a), \text{kind} \ a, \text{Node} \ (\text{targetnode} \ a)) \rangle \\
\text{have src \ e = Node \ (\text{sourcenode} \ a) by simp} \\
\text{with } \langle \text{valid-edge} \ a \rangle \langle \text{Def} \ (\text{sourcenode} \ a) = \{\} \rangle \langle x \in \text{lift-Def} \ Def \ Entry \ Exit \ H \ L \ (\text{src} \ e) \rangle \\
\text{have sourcenode \ a \neq Entry} \\
\text{proof} \\
\text{assume sourcenode \ a = Entry} \\
\text{with } \langle \text{valid-edge} \ a \rangle \langle \text{kind} \ a = Q:\,r \hookrightarrow_p \text{fs} \rangle \\
\text{show False by (rule Entry-no-call-source)} \\
\text{qed} \\
\text{qed simp-all} \\
\text{next} \\
\text{fix \ n \ V} \\
\text{assume CFG.CFG.valid-node \ src \ try} \\
\text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) \ n} \\
\text{and } V \in \bigcup \text{set (lift-ParamUses ParamUses \ n)} \\
\text{hence } ((n = \text{NewEntry}) \lor n = \text{NewExit}) \lor (\exists m. \ n = \text{Node} \ m \land \text{valid-node} \ m) \\
\text{by (auto elim:lift-valid-edge.cases simp:CFG.valid-node-def)} \\
\text{thus } V \in \text{lift-Use} \ Use \ Entry \ Exit \ H \ L \ n \ \text{apply –} \\
\text{proof (erule disjE)+} \\
\text{assume } n = \text{NewEntry} \\
\text{with } \langle V \in \bigcup \text{set (lift-ParamUses ParamUses \ n)} \rangle \text{ show ?thesis by simp} \\
\text{next} \\
\text{assume } n = \text{NewExit} \\
\text{with } \langle V \in \bigcup \text{set (lift-ParamUses ParamUses \ n)} \rangle \text{ show ?thesis by simp} \\
\text{next} \\
\text{assume } \exists m. \ n = \text{Node} \ m \land \text{valid-node} \ m \\
\text{then obtain } m \text{ where } n = \text{Node} \ m \text{ and valid-node} \ m \text{ by blast} \\
\text{from } \langle V \in \bigcup \text{set (lift-ParamUses ParamUses \ n)} \rangle \langle n = \text{Node} \ m \rangle \\
\text{have } V \in \bigcup \text{set (ParamUses \ m)} \text{ by simp} \\
\text{with } \langle \text{valid-node} \ m \rangle \text{ have } V \in \text{Use} \ m \text{ by (rule ParamUses-in-Use)} \\
\text{with } \langle n = \text{Node} \ m \rangle \text{ show ?thesis by (fastforce intro:lift-Use-node)} \\
\text{qed} \\
\text{next} \\
49
fix a Q p f ins outs V
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and kend 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 kend a = Q p f by simp
  from ⟨valid-edge a⟩ ⟨knd a = Q p f⟩ ⟨(p, ins, outs) ∈ set procs⟩ ⟨V ∈ set outs⟩
  have V ∈ Use (sourcenode a) by (rule outs-in-Use)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  have src e = Node (sourcenode a) by simp
  with ⟨V ∈ Use (sourcenode a)⟩ show ?case by (fastforce intro:lift-Use-node)
qed simp-all
next
fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V ⊄ lift-Def Def Entry Exit H L (src a) and intra-kind (knd a)
and pred (knd a) s
thus state-val (transfer (knd a) s) V = state-val s V
proof (induct rule:lift-valid-edge.induct)
case (lee-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨intra-kind (knd e)⟩ ⟨pred (knd e) s⟩
  have intra-kind (knd a) and pred (knd a) s
  and kend e = kind a and src e = Node (sourcenode a) by simp-all
  from ⟨V ⊄ lift-Def Def Entry Exit H L (src e)⟩ ⟨src e = Node (sourcenode a)⟩
  have V ⊄ Def (sourcenode a) by (auto dest: lift-Def-node)
  from ⟨valid-edge a⟩ ⟨V ⊄ Def (sourcenode a)⟩ ⟨intra-kind (knd a)⟩
  ⟨pred (knd a) s⟩
  have state-val (transfer (knd a) s) V = state-val s V
  by (rule CFG-intra-edge-no-Def-equal)
  with ⟨knd e = kind a⟩ show ?case by simp
next
case (lee-Entry-edge e)
  from ⟨e = (NewEntry, (λs. True), Node Entry))⟩ ⟨pred (knd e) s⟩
  show ?case by (cases s) auto
next
case (lee-Exit-edge e)
  from ⟨e = (Node Exit, (λs. True), NewExit)⟩ ⟨pred (knd e) s⟩
  show ?case by (cases s) auto
next
case (lee-Entry-Exit-edge e)
  from ⟨e = (NewEntry, (λs. False), 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

∀ 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 (knd a) by simp

show ?case

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) by simp

proof (cases targetnode a = Exit)

case Truc

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 (knd 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) by pred (knd e) s; pred (knd e) s'
show ?thesis by\{(cases s,\ auto,cases s',\ auto)\}

next

\textbf{case False}

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

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

\textbf{have} V' \in lift-Use Use Entry Exit {H \ L} (src e)

\textbf{by} \{fastforce intro:lift-Use-node\}

\}

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

\textbf{state-val s V = state-val s' V}

\textbf{by} fastforce

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

\textbf{have} \forall V \in Def (source\node a). state-val (transfer (kind a) s) V =

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

\textbf{by} \{erule CFG-intra-edge-transfer-uses-only-Use,\ auto\}

\textbf{from}\ \langle V \in lift-Def Def Entry Exit {H \ L} (src e)\rangle \textbf{False}

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

\textbf{have} V \in Def (source\node a) \textbf{by}(fastforce elim:lift-Def-set.cases)

\textbf{with}\ \forall V \in Def (source\node a). state-val (transfer (kind a) s) V =

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

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

\textbf{show} ?thesis by simp

qed

next

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

\textbf{from}\ \langle V \in lift-Def Def Entry Exit {H \ L} (src e)\rangle

\textbf{\langle e = (NewEntry, (λs. True), Node Entry)\rangle}

\textbf{have} False \textbf{by}(fastforce elim:lift-Def-set.cases)

\textbf{thus} ?case by simp

next

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

\textbf{from}\ \langle V \in lift-Def Def Entry Exit {H \ L} (src e)\rangle

\textbf{\langle e = (Node Exit, (λs. True), NewExit)\rangle}

\textbf{have} False

\textbf{by}(fastforce elim:lift-Def-set.cases intro!:Entry-noteq-Exit simp:Exit-empty)

\textbf{thus} ?case by simp

next

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

\textbf{thus} ?case by\{cases s\} \textbf{auto}

qed

qed

next

\textbf{fix} a s s'

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

\textbf{and pred (kind a) s and snd (hd s) = snd (\text{hd s'})}

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

52
and length s = length s′

thus pred (knd a) s′

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

  case (lee-edge a e)

  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a)); pred (knd e) s⟩

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

  from ⟨src e = Node (sourcenode a); ∀ ⋀ ∈ lift-Use Use Entry Exit H L (src e). state-val ⋀ V = state-val s′ ⋀ V⟩

  have ∀ V ∈ Use (sourcenode a). state-val s V = state-val s′ V by (auto dest: lift-Use-node)

  from ⟨valid-edge a⟩ ⟨pred (knd a) s⟩ ⟨snd (hd s) = snd (hd s′)⟩

  this ⟨length s = length s′⟩

  have pred (kind a) s′ by (rule CFG-edge-Uses-pred-equal)

  with ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩

  show ?case by simp

next

  case (lee-Entry-edge e)

  thus ?case by (cases s′) auto

next

  case (lee-Exit-edge e)

  thus ?case by (cases s′) auto

next

  case (lee-Entry-Exit-edge e)

  thus ?case by (cases s) auto

qed

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 fs = length ins

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

  case (lee-edge a e)

  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a)); knd e = Q:r→pfs⟩

  have kind a = Q:r→pfs by simp

  from ⟨valid-edge a; knd a = Q:r→pfs⟩ ⟨(p, ins, outs) ∈ set procs; show ?case by (rule CFG-call-edge-length)⟩

  qed simp-all

next

  fix a Q′ r′ p′ fs’ s′

  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a

  and knd a = Q:r→pfs and knd a′ = Q′:r′→p′fs’

  and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′

  and src a = src a′ and pred (knd a) s and pred (knd a′) s

  from ⟨valid-edge a; knd a = Q:r→pfs⟩ ⟨pred (knd 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)
simp
V
Q
′
next:
⟩
r
←
⟨
x
obtain
sourcenode
have
⟨
from
lift-valid-edge
valid-edge
sourcenode
targetnode
kind
Entry
Exit
a
assume
Q
r
p
fs
and
pred
(kind
x)
s
by
 elimin::lift-valid-edge.cases
from
lift-valid-edge
valid-edge
sourcenode
targetnode
kind
Entry
Exit
a
′
and
kind
x
′
= Q
r
p
fs
and
pred
(kind
x)
s
obtain
x
′
where
a
a
′
= (Node
(sourcenode
x),
kind
x
′
,
Node
(targetnode
x)
)
and
valid-edge
x
′
and
sourcenode
x
′
= Node
(sourcenode
x)
and
kind
x
′
= Q
r
p
fs
and
pred
(kind
x
′)
s
by
elim::lift-valid-edge.cases
from
(sourcenode
x
= sourcenode
x
′)
by
simp
from
valid-edge
x
(kind
x
= Q
r
p
fs
)
(kind
x
′
= Q
r
p
fs
)
(sourcenode
x
= sourcenode
x
′
)
(pred
(kind
x)
s)
(pred
(kind
x
′)
s)
with
a
a
′
show
a
a
′
by
rule
CFG-call-determ
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
r
p
fs
and
i
<
length
ins
and
(p,
ins,
outs)
∈
set
procs
and
pred
(kind
a)
s
and
pred
(kind
a)
s
′
and
∀
V
∈
ParamUses
ParamUses
(sourcenode
x
′)
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
(lve-edge
a
e)
from
(e
= (Node
(sourcenode
a),
kind
a,
Node
(targetnode
a)))
(kind
e
= Q
r
p
fs
)
(pred
(kind
e)
s)
(pred
(kind
e)
s
′
)
have
kind
a
= Q
r
p
fs
and
pred
(kind
a)
s
and
pred
(kind
a)
s
′
and
sourcenode
a
by
simp
all
from
∀
V
∈
ParamUses
ParamUses
(sourcenode
a)
i.
state-val
s
V
= state-val
s
′
V
have
∀
V
∈
ParamUses
ParamUses
(sourcenode
a)
i.
state-val
s
V
= state-val
s
′
V
by
simp
with
valid-edge
a
(kind
a
= Q
r
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
′
assume
lift-valid-edge
valid-edge
sourcenode
targetnode
kind
Entry
Exit
a
and
kind
a
= Q
r
p
fs
′
and
(p,
ins,
outs)
∈
set
procs
thus
f
′
cf
′
= cf
′
lift-ParamDefs
ParamDefs
(trg
a)
[↦]
map
cf
outs
proof
(induct
rule::lift-valid-edge.induct)
case
(lve-edge
a
e)
from
(e
= (Node
(sourcenode
a),
kind
a,
Node
(targetnode
a)))
(kind
e
= Q
r
p
fs
′
)
have kind a = Q′←p′f′ and trg e = Node (targetnode a) by simp-all
from (valid-edge a) (kind a = Q′←p′f′) ⟨p, ins, outs⟩ ∈ set procs
have f′ cf cf′ = cf′(ParamDefs (targetnode a) [:=] map cf outs)
  by (rule CFG-return-edge-fun)
with (trg e = Node (targetnode a)) show ?case by simp
qed simp-all
next
fix a a′
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′
  and src a = src a′ and trg a ‰ trg a′
  and intra-kind (knd a) and intra-kind (knd a′)
thus ∃ Q Q′. knd a = (Q) ∨ ∧ knd a′ = (Q′) ∨ ∧
  (∀ s. (Q s −→¬ Q′ s) ∧ (Q′ s −→¬ Q s))
proof (induct rule:lift-valid-edge.induct)
  case (lve-edge a e)
  from ⟨lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′
    (valid-edge a) e = (Node (source node a), kind a, Node (targetnode a))⟩
  ⟨src e = src a′, try e ≠ try a′, ⟨intra-kind (knd e)⟩, ⟨intra-kind (knd a′)⟩⟩
  show ?case
  proof (induct rule:lift-valid-edge.induct)
  case lve-edge thus ?case by (auto dest:deterministic)
  next
  case (lve-Exit-edge e′)
  from ⟨e = (Node (source node a), kind a, Node (targetnode a))⟩
  ⟨e′ = (Node Exit, (λs. True) ∨, NewExit), ⟨src e = src e′⟩⟩
  have sourcenode a = Exit by simp
  with ⟨valid-edge a⟩ have False by (rule Exit-source)
  thus ?case by simp
  qed auto
  qed (fastforce elim:lift-valid-edge.cases)+
  qed
qed

lemma lift-CFGExit:
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
shows CFGExit src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main NewExit
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by (rule wf)
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc
  get-return-edges procs Main Exit
by(rule pd)
interpret 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
  thus False by(fastforce elim:lift-valid-edge_cases)
next
show lift-get-proc get-proc Main NewExit = Main by simp
next
fix a Q p f
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q←p f and trg a = NewExit
  thus False by(fastforce elim:lift-valid-edge_cases)
next
show ∃a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
  src a = NewEntry ∧ trg a = NewExit ∧ knd a = (λs. 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 sourcenode targetnode kind
procs Main lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L
lift-ParamDefs ParamDefs lift-ParamUses ParamUses
by (fastforce intro: lift-CFG-wf wf pd)
interpret CFGExit: CFGExit src trg knd
  lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main NewExit
by (fastforce intro: lift-CFGExit wf pd)
show ?thesis
proof
  show lift-Def Def Entry Exit H L NewExit = {} ∧
       lift-Use Use Entry Exit H L NewExit = {}
  by (fastforce elim: lift-Def-set.cases lift-Use-set.cases)
qed

3.2.2 Lifting the SDG

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

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

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

proof
assume \( m = \text{sourcenode } a \)
show \( \text{thesis} \)
proof (cases \( m = \text{Entry} \))
case \( \text{True} \)
have \( \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit} \)
\( (\text{NewEntry},(\lambda s. \text{True}) \sqrt , \text{Node Entry}) \)
by (fastforce intro: lve-Entry-edge)
with \( (m = \text{Entry}) \) show \( \text{thesis} \) by (fastforce simp: CFGExit.valid-node-def)
next
case \( \text{False} \)
with \( (m = \text{sourcenode } a) \) \( (\text{valid-edge } a) \)
have \( \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit} \)
\( (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a)) \)
by (fastforce intro: lve-edge)
with \( (m = \text{sourcenode } a) \) show \( \text{thesis} \) by (fastforce simp: CFGExit.valid-node-def)
qed
next
assume \( m = \text{targetnode } a \)
show \( \text{thesis} \)
proof (cases \( m = \text{Exit} \))
case \( \text{True} \)
have \( \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit} \)
\( (\text{Node Exit},(\lambda s. \text{True}) \sqrt , \text{NewExit}) \)
by (fastforce intro: lve-Exit-edge)
with \( (m = \text{Exit}) \) show \( \text{thesis} \) by (fastforce simp: CFGExit.valid-node-def)
next
case \( \text{False} \)
with \( (m = \text{targetnode } a) \) \( (\text{valid-edge } a) \)
have \( \text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit} \)
\( (\text{Node } (\text{sourcenode } a), \text{kind } a, \text{Node } (\text{targetnode } a)) \)
by (fastforce intro: lve-edge)
with \( (m = \text{targetnode } a) \) show \( \text{thesis} \) 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 = as\( \rightarrow^* \) m' and \( \forall c \in \text{set } cs. \text{valid-edge } c \)
and \( m \neq \text{Entry} \lor m' \neq \text{Exit} \)

hence \( \exists cs'. \text{cs} = \text{CFG.CFG.valid-path-aux } \text{kind} \)
\( (\text{lift-get-return-edges } \text{get-return-edges } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind}) \)
\( \text{cs} \lor \text{cs'} \land \text{list-all2 } (\lambda c c'. \text{c' } = (\text{Node } (\text{sourcenode } c), \text{kind } c, \text{Node } (\text{targetnode } c))) \)
\( \text{cs} \lor \text{cs'} \land \text{CFG.CFG.path src trg} \)
\( (\text{lift-valid-edge } \text{valid-edge } \text{sourcenode } \text{targetnode } \text{kind } \text{Entry } \text{Exit}) \)
\( (\text{Node } m) \lor \text{Node } m' \)

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 (rule path-valid-node)
obtain \( cs' \) where \( cs' = \)
\[\text{map} (\lambda c. (\text{Node (sourcenode } c\text{),kind } c, \text{Node (targetnode } c\text{) })) \text{ by simp}\]
hence list-all2 \( (\lambda c' . c' = (\text{Node (sourcenode } c\text{), kind } c, \text{ Node (targetnode } c\text{)}) ) \) \( cs \) \( cs' \)
by (simp add: list-all2-conv-all-nth)
with \( \langle \text{valid-node } m \rangle \) show ?case
apply (rule-tac \( x = cs' \) in \( \text{exI} \))
apply (rule-tac \( x = [] \) in \( \text{exI} \))
by (fastforce intro: CFGExit.empty-path lift-valid-node)
next
case (vpa-intra \( cs a as \))
note \( IH = \langle \forall m . \langle m \rightarrow \ast \rangle \land \forall c \in \text{set } cs . \text{ valid-edge } c; m \neq \text{Entry } \lor m' \rangle \)
\( \neq \) Exit \( \implies \)
\( \exists cs' es . \text{CFG.valid-path-aux knd} \)
\( \langle \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind} \rangle \) \( cs' \) \( es \wedge \)
list-all2 \( (\lambda c' . c' = (\text{Node (sourcenode } c\text{), kind } c, \text{ Node (targetnode } c\text{)}) ) \) \( cs \) \( cs' \wedge \text{CFG.path src trg} \)
\( \langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \rangle \) \( \langle \text{Node } m \rangle \) \( es \langle \text{Node } m' \rangle \).
from \( m - a \# as \rightarrow m' \) have \( m = \text{sourcenode } a \) and valid-edge \( a \) and targetnode \( a - as \rightarrow m' \) by (auto elim: path-split-Cons)
show ?case
proof (cases sourcenode \( a = \text{Entry } \land \text{targetnode } a = \text{Exit} ))
case True
with \( \langle m = \text{sourcenode } a \rangle \langle m \neq \text{Entry } \lor m' \neq \text{Exit} \rangle \)
have \( m' \neq \text{Exit} \) by simp
from True have targetnode \( a = \text{Exit} \) by simp
with \( \langle \text{targetnode } a - as \rightarrow m' \rangle \) \( \text{have } m' = \text{Exit} \)
by \( \langle \text{drule path-Exit-source}, \text{auto} \rangle \)
with \( \langle m' \neq \text{Exit} \rangle \) \( \text{have } False \) by simp
thus \( \text{thesis} \) by simp
next
case False
let \( ?e = (\text{Node (sourcenode } a\text{),kind } a, \text{Node (targetnode } a\text{)}) \)
from False valid-edge \( a \)
have lift-valid-edge \( \text{valid-edge sourcenode targetnode kind Entry Exit } ?e \)
by (fastforce intro: lve-edge)
have targetnode \( a \neq \text{Entry} \)
proof
assume targetnode \( a = \text{Entry} \)
with \( \text{valid-edge } a \) show False by (rule Entry-target)
qed
hence \( \text{targetnode } a \neq \text{Entry } \lor m' \neq \text{Exit} \) by simp
from \( IH[\langle \text{targetnode } a - as \rightarrow m' \rangle \land \forall c \in \text{set } cs . \text{ valid-edge } c \rangle \) this]
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' es
and list:list-all2
  (λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) cs cs'
and path:CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node (targetnode a)) es (Node m') by blast
from (intra-kind (kind a)) valid-path have CFG.valid-path-aux knd
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs' (?e#es) by (fastforce simp:intra-kind-def)
moreover
from path (m = sourcenode a)
  (lift-valid-edge valid-edge sourcenode kind Entry Exit ?e)
  have CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node m) (?e#es) (Node m') by (fastforce intro:CFGExit.Cons-path)
ultimately show ?thesis using list by blast
qed
next
case (vpa-Call cs a as Q r p fs)
  note IH = (∀m. [m − as→* m'; ∀ c∈set (a # cs). valid-edge c;
    m ≠ Entry ∨ m' ≠ Exit] →
    ∃ cs' es. CFG.valid-path-aux knd
    (lift-get-return-edges get-return-edges valid-edge sourcenode
    targetnode kind) cs' es ∧
    list-all2 (λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c)))
    (?a#cs) cs' ∧ CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node m) es (Node m'))
  from (m − a # as→* m') have m = sourcenode a and valid-edge a
    and targetnode a − as→* m' by (auto elim: path-split-Cons)
  from ∀ c∈set cs. valid-edge c (?valid-edge a)
  have ∀ c∈set (a # cs). valid-edge c by simp
  let ?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→p fs)
  show False by (rule Entry-no-call-source)
qed
with (?valid-edge a)
  have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e
    by (fastforce intro:lve-edge)
  have targetnode a ≠ Entry
proof
  assume targetnode a = Entry
  with (?valid-edge a) show False by (rule Entry-target)
qed
hence targetnode a ≠ Entry ∨ m' ≠ Exit by simp
from IH [OF \targetnode a \rightarrow set (a \# cs), valid-edge c \land this]

obtain cs' es

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

(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) \(\lambda\) cs' es

and list:list-all2

\((\lambda c'.\, c' = (Node (sourcenode c), kind c, Node (targetnode c)))) (a\#cs) cs'

and path:CFG.path src try

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

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

from list obtain cx csx where cs' = cx#csx

and cx:cx = (Node (sourcenode a), kind a, Node (targetnode a))

and list':list-all2

\((\lambda 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;\rightarrow 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 \(\lambda m = sourcenode a\)

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

have CFG.path src try

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

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

ultimately show ?case using list' by blast

next

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

note IH' = \(\forall m.\, [m \rightarrow set \# a]. valid-edge c; m \neq Entry \lor m' \neq Exit\)

\exists cs' es. CFG.valid-path-aux knd

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

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

\([\] \land CFG.path src try

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

(Node m) es (Node m'):

from \(m \rightarrow a \# as \rightarrow m'\) have m = sourcenode a and valid-edge a

and targetnode a \rightarrow as \rightarrow m' by (auto elim: path-split-Cons)

let \(\forall e = (Node (sourcenode a), kind a, Node (targetnode a))\)

have targetnode a \neq Exit

proof

assume targetnode a = Exit

with (valid-edge a) (\(\lambda kind a = Q;\rightarrow p;fs\)) 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: lift-edge)

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

hence targetnode a ≠ Entry ∨ m' ≠ Exit by simp
from IH[OF ⟨targetnode a = as→* m' - this⟩] obtain es
where valid-path:CFG.valid-path-aux knd
  (lift-get-return-edges valid-return-edges valid-edge sourcenode
targetnode kind) [] es
  and path:CFG.path src try
  (lift-valid-edge valid-edge source node targetnode kind Entry Exit)
  (Node (targetnode a)) es (Node m') by auto
from valid-path ⟨kind a = Q←pf⟩
have CFG.valid-path-aux knd
  (lift-get-return-edges valid-return-edges valid-edge sourcenode
targetnode kind) [] (?e#es) by simp
moreover
from path ⟨m = sourcenode a⟩
  ⟨lift-valid-edge valid-edge source node targetnode kind Entry Exit ?e⟩
have CFG.path src try
  (lift-valid-edge valid-edge source node 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 = ⟨∀ c∈set cs'. valid-edge c; m ≠ Entry ∨ m' ≠ Exit ⟩
  have ⟨csx es. CFG.valid-path-aux knd
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) csx es ∧
  list-all2 (λ c c'. c' = (Node (sourcenode c), kind c, Node (targetnode c)))
  cs' csx ∧ CFG.path src try
  (lift-valid-edge valid-edge source node targetnode kind Entry Exit)
  (Node m) es (Node m')⟩
  from ⟨m ≠ a ≤ as→* m'⟩ have m = sourcenode a and valid-edge a
  and targetnode a = as→* m' by (auto elim:path-split-Cons)
  from ∀ c∈set cs. valid-edge c ⟨cs = c' ≠ cs'⟩
  have valid-edge c' and ∀ c∈set cs'. valid-edge c by simp-all
  let ℓe = (Node (sourcenode a), kind a, Node (targetnode a))
  have targetnode a ≠ Exit
proof
  assume targetnode a = Exit
  with ⟨valid-edge a⟩ ⟨kind a = Q←pf⟩ show False by (rule Exit-no-return-target)
qed

with ⟨valid-edge a⟩
have lift-valid-edge valid-edge source node kind Entry Exit ℓe
  by (fastforce intro: lve-edge)
have targetnode a ≠ Entry
proof

62
\begin{verbatim}
assume targetnode a = Entry
with (valid-edge a) show False by (rule Entry-target)

qed

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

from IH[OF \langle targetnode a = as→* m' \rangle \land \forall c | set cs'. valid-edge c | this]

generate csx es

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

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

and list: list-all2

(\lambda c. c' = (Node (sourcenode c), kind c, Node (targetnode c))) es' csx

and path:CFG.path src try

(lift-valid-edge valid-edge source-destination targetnode kind Entry Exit)

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

from \langle valid-edge c' \rangle \langle a \in get-return-edges c' \rangle

have \? e ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind

(Node (source-destination c'), kind c', Node (targetnode c')) by (fastforce intro: lift-get-return-edgesI)

with valid-path \langle kind a = Q←pf \rangle

have CFG.valid-path-aux kind

(lift-get-return-edges get-return-edges valid-edge source-destination targetnode kind)

((Node (source-destination c'), kind c', Node (targetnode c')) \# csx) \? e \# es

by simp

moreover

from path \langle m = sourcenode a \rangle

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

have CFG.path src try

(lift-valid-edge valid-edge source-destination targetnode kind Entry Exit)

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

ultimately show ?case using \langle kind a = Q←pf \rangle by blast

qed

hence lift-valid-path: \\exists m as m', \{ m = as→*' m' ; m ≠ Entry \lor m' ≠ Exit \}

\implies \exists es. CFG.CFG.valid-path' src try kind

(lift-valid-edge valid-edge source-destination targetnode kind Entry Exit)

(lift-get-return-edges get-return-edges valid-edge source-destination targetnode kind)

(Node m) es (Node m')

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

show ?thesis

proof

fix n assume CFG.CFG.valid-node src try

(lift-valid-edge valid-edge source-destination targetnode kind Entry Exit) n

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

\end{verbatim}
by (auto elim: lift-valid-edge.cases simp: CFGExit.valid-node-def)
thus 3 as. CFGCFG, valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) NewEntry as n apply —
proof (erule disjE)+
assume n = NewEntry
hence CFGCFG, valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) NewEntry [] n
by (fastforce intro: CFGExit.empty-path
  simp: CFGExit_vp-def CFGExit.valid-path-def)
thus ?thesis by blast
next
assume n = NewExit
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry, (λs. False) ∘ NewExit) by (fastforce intro: lve-Entry-Exit-edge)
hence CFGCFG, path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry [(NewEntry, (λs. False) ∘ NewExit)] NewExit
by (fastforce dest: CFGExit.path-edge)
with (n = NewExit) have CFGCFG, valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) NewEntry [(NewEntry, (λs. False) ∘ NewExit)] n
by (fastforce simp: CFGExit_vp-def CFGExit.valid-path-def)
thus ?thesis by blast
next
assume ∃ m. n = Node m ∧ valid-node m
then obtain m where n = Node m and valid-node m by blast
from ⟨valid-node m⟩
show ?thesis
proof (cases m rule: valid-node-cases)
  case Entry
  have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (NewEntry, (λs. True) ∘ Node Entry) by (fastforce intro: lve-Entry-Edge)
  with (m = Entry) (n = Node m) have CFGCFG, path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry [(NewEntry, (λs. True) ∘ Node Entry)] n
  by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
    simp: CFGExit.valid-node-def)
  thus ?thesis by (fastforce simp: CFGExit_vp-def CFGExit.valid-path-def)
next
  case Exit
  from inner obtain ax where valid-edge ax and intra-kind (kind ax)
  and inner-node (sourcenode ax)
  and targetnode ax = Exit by (erule inner-node-Exit-edge)
  hence lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit

64
(Node (sourcenode ax), kind ax, Node Exit)
by (auto intro: lift-valid-edge lve-edge simp: inner-node-def)
hence CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node (sourcenode ax)) [[(Node (sourcenode ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)
with (intra-kind (kind ax))
have slp-edge: CFG_CFG.same-level-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind)
(Node (sourcenode ax)) [[(Node (sourcenode ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def
intra-kind-def)
with (sourcenode ax ≠ Exit)
proof
assume sourcenode ax = Exit
with (valid-edge ax) show False by (rule Exit-source)
qed
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry, (λs. True), Node Entry) by (fastforce intro: lve-Entry-edge)
hence CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node (sourcenode ax)) [[(Node (sourcenode ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
simp: CFGExit.valid-node-def)
with (intra-kind (kind ax))
have slp-edge: CFG_CFG.same-level-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind)
(Node (sourcenode ax)) [[(Node (sourcenode ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def
intra-kind-def)
from (inner-node (sourcenode ax)) have valid-node (sourcenode ax)
by (rule inner-is-valid)
then obtain asx where Entry – asx→ι ax+ source node ax
by (fastforce dest: Entry-path)
with (sourcenode ax ≠ Exit)
have ∃ es. CFG_CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node Entry) es (Node (sourcenode ax))
by (fastforce intro: lift-valid-path)
then obtain es where CFG_CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node Entry) es (Node (sourcenode ax)) by blast
with \( \text{slp-edge} \) have \( \text{CFG.CFG.valid-path'}\ src\ trg\ knd \)

\[
(\text{lift-valid-edge valid-edge source-node target-node kind Entry Exit})
\]
\[
(\text{lift-get-return-edges get-return-edges valid-edge source-node target-node kind})
\]
\[
(\text{Node Entry}) \ (es@[(\text{Node (source-node ax),kind ax,Node Exit})]) \ (\text{Node Exit})
\]

by \(-\) (rule \( \text{CFGExit.ep-slp-Append} \))

with \( \text{slp-edge'} \) have \( \text{CFG.CFG.valid-path'}\ src\ trg\ knd \)

\[
(\text{lift-valid-edge valid-edge source-node target-node kind Entry Exit})
\]
\[
(\text{lift-get-return-edges get-return-edges valid-edge source-node target-node kind})
\]
\[
(\text{NewEntry}) \ (es@[(\text{NewEntry,(\(\lambda\)s.True),Node Exit})]) \ (\text{Node Exit})
\]

by (rule \( \text{CFGExit.slp-vp-Append} \))

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

next

case \( \text{inner} \)

have \( \text{lift-valid-edge valid-edge source-node target-node kind Entry Exit} \)
\[
(\text{NewEntry,(\(\lambda\)s.True),Node Entry})
\]

by (fastforce intro:lift-Entry-edge)

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

by (fastforce simp:CFGExit.valid-node-def)

hence \( \text{slp-edge:CFG.CFG.same-level-path'}\ src\ trg\ knd \)

by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path)

from \( \text{valid-node m} \) obtain as where \( \text{Entry} -\text{as}\rightarrow^* m \)

by (fastforce dest:Entry-path)

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

have \( \exists\ es. \text{CFG.CFG.valid-path'}\ src\ trg\ knd \)

by (fastforce intro:lift-valid-path simp:inner-node-def)

then obtain \( es \) where \( \text{CFG.CFG.valid-path'}\ src\ trg\ knd \)

by blast

with \( \text{slp-edge} \) have \( \text{CFG.CFG.valid-path'}\ src\ trg\ knd \)

by (rule \( \text{CFGExit.slp-vp-Append} \))

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

qed
qed

next

fix n assume \textit{CFG.CFG.valid-node src try}
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n

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

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

\textbf{thus} \exists \ as \ \textit{CFG.CFG.valid-path’ src try knd}
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind Entry Exit)

n as \texttt{NewExit} \textbf{apply} --

\textbf{proof}(erule disjE)+

assume n = \texttt{NewEntry}

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,\langle \lambda s. \texttt{False} \rangle \_ \_,\texttt{NewExit}) by(fastforce intro:lve-Entry-Exit-edge)

\textbf{hence} CFG.CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry [[\langle \texttt{NewEntry},\langle \lambda s. \texttt{False} \rangle \_ \_,\texttt{NewExit} \rangle \_ \_,\texttt{NewExit}]] NewExit

by(fastforce dest:CFGExit.path-edge)

with (n = \texttt{NewEntry}) have CFG.CFG.valid-path’ src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind Entry Exit)

n [[\langle \texttt{NewEntry},\langle \lambda s. \texttt{False} \rangle \_ \_,\texttt{NewExit} \rangle \_ \_,\texttt{NewExit}]] NewExit

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

\textbf{thus} \ ?thesis by blast

next

assume n = \texttt{NewExit}

\textbf{hence} CFG.CFG.valid-path’ src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind Entry Exit)

n [[\langle \texttt{NewEntry},\langle \lambda s. \texttt{False} \rangle \_ \_,\texttt{NewExit} \rangle \_ \_,\texttt{NewExit}]] NewExit

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

\textbf{thus} \ ?thesis by blast

next

assume \exists \ m. n = \texttt{Node m} \land \texttt{valid-node m}

then obtain m where n = \texttt{Node m} \textbf{and} \ texttt{valid-node m} \textbf{by} blast

from (valid-node m)

\textbf{show} \ ?thesis

\textbf{proof}(cases m rule:valid-node-cases)

\textbf{case} Entry

from inner obtain ax where valid-edge ax \textbf{and} intra-kind (kind ax) \textbf{and} inner-node (targetnode ax) \textbf{and} sourcenode ax = Entry

by(erule inner-node-entry-edge)

\textbf{hence} lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(Node Entry,kind ax,\texttt{Node (targetnode ax)})

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

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

with (intra-kind (kind ax))

have slp-edge: CFG.CFG.same-level-path’ src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind)
(Node Exit) [(Node Exit, kind ax, Node (targetnode ax))]
(Node (targetnode ax))
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def
intra-kind-def)

have targetnode ax ≠ Entry

proof
assume targetnode ax = Entry
with (valid-edge ax): show False by (rule Entry-target)

qed

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

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

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

from (inner-node (targetnode ax)) have valid-node (targetnode ax)
by (rule inner-is-valid)

then obtain asx where targetnode ax −→ asx∗ Exit
by (fastforce dest: Exit-path)

with (targetnode ax ≠ Entry)

have ∃ es. CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind) (Node (targetnode ax)) es (Node Exit)
by (fastforce intro: lift-valid-path)

then obtain es where CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind) (Node (targetnode ax)) es (Node Exit) by blast

with slp-edge have CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind)
(Node Entry) \(((\text{Node Entry,kind ax,Node (targetnode ax))})@es\) (Node Exit)

by (rule \text{CFGExit}.slp-vp-Append)

with slp-edge' have \text{CFG}.\text{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)

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

([Node Exit,(\lambda s.\text{True})\ join\ NewExit]) NewExit
by − (rule \text{CFGExit}.vp-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,(\lambda s.\text{True})\ join\ NewExit) by (fastforce intro:lev-Exit-edge)

with (m = Exit) (n = Node m) have \text{CFG}.\text{CFG}.path src try
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)

(n (\text{Node Exit,(\lambda s.\text{True})\ join\ NewExit})) NewExit
by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path
simp:CFGExit.valid-node-def)

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

next

case inner

have lift-valid-edge valid-edge source-node target-node kind Entry Exit
(Node Exit,(\lambda s.\text{True})\ join\ NewExit) by (fastforce intro:lev-Exit-edge)

hence \text{CFG}.path src try
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)

(Node Exit) ([Node Exit,(\lambda s.\text{True})\ join\ 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,(\lambda s.\text{True})\ join\ 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 \exists es, \text{CFG}.\text{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 \text{CFG}.\text{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 \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) (Node m) (es@[(Node Exit,(λs. True)⊥,NewExit)]) NewExit
by -(rule CFGExit.op-slp-Append)
with (n = Node m) show ?thesis by simp blast
qed
qed
next
fix n n'
assume method-exit1:CFGExit.CFGExit.method-exit src knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n
and method-exit2:CFGExit.CFGExit.method-exit src knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n'
and lift-eq:lift-get-proc get-proc Main n = lift-get-proc get-proc Main n'
from method-exit1 show n = n'
proof(rule CFGExit.method-exit-cases)
assume n = NewExit
from method-exit2 show ?thesis
proof(rule CFGExit.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 sourcenode targetnode kind Entry Exit a
and knd a = Q←pf
hence 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←pf) have knd a = Q←Mainf 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
qed
next
fix a Q f p
assume n = src a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q←pf
then obtain x where valid-edge x and src a = Node (sourcenode x)
and kind x = Q←pf
by(fastforce elim:lift-valid-edge.cases)
hence method-exit (sourcenode x) by(fastforce simp:method-exit-def)
from method-exit2 show ?thesis
proof(rule CFGExit.method-exit-cases)
assume n' = NewExit
from (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a)
<knd a = Q←pf>
have lift-get-proc get-proc Main \((\text{src } a) = p\) by -(rule CFGExit.get-proc-return)
with CFGExit.get-proc-Exit lift-eq \((n = \text{src } a) \land (n' = \text{NewExit})\)
have \(p = \text{Main}\) by simp
with \((\text{knd } a = Q \leftarrow pf)\) have \(\text{knd } a = Q \leftarrow \text{Main} f\) by simp
with \((\text{lift-valid-edge valid-edge source-node target-node kind Entry Exit } a)\)
have \(\text{False}\) by (rule CFGExit.Main-no-return-source)
thus \(?\text{thesis}\) by simp
next
fix \(a' \ Q' \ f' \ p'\)
assume \(n' = \text{src } a'\)
and \(\text{lift-valid-edge valid-edge source-node target-node kind Entry Exit } a'\)
and \(\text{knd } a' = Q' \leftarrow pf'\)
then obtain \(x'\) where \(\text{valid-edge } x'\) and \(\text{src } a' = \text{Node } (\text{source-node } x')\)
and \(\text{knd } x' = Q' \leftarrow pf'\)
by (fastforce elim: lift-valid-edge.cases)
hence method-exit \((\text{source-node } x')\) by (fastforce simp: method-exit-def)
with \((\text{method-exit } (\text{source-node } x'))\) lift-eq \((n = \text{src } a' \land (n' = \text{src } a')\)
\((\text{src } a = \text{Node } (\text{source-node } x') \land (\text{src } a' = \text{Node } (\text{source-node } x'))\)
\((n = \text{src } a \land n' = \text{src } a')\)
show ?thesis by simp
qed
qed
qed

lemma lift-SDG:
assumes SDG:SDG source-node target-node kind valid-edge Entry get-proc get-return-edges proc Main Exit Def Use ParamDefs ParamUses
and inner;CFGExit.inner-node source-node target-node kind valid-edge Entry Exit nx
shows SDG src trg knx
\((\text{lift-valid-edge valid-edge source-node target-node kind Entry Exit } a)\) NewEntry
\((\text{lift-get-proc get-proc Main})\)
\((\text{lift-get-return-edges get-return-edges valid-edge source-node target-node kind})\)
\(\text{procs Main NewExit} \ (\text{lift-Def Def Entry Exit } H L) \ (\text{lift-Use Entry Exit } H L)\)
\((\text{lift-ParamDefs ParamDefs}) \ (\text{lift-ParamUses ParamUses})\)
proof –
interpret SDG source-node target-node kind valid-edge Entry get-proc
get-return-edges proc Main Exit Def Use ParamDefs ParamUses
by(rule SDG)
have wf:CFGExit-wf source-node target-node kind valid-edge Entry get-proc
get-return-edges proc Main Exit Def Use ParamDefs ParamUses
by(unfold-locales)
have pt:Postdomination source-node target-node kind valid-edge Entry get-proc
get-return-edges proc Main Exit
by(unfold-locales)
interpret $\text{wf}':\text{CFGExit-wf src trg knd}$

lift-valid-edge valid-edge source-node target-node kind Entry Exit NewEntry
lift-get-proc get-proc Main

lift-get-return-edges get-return-edges valid-edge source-node target-node kind
procs Main NewExit lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L
lift-ParamDefs ParamDefs lift-ParamUses ParamUses
by (fastforce intro: lift-CFGExit-wf $\text{wf}$ $\text{pd}$)

interpret $\text{pd}':\text{Postdomination src trg knd}$

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
by (fastforce intro: lift-Postdomination $\text{wf}$ $\text{pd}$ inner)

show ?thesis by (unfold-locales)

qed

3.2.3 Low-deterministic security via the lifted graph

lemma Lift-NonInterferenceGraph:

fixes valid-edge and source-node and target-node 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 source-node target-node 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 source-node target-node 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 $\text{SDG}': \text{SDG}$ source-node target-node kind valid-edge Entry get-proc
get-return-edges procs Main NewExit Def Use ParamDefs ParamUses

and inner: $\text{CFGExit}.inner-node source-node target-node valid-edge Entry Exit \forall x$
and $H \cap L = \{\} \text{ and } H \cup L = \text{UNIV}$

shows NonInterferenceInterGraph src trg knd $lve$ NewEntry $lget-proc$
lift-return-edges procs Main NewExit $lDef$ $lUse$ $lParamDefs$ $lParamUses$ H L
(Node Entry) (Node Exit)

proof –

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

interpret $\text{SDG}': \text{SDG}$ src trg knd $lve$ NewEntry $lget-proc$ $lget-return-edges$
procs Main NewExit $lDef$ $lUse$ $lParamDefs$ $lParamUses$
by (fastforce intro: lift-SDG $\text{SDG}$ inner simp: $lve$ $lget-proc$ $lget-return-edges$ $lDef$
  $lUse$ $lParamDefs$ $lParamUses$)

show ?thesis

proof

fix $a$ assume $lve \ a$ and src $a = \text{NewEntry}$

thus try $a = \text{NewExit} \lor \ try \ a = \text{Node Entry}$

72
by (fastforce elim: lift-valid-edge.cases simp:lve)  
next  
show \exists a. lve a \land src a = NewEntry \land trg a = Node Entry \land knd a = (\lambda s. 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 \langle e = (Node (sourcenode a), kind a, Node (targetnode a))\rangle  
  ⟨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 \lor src a = Node Exit  
  by (fastforce elim: lift-valid-edge.cases simp:lve)  
next  
show \exists a. lve a \land src a = Node Exit \land trg a = NewExit \land knd a = (\lambda s. 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 \langle e = (Node (sourcenode a), kind a, Node (targetnode a))\rangle  
  ⟨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
\begin{verbatim}
from Entry-noteq-Exit lUse show lUse (Node Exit) = L
by (fastforce elim:lift-Use-set.cases intro:lift-Use-Low)
next
from \(H \cap L = \{\}\) show \(H \cap L = \{\}\).
next
from \(H \cup L = \text{UNIV}\) show \(H \cup L = \text{UNIV}\).
qed
qed
end
\end{verbatim}

References


74