Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

October 26, 2026

Abstract

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

1 Introduction

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

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

2 HRB Slicing guarantees IFC Noninterference

theory *NonInterferenceInter*
  imports *HRB−Slicing.FundamentalProperty*
begin

2.1 Assumptions of this Approach

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

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

locale *NonInterferenceInterGraph* =
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 list* **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 = $(\lambda s.\ True)_{\surd}$
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 = $(\lambda s.\ True)_{\surd}$
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* ∩ *L* = {}
and *HighLowUNIV*: *H* ∪ *L* = *UNIV*

begin

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

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

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

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

lemma *get-proc-High*:
get-proc (*-High-*) = *Main*
proof −
from *High-target-Entry-edge* **obtain** *a* **where** *valid-edge a*
and *sourcenode a* = (*-Entry-*) **and** *targetnode a* = (*-High-*)
and *intra-kind* (*kind a*) **by**(*fastforce simp*:*intra-kind-def*)
from ‹*valid-edge a*› ‹*intra-kind* (*kind a*)›
have *get-proc* (*sourcenode a*) = *get-proc* (*targetnode a*) **by**(*rule get-proc-intra*)
with ‹*sourcenode a* = (*-Entry-*)› ‹*targetnode a* = (*-High-*)› *get-proc-Entry*
show *?thesis* **by** *simp*
qed

lemma *Entry-path-High-path*:
assumes (*-Entry-*) $-as\rightarrow*$ n **and** *inner-node* n
obtains a' as' **where** $as = a'\#as'$ **and** (*-High-*) $-as'\rightarrow*$ n
and *kind* $a' = (\lambda s.\ True)_{\surd}$
proof(*atomize-elim*)
from ‹(*-Entry-*) $-as\rightarrow*$ n› ‹*inner-node* n›
show $\exists a'\ as'.\ as = a'\#as' \wedge$ (*-High-*) $-as'\rightarrow* n \wedge$ *kind* $a' = (\lambda s.\ True)_{\surd}$
proof(*induct* $n'\equiv$(*-Entry-*) *as* n *rule*:*path.induct*)
case (*Cons-path* n'' *as* n' a)
from ‹$n'' -as\rightarrow* n'$› ‹*inner-node* n'› **have** $n'' \neq$ (*-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* $a' = (\lambda s.\ True)_{\surd}$
by *blast*
with ‹*valid-edge* a› ‹*sourcenode* a = (*-Entry-*)› ‹*targetnode* $a = n''$›
‹n'' = (*-High-*)›
have $a = a'$ **by**(*auto dest*:*edge-det*)
with ‹$n'' -as\rightarrow* n'$› ‹n'' = (*-High-*)› ‹*kind* $a' = (\lambda s.\ True)_{\surd}$› **show** *?case* **by** *blast*
qed *fastforce*
qed

lemma *Exit-path-Low-path*:
assumes $n\ -as\rightarrow*$ (*-Exit-*) **and** *inner-node* n
obtains a' as' **where** $as = as'@[a']$ **and** $n\ -as'\rightarrow*$ (*-Low-*)
and *kind* $a' = (\lambda s.\ True)_{\surd}$
proof(*atomize-elim*)
from ‹$n\ -as\rightarrow*$ (*-Exit-*)›
show $\exists as'\ a'.\ as = as'@[a'] \wedge n\ -as'\rightarrow*$ (*-Low-*) $\wedge$ *kind* $a' = (\lambda s.\ True)_{\surd}$
proof(*induct as rule*:*rev-induct*)
case *Nil*
with ‹*inner-node* n› **show** *?case* **by** *fastforce*
next
case (*snoc* a' *as'*)
from ‹$n\ -as'@[a']\rightarrow*$ (*-Exit-*)›
have $n\ -as'\rightarrow*$ *sourcenode* a' **and** *valid-edge* a' **and** *targetnode* a' = (*-Exit-*)
by(*auto elim*:*path-split-snoc*)
{ assume *sourcenode* a' = (*-Entry-*)
with ‹$n\ -as'\rightarrow*$ *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*!:*Exit-edge-Entry-or-Low*)
from *Low-source-Exit-edge*
obtain *ax* **where** *valid-edge ax* **and** *sourcenode ax* = (*-Low-*)

and *targetnode ax* = (*-Exit-*) **and** *kind ax* = $(\lambda s.\ True)_{\surd}$
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* = $(\lambda s.\ True)_{\surd}$›
show *?case* **by** *blast*
qed
qed

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

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

2.2 Low Equivalence

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

definition *lowEquivalence* :: ('*var* ⇀ '*val*) *list* ⇒ ('*var* ⇀ '*val*) *list* ⇒ *bool*
(**infixl** ‹$\approx_L$› *50*)
where $s \approx_L s' \equiv \forall V \in L.\ hd\ s\ V = hd\ s'\ V$

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

lemma *relevant-vars-Entry*:
assumes $V \in rv\ S$ (*CFG-node* (*-Entry-*)) **and** (*-High-*) $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$
shows $V \in L$
proof −
from ‹$V \in rv\ S$ (*CFG-node* (*-Entry-*))› **obtain** *as n'*
where (*-Entry-*) −*as*→$_\iota$∗ *parent-node n'*
and *n'* ∈ *HRB-slice S* **and** $V \in Use_{SDG}\ n'$
and ∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes as*)
⟶ $V \notin Def_{SDG}\ n''$ **by**(*fastforce elim*:*rvE*)
from ‹(*-Entry-*) −*as*→$_\iota$∗ *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 ‹$V \in Use_{SDG}\ n'$› **have** *False*
by −(*drule SDG-Use-parent-Use*,*simp add*:*Entry-empty*)
thus *?thesis* **by** *simp*

```
  next
    case Exit
    with ‹V ∈ Use_SDG n'› have False
      by −(drule SDG-Use-parent-Use,simp add:Exit-empty)
    thus ?thesis by simp
  next
    case inner
    with ‹(-Entry-) −as→_ι* parent-node n'› obtain a' as' where as = a'#as'
      and (-High-) −as'→_ι* parent-node n'
      by(fastforce elim:Entry-path-High-path simp:intra-path-def)
    from ‹(-Entry-) −as→_ι* parent-node n'› ‹as = a'#as'›
   have sourcenode a' = (-Entry-) by(fastforce elim:path.cases simp:intra-path-def)
    show ?thesis
    proof(cases as' = [])
      case True
      with ‹(-High-) −as'→_ι* parent-node n'› have parent-node n' = (-High-)
        by(fastforce simp:intra-path-def)
      with ‹n' ∈ HRB-slice S› ‹(-High-) ∉ ⌊HRB-slice S⌋_CFG›
      have False
        by(fastforce dest:valid-SDG-node-in-slice-parent-node-in-slice
                  simp:SDG-to-CFG-set-def)
      thus ?thesis by simp
    next
      case False
     with ‹(-High-) −as'→_ι* parent-node n'› have hd (sourcenodes as') = (-High-)
        by(fastforce intro:path-sourcenode simp:intra-path-def)
      from False have hd (sourcenodes as') ∈ set (sourcenodes as')
        by(fastforce intro:hd-in-set simp:sourcenodes-def)
      with ‹as = a'#as'› have hd (sourcenodes as') ∈ set (sourcenodes as)
        by(simp add:sourcenodes-def)
      from ‹hd (sourcenodes as') = (-High-)›
      have valid-node (hd (sourcenodes as')) by simp
      have valid-SDG-node (CFG-node (-High-)) by simp
      with ‹hd (sourcenodes as') = (-High-)›
        ‹hd (sourcenodes as') ∈ set (sourcenodes as)›
        ‹∀ n''. valid-SDG-node n'' ∧ parent-node n'' ∈ set (sourcenodes as)
        ⟶ V ∉ Def_SDG n''›
      have V ∉ Def (-High-)
      by(fastforce dest:CFG-Def-SDG-Def[OF ‹valid-node (hd (sourcenodes as'))›])
      hence V ∉ H by(simp add:DefHigh)
      thus ?thesis by(rule not-High-Low)
    qed
  qed
qed


lemma lowEquivalence-relevant-nodes-Entry:
  assumes s ≈_L s' and (-High-) ∉ ⌊HRB-slice S⌋_CFG
```

shows $\forall\, V \in rv\ S\ (\textit{CFG-node}\ (\textit{-Entry-}))$. $hd\ s\ V = hd\ s'\ V$
proof
fix V **assume** $V \in rv\ S\ (\textit{CFG-node}\ (\textit{-Entry-}))$
with ‹$(\textit{-High-}) \notin \lfloor \textit{HRB-slice}\ S \rfloor_{\textit{CFG}}$› **have** $V \in L$ **by** $-(\textit{rule relevant-vars-Entry})$
with ‹$s \approx_L s'$› **show** $hd\ s\ V = hd\ s'\ V$ **by**$(\textit{simp add:lowEquivalence-def})$
qed

2.3 The Correctness Proofs

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

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

lemma *slpa-rv-Low-Use-Low*:
assumes $\textit{CFG-node}\ (\textit{-Low-}) \in S$
shows $[\![\textit{same-level-path-aux cs as};\ \textit{upd-cs cs as} = [];\ \textit{same-level-path-aux cs as}';$
$\forall\, c \in \textit{set cs}.\ \textit{valid-edge}\ c;\ m -as\rightarrow* (\textit{-Low-});\ m -as'\rightarrow* (\textit{-Low-});$
$\forall\, i < \textit{length cs}.\ \forall\, V \in rv\ S\ (\textit{CFG-node}\ (\textit{sourcenode}\ (cs!i)))).$
$\textit{fst}\ (s!\textit{Suc}\ i)\ V = \textit{fst}\ (s'!\textit{Suc}\ i)\ V;\ \forall\, i < \textit{Suc}\ (\textit{length cs}).\ \textit{snd}\ (s!i) = \textit{snd}\ (s'!i);$
$\forall\, V \in rv\ S\ (\textit{CFG-node}\ m).\ \textit{state-val}\ s\ V = \textit{state-val}\ s'\ V;$
$\textit{preds}\ (\textit{slice-kinds}\ S\ as)\ s;\ \textit{preds}\ (\textit{slice-kinds}\ S\ as')\ s';$
$\textit{length}\ s = \textit{Suc}\ (\textit{length cs});\ \textit{length}\ s' = \textit{Suc}\ (\textit{length cs})[\!]$
$\Longrightarrow \forall\, V \in \textit{Use}\ (\textit{-Low-}).\ \textit{state-val}\ (\textit{transfers}(\textit{slice-kinds}\ S\ as)\ s)\ V =$
$\textit{state-val}\ (\textit{transfers}(\textit{slice-kinds}\ S\ as')\ s')\ V$
proof$(\textit{induct arbitrary:m as' s s' rule:slpa-induct})$
case $(\textit{slpa-empty cs})$
from ‹$m -[]\rightarrow* (\textit{-Low-})$› **have** $m = (\textit{-Low-})$ **by** *fastforce*
from ‹$m -[]\rightarrow* (\textit{-Low-})$› **have** *valid-node m*
by$(\textit{rule path-valid-node})+$
{ fix V **assume** $V \in \textit{Use}\ (\textit{-Low-})$
moreover
from ‹*valid-node m*› ‹$m = (\textit{-Low-})$› **have** $(\textit{-Low-}) -[]\rightarrow_\iota* (\textit{-Low-})$
by$(\textit{fastforce intro:empty-path simp:intra-path-def})$
moreover
from ‹*valid-node m*› ‹$m = (\textit{-Low-})$› ‹$\textit{CFG-node}\ (\textit{-Low-}) \in S$›
have $\textit{CFG-node}\ (\textit{-Low-}) \in \textit{HRB-slice}\ S$
by$(\textit{fastforce intro:HRB-slice-refl})$
ultimately have $V \in rv\ S\ (\textit{CFG-node}\ m)$
using ‹$m = (\textit{-Low-})$›
by$(\textit{auto intro!:rvI CFG-Use-SDG-Use simp:sourcenodes-def})$ **}**
hence $\forall\, V \in \textit{Use}\ (\textit{-Low-}).\ V \in rv\ S\ (\textit{CFG-node}\ m)$ **by** *simp*
show *?case*
proof$(\textit{cases}\ L = \{\})$
case *True* **with** *UseLow* **show** *?thesis* **by** *simp*
next

```
    case False
    from ‹m −as'→∗ (-Low-)› ‹m = (-Low-)› have as' = []
    proof(induct m as' m'≡(-Low-) rule:path.induct)
      case (Cons-path m'' as a m)
      from ‹valid-edge a› ‹sourcenode a = m› ‹m = (-Low-)›
      have targetnode a = (-Exit-) by −(rule Exit-successor-of-Low,simp+)
      with ‹targetnode a = m''› ‹m'' −as→∗ (-Low-)›
      have (-Low-) = (-Exit-) by −(drule path-Exit-source,auto)
      with False have False by −(drule Low-neq-Exit,simp)
      thus ?case by simp
    qed simp
    with ‹∀ V ∈ Use (-Low-). V ∈ rv S (CFG-node m)›
      ‹∀ V ∈ rv S (CFG-node m). state-val s V = state-val s' V› Nil
    show ?thesis by(auto simp:slice-kinds-def)
  qed
next
  case (slpa-intra cs a as)
  note IH = ‹⋀m as' s s'. ⟦upd-cs cs as = []; same-level-path-aux cs as';
    ∀ a∈set cs. valid-edge a; m −as→∗ (-Low-); m −as'→∗ (-Low-);
    ∀ i<length cs. ∀ V∈rv S (CFG-node (sourcenode (cs ! i))).
    fst (s ! Suc i) V = fst (s' ! Suc i) V;
    ∀ i<Suc (length cs). snd (s ! i) = snd (s' ! i);
    ∀ V∈rv S (CFG-node m). state-val s V = state-val s' V;
    preds (slice-kinds S as) s; preds (slice-kinds S as') s';
    length s = Suc (length cs); length s' = Suc (length cs)⟧
    ⟹ ∀ V∈Use (-Low-). state-val (transfers(slice-kinds S as) s) V =
    state-val (transfers(slice-kinds S as') s') V›
  note rvs = ‹∀ i<length cs. ∀ V∈rv S (CFG-node (sourcenode (cs ! i))).
    fst (s ! Suc i) V = fst (s' ! Suc i) V›
  from ‹m −a # as→∗ (-Low-)› have sourcenode a = m and valid-edge a
    and targetnode a −as→∗ (-Low-) by(auto elim:path-split-Cons)
  show ?case
  proof(cases L = {})
    case True with UseLow show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases as')
      case Nil
      with ‹m −as'→∗ (-Low-)› have m = (-Low-) by fastforce
      with ‹valid-edge a› ‹sourcenode a = m› have targetnode a = (-Exit-)
        by −(rule Exit-successor-of-Low,simp+)
      from Low-source-Exit-edge obtain a' where valid-edge a'
        and sourcenode a' = (-Low-) and targetnode a' = (-Exit-)
        and kind a' = (λs. True)√ by blast
      from ‹valid-edge a› ‹sourcenode a = m› ‹m = (-Low-)›
        ‹targetnode a = (-Exit-)› ‹valid-edge a'› ‹sourcenode a' = (-Low-)›
        ‹targetnode a' = (-Exit-)›
      have a = a' by(fastforce dest:edge-det)
```

```
  with ‹kind a' = (λs. True)√› have kind a = (λs. True)√ by simp
  with ‹targetnode a = (-Exit-)› ‹targetnode a −as→* (-Low-)›
  have (-Low-) = (-Exit-) by −(drule path-Exit-source,auto)
  with False have False by −(drule Low-neq-Exit,simp)
  thus ?thesis by simp
next
  case (Cons ax asx)
  with ‹m −as'→* (-Low-)› have sourcenode ax = m and valid-edge ax
    and targetnode ax −asx→* (-Low-) by(auto elim:path-split-Cons)
  from ‹preds (slice-kinds S (a # as)) s›
 obtain cf cfs where [simp]:s = cf#cfs by(cases s)(auto simp:slice-kinds-def)
  from ‹preds (slice-kinds S as') s'› ‹as' = ax # asx›
  obtain cf' cfs' where [simp]:s' = cf'#cfs'
    by(cases s')(auto simp:slice-kinds-def)
  have intra-kind (kind ax)
  proof(cases kind ax rule:edge-kind-cases)
    case (Call Q r p fs)
    have False
    proof(cases sourcenode a ∈ ⌊HRB-slice S⌋CFG)
      case True
      with ‹intra-kind (kind a)› have slice-kind S a = kind a
        by −(rule slice-intra-kind-in-slice)
      from ‹valid-edge ax› ‹kind ax = Q:r↪pfs›
      have unique:∃!a'. valid-edge a' ∧ sourcenode a' = sourcenode ax ∧
        intra-kind(kind a') by(rule call-only-one-intra-edge)
      from ‹valid-edge ax› ‹kind ax = Q:r↪pfs› obtain x
        where x ∈ get-return-edges ax by(fastforce dest:get-return-edge-call)
      with ‹valid-edge ax› obtain a' where valid-edge a'
        and sourcenode a' = sourcenode ax and kind a' = (λcf. False)√
        by(fastforce dest:call-return-node-edge)
      with ‹valid-edge a› ‹sourcenode a = m› ‹sourcenode ax = m›
        ‹intra-kind (kind a)› unique
      have a' = a by(fastforce simp:intra-kind-def)
      with ‹kind a' = (λcf. False)√› ‹slice-kind S a = kind a›
        ‹preds (slice-kinds S (a#as)) s›
      have False by(cases s)(auto simp:slice-kinds-def)
      thus ?thesis by simp
    next
      case False
      with ‹kind ax = Q:r↪pfs› ‹sourcenode a = m› ‹sourcenode ax = m›
      have slice-kind S ax = (λcf. False):r↪pfs
        by(fastforce intro:slice-kind-Call)
      with ‹as' = ax # asx› ‹preds (slice-kinds S as') s'›
      have False by(cases s')(auto simp:slice-kinds-def)
      thus ?thesis by simp
    qed
    thus ?thesis by simp
  next
    case (Return Q p f)
```

```
    from ‹valid-edge ax› ‹kind ax = Q↪pf› ‹valid-edge a› ‹intra-kind (kind a)›
      ‹sourcenode a = m› ‹sourcenode ax = m›
    have False by −(drule return-edges-only,auto simp:intra-kind-def)
    thus ?thesis by simp
  qed simp
  with ‹same-level-path-aux cs as'› ‹as' = ax#asx›
  have same-level-path-aux cs asx by(fastforce simp:intra-kind-def)
  show ?thesis
  proof(cases targetnode a = targetnode ax)
    case True
    with ‹valid-edge a› ‹valid-edge ax› ‹sourcenode a = m› ‹sourcenode ax =
m›
    have a = ax by(fastforce intro:edge-det)
    with ‹valid-edge a› ‹intra-kind (kind a)› ‹sourcenode a = m›
      ‹∀ V∈rv S (CFG-node m). state-val s V = state-val s' V›
      ‹preds (slice-kinds S (a # as)) s›
      ‹preds (slice-kinds S as') s'› ‹as' = ax # asx›
    have rv:∀ V∈rv S (CFG-node (targetnode a)).
      state-val (transfer (slice-kind S a) s) V =
      state-val (transfer (slice-kind S a) s') V
      by −(rule rv-edge-slice-kinds,auto)
    from ‹upd-cs cs (a # as) = []› ‹intra-kind (kind a)›
    have upd-cs cs as = [] by(fastforce simp:intra-kind-def)
    from ‹targetnode ax −asx→* (-Low-)› ‹a = ax›
    have targetnode a −asx→* (-Low-) by simp
    from ‹valid-edge a› ‹intra-kind (kind a)›
    obtain cfx
      where cfx:transfer (slice-kind S a) s = cfx#cfs ∧ snd cfx = snd cf
      apply(cases cf)
      apply(cases sourcenode a ∈ ⌊HRB-slice S⌋CFG) apply auto
      apply(fastforce dest:slice-intra-kind-in-slice simp:intra-kind-def)
      apply(auto simp:intra-kind-def)
      apply(drule slice-kind-Upd) apply auto
      by(erule kind-Predicate-notin-slice-slice-kind-Predicate) auto
    from ‹valid-edge a› ‹intra-kind (kind a)›
    obtain cfx'
      where cfx':transfer (slice-kind S a) s' = cfx'#cfs' ∧ snd cfx' = snd cf'
      apply(cases cf')
      apply(cases sourcenode a ∈ ⌊HRB-slice S⌋CFG) apply auto
      apply(fastforce dest:slice-intra-kind-in-slice simp:intra-kind-def)
      apply(auto simp:intra-kind-def)
      apply(drule slice-kind-Upd) apply auto
      by(erule kind-Predicate-notin-slice-slice-kind-Predicate) auto
    with cfx ‹∀ i < Suc (length cs). snd (s!i) = snd (s'!i)›
    have snds:∀ i<Suc(length cs).
      snd (transfer (slice-kind S a) s ! i) =
      snd (transfer (slice-kind S a) s' ! i)
      by auto(case-tac i,auto)
    from rvs cfx cfx' have rvs':∀ i<length cs.
```

```
    ∀ V∈rv S (CFG-node (sourcenode (cs ! i))).
    fst (transfer (slice-kind S a) s ! Suc i) V =
    fst (transfer (slice-kind S a) s' ! Suc i) V
    by fastforce
  from ‹preds (slice-kinds S (a # as)) s›
  have preds (slice-kinds S as)
    (transfer (slice-kind S a) s) by(simp add:slice-kinds-def)
  moreover
  from ‹preds (slice-kinds S as') s'› ‹as' = ax # asx› ‹a = ax›
  have preds (slice-kinds S asx) (transfer (slice-kind S a) s')
    by(simp add:slice-kinds-def)
  moreover
  from ‹valid-edge a› ‹intra-kind (kind a)›
  have length (transfer (slice-kind S a) s) = length s
    by(cases sourcenode a ∈ ⌊HRB-slice S⌋CFG)
  (auto dest:slice-intra-kind-in-slice slice-kind-Upd
    elim:kind-Predicate-notin-slice-slice-kind-Predicate simp:intra-kind-def)
  with ‹length s = Suc (length cs)›
  have length (transfer (slice-kind S a) s) = Suc (length cs)
    by simp
  moreover
  from ‹a = ax› ‹valid-edge a› ‹intra-kind (kind a)›
  have length (transfer (slice-kind S a) s') = length s'
    by(cases sourcenode ax ∈ ⌊HRB-slice S⌋CFG)
  (auto dest:slice-intra-kind-in-slice slice-kind-Upd
    elim:kind-Predicate-notin-slice-slice-kind-Predicate simp:intra-kind-def)
  with ‹length s' = Suc (length cs)›
  have length (transfer (slice-kind S a) s') = Suc (length cs)
    by simp
  moreover
  from IH[OF ‹upd-cs cs as = []› ‹same-level-path-aux cs asx›
    ‹∀ c∈set cs. valid-edge c› ‹targetnode a −as→∗ (-Low-)›
    ‹targetnode a −asx→∗ (-Low-)› rvs' snds rv calculation]
    ‹as' = ax # asx› ‹a = ax›
  show ?thesis by(simp add:slice-kinds-def)
next
  case False
  from ‹∀ i < Suc(length cs). snd (s!i) = snd (s'!i)›
  have snd (hd s) = snd (hd s') by(erule-tac x=0 in allE) fastforce
  with ‹valid-edge a› ‹valid-edge ax› ‹sourcenode a = m›
    ‹sourcenode ax = m› ‹as' = ax # asx› False
    ‹intra-kind (kind a)› ‹intra-kind (kind ax)›
    ‹preds (slice-kinds S (a # as)) s›
    ‹preds (slice-kinds S as') s'›
    ‹∀ V∈rv S (CFG-node m). state-val s V = state-val s' V›
    ‹length s = Suc (length cs)› ‹length s' = Suc (length cs)›
  have False by(fastforce intro!:rv-branching-edges-slice-kinds-False[of a ax])
  thus ?thesis by simp
qed
```

```
    qed
  qed
next
  case (slpa-Call cs a as Q r p fs)
  note IH = ‹⋀m as' s s'.
    ⟦upd-cs (a # cs) as = []; same-level-path-aux (a # cs) as';
    ∀ c∈set (a # cs). valid-edge c; m −as→* (-Low-); m −as'→* (-Low-);
    ∀ i<length (a # cs). ∀ V∈rv S (CFG-node (sourcenode ((a # cs) ! i))).
    fst (s ! Suc i) V = fst (s' ! Suc i) V;
    ∀ i<Suc (length (a # cs)). snd (s ! i) = snd (s' ! i);
    ∀ V∈rv S (CFG-node m). state-val s V = state-val s' V;
    preds (slice-kinds S as) s; preds (slice-kinds S as') s';
    length s = Suc (length (a # cs)); length s' = Suc (length (a # cs))⟧
    ⟹ ∀ V∈Use (-Low-). state-val (transfers(slice-kinds S as) s) V =
    state-val (transfers(slice-kinds S as') s') V›
  note rvs = ‹∀ i<length cs. ∀ V∈rv S (CFG-node (sourcenode (cs ! i))).
    fst (s ! Suc i) V = fst (s' ! Suc i) V›
  from ‹m −a # as→* (-Low-)› have sourcenode a = m and valid-edge a
    and targetnode a −as→* (-Low-) by(auto elim:path-split-Cons)
  from ‹∀ c∈set cs. valid-edge c› ‹valid-edge a›
  have ∀ c∈set (a # cs). valid-edge c by simp
  show ?case
   proof(cases L = {})
    case True with UseLow show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases as')
      case Nil
      with ‹m −as'→* (-Low-)› have m = (-Low-) by fastforce
      with ‹valid-edge a› ‹sourcenode a = m› have targetnode a = (-Exit-)
        by −(rule Exit-successor-of-Low,simp+)
      from Low-source-Exit-edge obtain a' where valid-edge a'
        and sourcenode a' = (-Low-) and targetnode a' = (-Exit-)
        and kind a' = (λs. True)√ by blast
      from ‹valid-edge a› ‹sourcenode a = m› ‹m = (-Low-)›
        ‹targetnode a = (-Exit-)› ‹valid-edge a'› ‹sourcenode a' = (-Low-)›
        ‹targetnode a' = (-Exit-)›
      have a = a' by(fastforce dest:edge-det)
      with ‹kind a' = (λs. True)√› have kind a = (λs. True)√ by simp
      with ‹targetnode a = (-Exit-)› ‹targetnode a −as→* (-Low-)›
      have (-Low-) = (-Exit-) by −(drule path-Exit-source,auto)
      with False have False by −(drule Low-neq-Exit,simp)
      thus ?thesis by simp
    next
      case (Cons ax asx)
      with ‹m −as'→* (-Low-)› have sourcenode ax = m and valid-edge ax
        and targetnode ax −asx→* (-Low-) by(auto elim:path-split-Cons)
      from ‹preds (slice-kinds S (a # as)) s›
```

obtain *cf cfs* **where** [*simp*]:$s = cf \# cfs$ **by**(*cases s*)(*auto simp*:*slice-kinds-def*)
from ‹*preds* (*slice-kinds S as'*) *s'*› ‹$as' = ax \# asx$›
obtain *cf' cfs'* **where** [*simp*]:$s' = cf' \# cfs'$
by(*cases s'*)(*auto simp*:*slice-kinds-def*)
have $\exists\, Q\ r\ p\ fs.$ *kind ax* = Q:$r \hookrightarrow_p fs$
proof(*cases kind ax rule*:*edge-kind-cases*)
case *Intra*
have *False*
proof(*cases sourcenode ax* $\in \lfloor$*HRB-slice S*$\rfloor_{CFG}$)
case *True*
with ‹*intra-kind* (*kind ax*)›
have *slice-kind S ax* = *kind ax*
by −(*rule slice-intra-kind-in-slice*)
from ‹*valid-edge a*› ‹*kind a* = Q:$r \hookrightarrow_p fs$›
have *unique*:$\exists!a'.$ *valid-edge a'* $\wedge$ *sourcenode a'* = *sourcenode a* $\wedge$
intra-kind(*kind a'*) **by**(*rule call-only-one-intra-edge*)
from ‹*valid-edge a*› ‹*kind a* = Q:$r \hookrightarrow_p fs$› **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'* = $(\lambda cf.\ False)_{\surd}$
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'* = $(\lambda cf.\ False)_{\surd}$›
‹*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:$r \hookrightarrow_p fs$› ‹*sourcenode ax* = *m*› ‹*sourcenode a* = *m*›
have *slice-kind S a* = $(\lambda cf.\ False)$:$r \hookrightarrow_p fs$
by(*fastforce intro*:*slice-kind-Call*)
with ‹*preds* (*slice-kinds S* (*a* # *as*)) *s*›
have *False* **by**(*simp add*:*slice-kinds-def*)
thus *?thesis* **by** *simp*
qed
thus *?thesis* **by** *simp*
next
case (*Return Q' p' f'*)
from ‹*valid-edge ax*› ‹*kind ax* = $Q' \leftarrow_{p'} f'$› ‹*valid-edge a*› ‹*kind a* = Q:$r \hookrightarrow_p fs$›
‹*sourcenode a* = *m*› ‹*sourcenode ax* = *m*›
have *False* **by** −(*drule return-edges-only*,*auto*)
thus *?thesis* **by** *simp*
qed *simp*
have *sourcenode a* ∈ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$
proof(*rule ccontr*)
assume *sourcenode a* ∉ $\lfloor$*HRB-slice S*$\rfloor_{CFG}$

from *this* ‹*kind a = Q:r↪pfs*›
have *slice-kind S a = (λcf. False):r↪pfs*
by(*rule slice-kind-Call*)
with ‹*preds* (*slice-kinds S* (*a # as*)) *s*›
show *False* **by**(*simp add:slice-kinds-def*)
qed
with ‹*preds* (*slice-kinds S* (*a # as*)) *s*› ‹*kind a = Q:r↪pfs*›
have *pred* (*kind a*) *s*
by(*fastforce dest:slice-kind-Call-in-slice simp:slice-kinds-def*)
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋*CFG*›
‹*sourcenode a = m*› ‹*sourcenode ax = m*›
have *sourcenode ax* ∈ ⌊*HRB-slice S*⌋*CFG* **by** *simp*
with ‹*as′ = ax # asx*› ‹*preds* (*slice-kinds S as′*) *s′*›
‹∃ *Q r p fs. kind ax = Q:r↪pfs*›
have *pred* (*kind ax*) *s′*
by(*fastforce dest:slice-kind-Call-in-slice simp:slice-kinds-def*)
{ fix *V* **assume** *V* ∈ *Use* (*sourcenode a*)
from ‹*valid-edge a*› **have** *sourcenode a −[]→ι∗ sourcenode a*
by(*fastforce intro:empty-path simp:intra-path-def*)
with ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋*CFG*›
‹*valid-edge a*› ‹*V* ∈ *Use* (*sourcenode a*)›
have *V* ∈ *rv S* (*CFG-node* (*sourcenode a*))
by(*auto intro!:rvI CFG-Use-SDG-Use simp:SDG-to-CFG-set-def sourcenodes-def*)
}
with ‹∀ *V*∈*rv S* (*CFG-node m*). *state-val s V = state-val s′ V*›
‹*sourcenode a = m*›
have *Use*:∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V = state-val s′ V* **by** *simp*
from ‹∀ *i*<*Suc* (*length cs*). *snd* (*s ! i*) = *snd* (*s′ ! i*)›
have *snd* (*hd s*) = *snd* (*hd s′*) **by** *fastforce*
with ‹*valid-edge a*› ‹*kind a = Q:r↪pfs*› ‹*valid-edge ax*›
‹∃ *Q r p fs. kind ax = Q:r↪pfs*› ‹*sourcenode a = m*› ‹*sourcenode ax = m*›
‹*pred* (*kind a*) *s*› ‹*pred* (*kind ax*) *s′*› *Use* ‹*length s = Suc* (*length cs*)›
‹*length s′ = Suc* (*length cs*)›
have [*simp*]:*ax = a* **by**(*fastforce intro!:CFG-equal-Use-equal-call*)
from ‹*same-level-path-aux cs as′*› ‹*as′ = ax#asx*› ‹*kind a = Q:r↪pfs*›
‹∃ *Q r p fs. kind ax = Q:r↪pfs*›
have *same-level-path-aux* (*a # cs*) *asx* **by** *simp*
from ‹*targetnode ax −asx→∗* (*-Low-*)› **have** *targetnode a −asx→∗* (*-Low-*)
by *simp*
from ‹*kind a = Q:r↪pfs*› ‹*upd-cs cs* (*a # as*) = []›
have *upd-cs* (*a # cs*) *as* = [] **by** *simp*
from ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋*CFG*› ‹*kind a = Q:r↪pfs*›
have *slice-kind*:*slice-kind S a* =
Q:*r*↪*p*(*cspp* (*targetnode a*) (*HRB-slice S*) *fs*)
by(*rule slice-kind-Call-in-slice*)
from ‹∀ *i*<*Suc* (*length cs*). *snd* (*s ! i*) = *snd* (*s′ ! i*)› *slice-kind*
have *snds*:∀ *i*<*Suc* (*length* (*a # cs*)).
snd (*transfer* (*slice-kind S a*) *s ! i*) =
snd (*transfer* (*slice-kind S a*) *s′ ! i*)

by *auto*(*case-tac i*,*auto*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› **obtain** *ins outs*
where (*p*,*ins*,*outs*) ∈ *set procs* **by**(*fastforce dest*!:*callee-in-procs*)
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have *length* (*ParamUses* (*sourcenode a*)) = *length ins*
by(*fastforce intro*:*ParamUses-call-source-length*)
with ‹*valid-edge a*›
have ∀ *i* < *length ins*. ∀ *V* ∈ (*ParamUses* (*sourcenode a*))!*i*. *V* ∈ *Use* (*sourcenode a*)
by(*fastforce intro*:*ParamUses-in-Use*)
with ‹∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s*′ *V*›
have ∀ *i* < *length ins*. ∀ *V* ∈ (*ParamUses* (*sourcenode a*))!*i*.
state-val s V = *state-val s*′ *V*
by *fastforce*
with ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
‹*pred* (*kind a*) *s*› ‹*pred* (*kind ax*) *s*′›
have ∀ *i* < *length ins*. (*params fs* (*fst* (*hd s*)))!*i* = (*params fs* (*fst* (*hd s*′)))!*i*
by(*fastforce intro*!:*CFG-call-edge-params*)
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹(*p*,*ins*,*outs*) ∈ *set procs*›
have *length fs* = *length ins* **by**(*rule CFG-call-edge-length*)
{ fix *i* **assume** *i* < *length fs*
with ‹*length fs* = *length ins*› **have** *i* < *length ins* **by** *simp*
from ‹*i* < *length fs*› **have** (*params fs* (*fst cf*))!*i* = (*fs*!*i*) (*fst cf*)
by(*rule params-nth*)
moreover
from ‹*i* < *length fs*› **have** (*params fs* (*fst cf*′))!*i* = (*fs*!*i*) (*fst cf*′)
by(*rule params-nth*)
ultimately have (*fs*!*i*) (*fst* (*hd s*)) = (*fs*!*i*) (*fst* (*hd s*′))
using ‹*i* < *length ins*›
‹∀ *i* < *length ins*. (*params fs* (*fst* (*hd s*)))!*i* = (*params fs* (*fst* (*hd s*′)))!*i*›
by *simp* **}**
hence ∀ *i* < *length fs*. (*fs* ! *i*) (*fst cf*) = (*fs* ! *i*) (*fst cf*′) **by** *simp*
{ fix *i* **assume** *i* < *length fs*
with ‹∀ *i* < *length fs*. (*fs* ! *i*) (*fst cf*) = (*fs* ! *i*) (*fst cf*′)›
have (*fs* ! *i*) (*fst cf*) = (*fs* ! *i*) (*fst cf*′) **by** *simp*
have ((*csppa* (*targetnode a*) (*HRB-slice S*) *0 fs*)!*i*)(*fst cf*) =
((*csppa* (*targetnode a*) (*HRB-slice S*) *0 fs*)!*i*)(*fst cf*′)
proof(*cases Formal-in*(*targetnode a*,*i* + *0*) ∈ *HRB-slice S*)
case *True*
with ‹*i* < *length fs*›
have (*csppa* (*targetnode a*) (*HRB-slice S*) *0 fs*)!*i* = *fs*!*i*
by(*rule csppa-Formal-in-in-slice*)
with ‹(*fs* ! *i*) (*fst cf*) = (*fs* ! *i*) (*fst cf*′)› **show** *?thesis* **by** *simp*
next
case *False*
with ‹*i* < *length fs*›
have (*csppa* (*targetnode a*) (*HRB-slice S*) *0 fs*)!*i* = *Map.empty*
by(*rule csppa-Formal-in-notin-slice*)
thus *?thesis* **by** *simp*

```
  qed }
hence eq:∀ i < length fs.
  ((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf) =
  ((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf')
  by(simp add:cspp-def)
{ fix i assume i < length fs
  hence (params (cspp (targetnode a) (HRB-slice S) fs)
    (fst cf))!i =
    ((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf)
    by(fastforce intro:params-nth)
  moreover
  from ‹i < length fs›
  have (params (cspp (targetnode a) (HRB-slice S) fs)
    (fst cf'))!i =
    ((cspp (targetnode a) (HRB-slice S) fs)!i)(fst cf')
    by(fastforce intro:params-nth)
  ultimately
  have (params (cspp (targetnode a) (HRB-slice S) fs)
    (fst cf))!i =
    (params (cspp (targetnode a) (HRB-slice S) fs)(fst cf'))!i
    using eq ‹i < length fs› by simp }
hence params (cspp (targetnode a) (HRB-slice S) fs)(fst cf) =
  params (cspp (targetnode a) (HRB-slice S) fs)(fst cf')
  by(simp add:list-eq-iff-nth-eq)
with slice-kind ‹(p,ins,outs) ∈ set procs›
obtain cfx where [simp]:
  transfer (slice-kind S a) (cf#cfs) = cfx#cf#cfs
  transfer (slice-kind S a) (cf'#cfs') = cfx#cf'#cfs'
  by auto
hence rv:∀ V∈rv S (CFG-node (targetnode a)).
  state-val (transfer (slice-kind S a) s) V =
  state-val (transfer (slice-kind S a) s') V by simp
from rvs ‹∀ V∈rv S (CFG-node m). state-val s V = state-val s' V›
  ‹sourcenode a = m›
have rvs':∀ i<length (a # cs).
  ∀ V∈rv S (CFG-node (sourcenode ((a # cs) ! i))).
  fst ((transfer (slice-kind S a) s) ! Suc i) V =
  fst ((transfer (slice-kind S a) s') ! Suc i) V
  by auto(case-tac i,auto)
from ‹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'› ‹as' = ax#asx›
have preds (slice-kinds S asx)
  (transfer (slice-kind S a) s') by(simp add:slice-kinds-def)
moreover
from ‹length s = Suc (length cs)›
have length (transfer (slice-kind S a) s) =
```

```
      Suc (length (a # cs)) by simp
    moreover
    from ‹length s' = Suc (length cs)›
    have length (transfer (slice-kind S a) s') =
      Suc (length (a # cs)) by simp
    moreover
    from IH[OF ‹upd-cs (a # cs) as = []› ‹same-level-path-aux (a # cs) asx›
      ‹∀ c∈set (a # cs). valid-edge c› ‹targetnode a −as→* (-Low-)›
      ‹targetnode a −asx→* (-Low-)› rvs' snds rv calculation] ‹as' = ax#asx›
    show ?thesis by(simp add:slice-kinds-def)
  qed
 qed
next
 case (slpa-Return cs a as Q p f c' cs')
 note IH = ‹⋀m as' s s'. ⟦upd-cs cs' as = []; same-level-path-aux cs' as';
  ∀ c∈set cs'. valid-edge c; m −as→* (-Low-); m −as'→* (-Low-);
  ∀ i<length cs'. ∀ V∈rv S (CFG-node (sourcenode (cs' ! i))).
  fst (s ! Suc i) V = fst (s' ! Suc i) V;
  ∀ i<Suc (length cs'). snd (s ! i) = snd (s' ! i);
  ∀ V∈rv S (CFG-node m). state-val s V = state-val s' V;
  preds (slice-kinds S as) s; preds (slice-kinds S as') s';
  length s = Suc (length cs'); length s' = Suc (length cs')⟧
  ⟹ ∀ V∈Use (-Low-). state-val (transfers(slice-kinds S as) s) V =
                     state-val (transfers(slice-kinds S as') s') V›
 note rvs = ‹ ∀ i<length cs. ∀ V∈rv S (CFG-node (sourcenode (cs ! i))).
  fst (s ! Suc i) V = fst (s' ! Suc i) V›
 from ‹m −a # as→* (-Low-)› have sourcenode a = m and valid-edge a
  and targetnode a −as→* (-Low-) by(auto elim:path-split-Cons)
 from ‹∀ c∈set cs. valid-edge c› ‹cs = c' # cs'›
 have valid-edge c' and ∀ c∈set cs'. valid-edge c by simp-all
 show ?case
 proof(cases L = {})
  case True with UseLow show ?thesis by simp
 next
  case False
  show ?thesis
  proof(cases as')
   case Nil
   with ‹m −as'→* (-Low-)› have m = (-Low-) by fastforce
   with ‹valid-edge a› ‹sourcenode a = m› have targetnode a = (-Exit-)
    by −(rule Exit-successor-of-Low,simp+)
   from Low-source-Exit-edge obtain a' where valid-edge a'
    and sourcenode a' = (-Low-) and targetnode a' = (-Exit-)
    and kind a' = (λs. True)√ by blast
   from ‹valid-edge a› ‹sourcenode a = m› ‹m = (-Low-)›
    ‹targetnode a = (-Exit-)› ‹valid-edge a'› ‹sourcenode a' = (-Low-)›
    ‹targetnode a' = (-Exit-)›
   have a = a' by(fastforce dest:edge-det)
   with ‹kind a' = (λs. True)√› have kind a = (λs. True)√ by simp
```

with ‹*targetnode a = (-Exit-)*› ‹*targetnode a −as→∗ (-Low-)*›
have *(-Low-) = (-Exit-)* **by** −(*drule path-Exit-source*,*auto*)
with *False* **have** *False* **by** −(*drule Low-neq-Exit*,*simp*)
thus *?thesis* **by** *simp*
next
case (*Cons ax asx*)
with ‹*m −as′→∗ (-Low-)*› **have** *sourcenode ax = m* **and** *valid-edge ax*
and *targetnode ax −asx→∗ (-Low-)* **by**(*auto elim*:*path-split-Cons*)
from ‹*valid-edge a*› ‹*valid-edge ax*› ‹*kind a = Q↩pf*›
‹*sourcenode a = m*› ‹*sourcenode ax = m*›
have ∃ *Q f*. *kind ax = Q↩pf* **by**(*auto dest*:*return-edges-only*)
with ‹*same-level-path-aux cs as′*› ‹*as′ = ax#asx*› ‹*cs = c′ # cs′*›
have *ax ∈ get-return-edges c′* **and** *same-level-path-aux cs′ asx* **by** *auto*
from ‹*valid-edge c′*› ‹*ax ∈ get-return-edges c′*› ‹*a ∈ get-return-edges c′*›
have [*simp*]:*ax = a* **by**(*rule get-return-edges-unique*)
from ‹*targetnode ax −asx→∗ (-Low-)*› **have** *targetnode a −asx→∗ (-Low-)*
by *simp*
from ‹*upd-cs cs (a # as) = []*› ‹*kind a = Q↩pf*› ‹*cs = c′ # cs′*›
‹*a ∈ get-return-edges c′*›
have *upd-cs cs′ as = []* **by** *simp*
from ‹*length s = Suc (length cs)*› ‹*cs = c′ # cs′*›
obtain *cf cfx cfs* **where** *s = cf#cfx#cfs*
by(*cases s*,*auto*,*case-tac list*,*fastforce+*)
from ‹*length s′ = Suc (length cs)*› ‹*cs = c′ # cs′*›
obtain *cf′ cfx′ cfs′* **where** *s′ = cf′#cfx′#cfs′*
by(*cases s′*,*auto*,*case-tac list*,*fastforce+*)
from *rvs* ‹*cs = c′ # cs′*› ‹*s = cf#cfx#cfs*› ‹*s′ = cf′#cfx′#cfs′*›
have *rvs1*:∀ *i*<*length cs′*.
∀ *V*∈*rv S (CFG-node (sourcenode (cs′ ! i*))).
fst ((cfx#cfs) ! Suc i) V = fst ((cfx′#cfs′) ! Suc i) V
and ∀ *V*∈*rv S (CFG-node (sourcenode c′*)).
(fst cfx) V = (fst cfx′) V
by *auto*
from ‹*valid-edge c′*› ‹*a ∈ get-return-edges c′*›
obtain *Qx rx px fsx* **where** *kind c′ = Qx*:*rx↪pxfsx*
by(*fastforce dest*!:*only-call-get-return-edges*)
have ∀ *V ∈ rv S (CFG-node (targetnode a*)).
V ∈ rv S (CFG-node (sourcenode c′))
proof
fix *V* **assume** *V ∈ rv S (CFG-node (targetnode a*))
from ‹*valid-edge c′*› ‹*a ∈ get-return-edges c′*›
obtain *a′* **where** *edge*:*valid-edge a′ sourcenode a′ = sourcenode c′*
targetnode a′ = targetnode a intra-kind (kind a′)
by −(*drule call-return-node-edge*,*auto simp*:*intra-kind-def*)
from ‹*V ∈ rv S (CFG-node (targetnode a*))›
obtain *as n′* **where** *targetnode a −as→ι∗ parent-node n′*
and *n′ ∈ HRB-slice S* **and** *V ∈ Use$_{SDG}$ n′*
and *all*:∀ *n″*. *valid-SDG-node n″ ∧ parent-node n″ ∈ set (sourcenodes as)*
⟶ V ∉ Def$_{SDG}$ n″ **by**(*fastforce elim*:*rvE*)

from ‹*targetnode a* $-as\rightarrow_\iota*$ *parent-node n'*› *edge*
have *sourcenode c'* $-a'\#as\rightarrow_\iota*$ *parent-node n'*
by(*fastforce intro*:*Cons-path simp*:*intra-path-def*)
from ‹*valid-edge c'*› ‹*kind c'* = *Qx*:*rx*$\hookrightarrow_{px}$*fsx*› **have** *Def* (*sourcenode c'*) = {}
by(*rule call-source-Def-empty*)
hence ∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* = *sourcenode c'*
⟶ *V* ∉ *Def*$_{SDG}$ *n''* **by**(*fastforce dest*:*SDG-Def-parent-Def*)
with *all* ‹*sourcenode a'* = *sourcenode c'*›
have ∀ *n''*. *valid-SDG-node n''* ∧ *parent-node n''* ∈ *set* (*sourcenodes* (*a'*#*as*))
⟶ *V* ∉ *Def*$_{SDG}$ *n''* **by**(*fastforce simp*:*sourcenodes-def*)
with ‹*sourcenode c'* $-a'\#as\rightarrow_\iota*$ *parent-node n'*›
‹*n'* ∈ *HRB-slice S*› ‹*V* ∈ *Use*$_{SDG}$ *n'*›
show *V* ∈ *rv S* (*CFG-node* (*sourcenode c'*))
by(*fastforce intro*:*rvI*)
qed
show *?thesis*
proof(*cases sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$)
case *True*
from ‹*valid-edge c'*› ‹*a* ∈ *get-return-edges c'*›
have *get-proc* (*targetnode c'*) = *get-proc* (*sourcenode a*)
by −(*drule intra-proc-additional-edge*,
auto dest:*get-proc-intra simp*:*intra-kind-def*)
moreover
from ‹*valid-edge c'*› ‹*kind c'* = *Qx*:*rx*$\hookrightarrow_{px}$*fsx*›
have *get-proc* (*targetnode c'*) = *px* **by**(*rule get-proc-call*)
moreover
from ‹*valid-edge a*› ‹*kind a* = *Q*$\hookleftarrow_p$*f*›
have *get-proc* (*sourcenode a*) = *p* **by**(*rule get-proc-return*)
ultimately have [*simp*]:*px* = *p* **by** *simp*
from ‹*valid-edge c'*› ‹*kind c'* = *Qx*:*rx*$\hookrightarrow_{px}$*fsx*›
obtain *ins outs* **where** (*p*,*ins*,*outs*) ∈ *set procs*
by(*fastforce dest*!:*callee-in-procs*)
with ‹*sourcenode a* ∈ ⌊*HRB-slice S*⌋$_{CFG}$›
‹*valid-edge a*› ‹*kind a* = *Q*$\hookleftarrow_p$*f*›
have *slice-kind*:*slice-kind S a* =
Q$\hookleftarrow_p$(λ*cf cf'*. *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 *rvs1* **have** *rvs'*:∀ *i*<*length cs'*.
∀ *V*∈*rv S* (*CFG-node* (*sourcenode* (*cs'* ! *i*))).

```
    fst ((transfer (slice-kind S a) s) ! Suc i) V =
    fst ((transfer (slice-kind S a) s') ! Suc i) V
    by fastforce
  from slice-kind ‹∀ i<Suc (length cs). snd (s ! i) = snd (s' ! i)› ‹cs = c' #
cs'›
    ‹s = cf#cfx#cfs› ‹s' = cf'#cfx'#cfs'›
  have snds:∀ i<Suc (length cs').
    snd (transfer (slice-kind S a) s ! i) =
    snd (transfer (slice-kind S a) s' ! i)
    apply auto apply(case-tac i) apply auto
    by(erule-tac x=Suc (Suc nat) in allE) auto
  have ∀ V∈rv S (CFG-node (targetnode a)).
    (rspp (targetnode a) (HRB-slice S) outs
    (fst cfx) (fst cf)) V =
    (rspp (targetnode a) (HRB-slice S) outs
    (fst cfx') (fst cf')) V
  proof
    fix V assume V ∈ rv S (CFG-node (targetnode a))
    show (rspp (targetnode a) (HRB-slice S) outs
      (fst cfx) (fst cf)) V =
      (rspp (targetnode a) (HRB-slice S) outs
      (fst cfx') (fst cf')) V
    proof(cases V ∈ set (ParamDefs (targetnode a)))
      case True
      then obtain i where i < length (ParamDefs (targetnode a))
        and (ParamDefs (targetnode a))!i = V
        by(fastforce simp:in-set-conv-nth)
      from ‹valid-edge a› ‹kind a = Q↩pf› ‹(p,ins,outs) ∈ set procs›
      have length(ParamDefs (targetnode a)) = length outs
        by(fastforce intro:ParamDefs-return-target-length)
      show ?thesis
      proof(cases Actual-out(targetnode a,i) ∈ HRB-slice S)
        case True
        with ‹i < length (ParamDefs (targetnode a))› ‹valid-edge a›
          ‹length(ParamDefs (targetnode a)) = length outs›
          ‹(ParamDefs (targetnode a))!i = V›[THEN sym]
        have rspp-eq:(rspp (targetnode a)
          (HRB-slice S) outs (fst cfx) (fst cf)) V =
          (fst cf)(outs!i)
          (rspp (targetnode a)
          (HRB-slice S) outs (fst cfx') (fst cf')) V =
          (fst cf')(outs!i)
          by(auto intro:rspp-Actual-out-in-slice)
        from ‹valid-edge a› ‹kind a = Q↩pf› ‹(p,ins,outs) ∈ set procs›
    have ∀ V ∈ set outs. V ∈ Use (sourcenode a) by(fastforce dest:outs-in-Use)
      have ∀ V ∈ Use (sourcenode a). V ∈ rv S (CFG-node m)
      proof
        fix V assume V ∈ Use (sourcenode a)
        from ‹valid-edge a› ‹sourcenode a = m›
```

```
        have parent-node (CFG-node m) −[]→ι* parent-node (CFG-node m)
          by(fastforce intro:empty-path simp:intra-path-def)
        with ‹sourcenode a ∈ ⌊HRB-slice S⌋CFG›
          ‹V ∈ Use (sourcenode a)› ‹sourcenode a = m› ‹valid-edge a›
        show V ∈ rv S (CFG-node m)
          by −(rule rvI,
                  auto intro!:CFG-Use-SDG-Use simp:SDG-to-CFG-set-def
sourcenodes-def)
      qed
      with ‹∀ V ∈ set outs. V ∈ Use (sourcenode a)›
      have ∀ V ∈ set outs. V ∈ rv S (CFG-node m) by simp
      with ‹∀ V∈rv S (CFG-node m). state-val s V = state-val s' V›
        ‹s = cf#cfx#cfs› ‹s' = cf'#cfx'#cfs'›
      have ∀ V ∈ set outs. (fst cf) V = (fst cf') V by simp
      with ‹i < length (ParamDefs (targetnode a))›
        ‹length(ParamDefs (targetnode a)) = length outs›
      have (fst cf)(outs!i) = (fst cf')(outs!i) by fastforce
      with rspp-eq show ?thesis by simp
    next
      case False
      with ‹i < length (ParamDefs (targetnode a))› ‹valid-edge a›
        ‹length(ParamDefs (targetnode a)) = length outs›
        ‹(ParamDefs (targetnode a))!i = V›[THEN sym]
      have rspp-eq:(rspp (targetnode a)
        (HRB-slice S) outs (fst cfx) (fst cf)) V =
        (fst cfx)((ParamDefs (targetnode a))!i)
        (rspp (targetnode a)
        (HRB-slice S) outs (fst cfx') (fst cf')) V =
        (fst cfx')((ParamDefs (targetnode a))!i)
        by(auto intro:rspp-Actual-out-notin-slice)
      from ‹∀ V∈rv S (CFG-node (sourcenode c')).
        (fst cfx) V = (fst cfx') V›
        ‹V ∈ rv S (CFG-node (targetnode a))›
        ‹∀ V ∈ rv S (CFG-node (targetnode a)).
        V ∈ rv S (CFG-node (sourcenode c'))›
        ‹(ParamDefs (targetnode a))!i = V›[THEN sym]
      have (fst cfx) (ParamDefs (targetnode a) ! i) =
        (fst cfx') (ParamDefs (targetnode a) ! i) by fastforce
      with rspp-eq show ?thesis by fastforce
    qed
  next
    case False
    with ‹∀ V∈rv S (CFG-node (sourcenode c')).
      (fst cfx) V = (fst cfx') V›
      ‹V ∈ rv S (CFG-node (targetnode a))›
      ‹∀ V ∈ rv S (CFG-node (targetnode a)).
      V ∈ rv S (CFG-node (sourcenode c'))›
    show ?thesis by(fastforce simp:rspp-def map-merge-def)
  qed
```

qed
with *sx sx′*
have *rv′*:∀ *V*∈*rv S* (*CFG-node* (*targetnode a*)).
state-val (*transfer* (*slice-kind S a*) *s*) *V* =
state-val (*transfer* (*slice-kind S a*) *s′*) *V*
by *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′*› ‹*as′* = *ax*#*asx*›
have *preds* (*slice-kinds S asx*)
(*transfer* (*slice-kind S a*) *s′*)
by(*simp add*:*slice-kinds-def*)
moreover
from ‹*length s* = *Suc* (*length cs*)› ‹*cs* = *c′* # *cs′*› *sx*
have *length* (*transfer* (*slice-kind S a*) *s*) = *Suc* (*length cs′*)
by(*simp*,*simp add*:‹*s* = *cf*#*cfx*#*cfs*›)
moreover
from ‹*length s′* = *Suc* (*length cs*)› ‹*cs* = *c′* # *cs′*› *sx′*
have *length* (*transfer* (*slice-kind S a*) *s′*) = *Suc* (*length cs′*)
by(*simp*,*simp add*:‹*s′* = *cf′*#*cfx′*#*cfs′*›)
moreover
from *IH*[*OF* ‹*upd-cs cs′ as* = []› ‹*same-level-path-aux cs′ asx*›
‹∀ *c*∈*set cs′*. *valid-edge c*› ‹*targetnode a* −*as*→∗ (*-Low-*)›
‹*targetnode a* −*asx*→∗ (*-Low-*)› *rvs′ snds rv′ calculation*] ‹*as′* = *ax*#*asx*›
show *?thesis* **by**(*simp add*:*slice-kinds-def*)
next
case *False*
from *this* ‹*kind a* = *Q*↪*pf*›
have *slice-kind*:*slice-kind S a* = (λ*cf*. *True*)↪*p*(λ*cf cf′*. *cf′*)
by(*rule slice-kind-Return*)
with ‹*s* = *cf*#*cfx*#*cfs*› ‹*s′* = *cf′*#*cfx′*#*cfs′*›
have [*simp*]:*transfer* (*slice-kind S a*) *s* = *cfx*#*cfs*
transfer (*slice-kind S a*) *s′* = *cfx′*#*cfs′* **by** *simp-all*
from *slice-kind* ‹∀ *i*<*Suc* (*length cs*). *snd* (*s* ! *i*) = *snd* (*s′* ! *i*)›
‹*cs* = *c′* # *cs′*› ‹*s* = *cf*#*cfx*#*cfs*› ‹*s′* = *cf′*#*cfx′*#*cfs′*›
have *snds*:∀ *i*<*Suc* (*length cs′*).
snd (*transfer* (*slice-kind S a*) *s* ! *i*) =
snd (*transfer* (*slice-kind S a*) *s′* ! *i*) **by** *fastforce*
from *rvs1* **have** *rvs′*:∀ *i*<*length cs′*.
∀ *V*∈*rv S* (*CFG-node* (*sourcenode* (*cs′* ! *i*))).
fst ((*transfer* (*slice-kind S a*) *s*) ! *Suc i*) *V* =
fst ((*transfer* (*slice-kind S a*) *s′*) ! *Suc i*) *V*
by *fastforce*
from ‹∀ *V* ∈ *rv S* (*CFG-node* (*targetnode a*)).
V ∈ *rv S* (*CFG-node* (*sourcenode c′*))›
‹∀ *V*∈*rv S* (*CFG-node* (*sourcenode c′*)).

```
      (fst cfx) V = (fst cfx') V›
    have rv':∀ V∈rv S (CFG-node (targetnode a)).
      state-val (transfer (slice-kind S a) s) V =
      state-val (transfer (slice-kind S a) s') V by simp
    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'› ‹as' = ax#asx›
    have preds (slice-kinds S asx)
      (transfer (slice-kind S a) s')
      by(simp add:slice-kinds-def)
    moreover
    from ‹length s = Suc (length cs)› ‹cs = c' # cs'›
    have length (transfer (slice-kind S a) s) = Suc (length cs')
      by(simp,simp add:‹s = cf#cfx#cfs›)
    moreover
    from ‹length s' = Suc (length cs)› ‹cs = c' # cs'›
    have length (transfer (slice-kind S a) s') = Suc (length cs')
      by(simp,simp add:‹s' = cf'#cfx'#cfs'›)
    moreover
    from IH[OF ‹upd-cs cs' as = []› ‹same-level-path-aux cs' asx›
      ‹∀ c∈set cs'. valid-edge c› ‹targetnode a −as→∗ (-Low-)›
      ‹targetnode a −asx→∗ (-Low-)› rvs' snds rv' calculation] ‹as' = ax#asx›
    show ?thesis by(simp add:slice-kinds-def)
  qed
 qed
 qed
qed


lemma rv-Low-Use-Low:
 assumes m −as→√∗ (-Low-) and m −as'→√∗ (-Low-) and get-proc m = Main
 and ∀ V ∈ rv S (CFG-node m). cf V = cf' V
 and preds (slice-kinds S as) [(cf,undefined)]
 and preds (slice-kinds S as') [(cf',undefined)]
 and CFG-node (-Low-) ∈ S
 shows ∀ V ∈ Use (-Low-).
   state-val (transfers(slice-kinds S as) [(cf,undefined)]) V =
   state-val (transfers(slice-kinds S as') [(cf',undefined)]) V
proof(cases as)
 case Nil
 with ‹m −as→√∗ (-Low-)› have valid-node m and m = (-Low-)
  by(auto intro:path-valid-node simp:vp-def)
 { fix V assume V ∈ Use (-Low-)
  moreover
  from ‹valid-node m› ‹m = (-Low-)› have (-Low-) −[]→ι∗ (-Low-)
   by(fastforce intro:empty-path simp:intra-path-def)
```

```
    moreover
    from ‹valid-node m› ‹m = (-Low-)› ‹CFG-node (-Low-) ∈ S›
    have CFG-node (-Low-) ∈ HRB-slice S
      by(fastforce intro:HRB-slice-refl)
    ultimately have V ∈ rv S (CFG-node m) using ‹m = (-Low-)›
      by(auto intro!:rvI CFG-Use-SDG-Use simp:sourcenodes-def) }
  hence ∀ V ∈ Use (-Low-). V ∈ rv S (CFG-node m) by simp
  show ?thesis
  proof(cases L = {})
    case True with UseLow show ?thesis by simp
  next
    case False
    from ‹m −as'→√* (-Low-)› have m −as'→* (-Low-) by(simp add:vp-def)
    from ‹m −as'→* (-Low-)› ‹m = (-Low-)› have as' = []
    proof(induct m as' m'≡(-Low-) rule:path.induct)
      case (Cons-path m'' as a m)
      from ‹valid-edge a› ‹sourcenode a = m› ‹m = (-Low-)›
      have targetnode a = (-Exit-) by −(rule Exit-successor-of-Low,simp+)
      with ‹targetnode a = m''› ‹m'' −as→* (-Low-)›
      have (-Low-) = (-Exit-) by −(drule path-Exit-source,auto)
      with False have False by −(drule Low-neq-Exit,simp)
      thus ?case by simp
    qed simp
    with Nil ‹∀ V ∈ rv S (CFG-node m). cf V = cf' V›
      ‹∀ V ∈ Use (-Low-). V ∈ rv S (CFG-node m)›
    show ?thesis by(fastforce simp:slice-kinds-def)
  qed
next
  case (Cons ax asx)
  with ‹m −as→√* (-Low-)› have sourcenode ax = m and valid-edge ax
    and targetnode ax −asx→* (-Low-)
    by(auto elim:path-split-Cons simp:vp-def)
  show ?thesis
  proof(cases L = {})
    case True with UseLow show ?thesis by simp
  next
    case False
    show ?thesis
    proof(cases as')
      case Nil
      with ‹m −as'→√* (-Low-)› have m = (-Low-) by(fastforce simp:vp-def)
      with ‹valid-edge ax› ‹sourcenode ax = m› have targetnode ax = (-Exit-)
        by −(rule Exit-successor-of-Low,simp+)
      from Low-source-Exit-edge obtain a' where valid-edge a'
        and sourcenode a' = (-Low-) and targetnode a' = (-Exit-)
        and kind a' = (λs. True)√ by blast
      from ‹valid-edge ax› ‹sourcenode ax = m› ‹m = (-Low-)›
        ‹targetnode ax = (-Exit-)› ‹valid-edge a'› ‹sourcenode a' = (-Low-)›
        ‹targetnode a' = (-Exit-)›
```

```
    have ax = a' by(fastforce dest:edge-det)
    with ‹kind a' = (λs. True)√› have kind ax = (λs. True)√ by simp
    with ‹targetnode ax = (-Exit-)› ‹targetnode ax −asx→∗ (-Low-)›
    have (-Low-) = (-Exit-) by −(drule path-Exit-source,auto)
    with False have False by −(drule Low-neq-Exit,simp)
    thus ?thesis by simp
  next
    case (Cons ax' asx')
   from ‹m −as→√∗ (-Low-)› have valid-path-aux [] as and m −as→∗ (-Low-)
      by(simp-all add:vp-def valid-path-def)
    from this ‹as = ax#asx› ‹get-proc m = Main›
    have same-level-path-aux [] as ∧ upd-cs [] as = []
      by −(rule vpa-Main-slpa[of - - m (-Low-)],
      (fastforce intro!:get-proc-Low simp:valid-call-list-def)+)
    hence same-level-path-aux [] as and upd-cs [] as = [] by simp-all
   from ‹m −as'→√∗ (-Low-)› have valid-path-aux [] as' and m −as'→∗ (-Low-)
      by(simp-all add:vp-def valid-path-def)
    from this ‹as' = ax'#asx'› ‹get-proc m = Main›
    have same-level-path-aux [] as' ∧ upd-cs [] as' = []
      by −(rule vpa-Main-slpa[of - - m (-Low-)],
      (fastforce intro!:get-proc-Low simp:valid-call-list-def)+)
    hence same-level-path-aux [] as' by simp
    from ‹same-level-path-aux [] as› ‹upd-cs [] as = []›
      ‹same-level-path-aux [] as'› ‹m −as→∗ (-Low-)› ‹m −as'→∗ (-Low-)›
      ‹∀ V ∈ rv S (CFG-node m). cf V = cf' V› ‹CFG-node (-Low-) ∈ S›
      ‹preds (slice-kinds S as) [(cf,undefined)]›
      ‹preds (slice-kinds S as') [(cf',undefined)]›
    show ?thesis by −(erule slpa-rv-Low-Use-Low,auto)
  qed
 qed
qed



lemma nonInterference-path-to-Low:
 assumes [cf] ≈L [cf'] and (-High-) ∉ ⌊HRB-slice S⌋CFG
 and CFG-node (-Low-) ∈ S
 and (-Entry-) −as→√∗ (-Low-) and preds (kinds as) [(cf,undefined)]
 and (-Entry-) −as'→√∗ (-Low-) and preds (kinds as') [(cf',undefined)]
 shows map fst (transfers (kinds as) [(cf,undefined)]) ≈L
       map fst (transfers (kinds as') [(cf',undefined)])
proof −
 from ‹(-Entry-) −as→√∗ (-Low-)› ‹preds (kinds as) [(cf,undefined)]›
   ‹CFG-node (-Low-) ∈ S›
 obtain asx where preds (slice-kinds S asx) [(cf,undefined)]
   and ∀ V ∈ Use (-Low-).
   state-val (transfers (slice-kinds S asx) [(cf,undefined)]) V =
   state-val (transfers (kinds as) [(cf,undefined)]) V
   and slice-edges S [] as = slice-edges S [] asx
```

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

theorem *nonInterference-path*:
assumes ⌊*cf*⌋ ≈$_L$ ⌊*cf′*⌋ **and** (*-High-*) ∉ ⌊*HRB-slice S*⌋$_{CFG}$
and *CFG-node* (*-Low-*) ∈ *S*
and (*-Entry-*) −*as*→$_{\surd}$* (*-Exit-*) **and** *preds* (*kinds as*) [(*cf*,*undefined*)]
and (*-Entry-*) −*as′*→$_{\surd}$* (*-Exit-*) **and** *preds* (*kinds as′*) [(*cf′*,*undefined*)]
shows *map fst* (*transfers* (*kinds as*) [(*cf*,*undefined*)]) ≈$_L$
map fst (*transfers* (*kinds as′*) [(*cf′*,*undefined*)])
proof −
from ‹(*-Entry-*) −*as*→$_{\surd}$* (*-Exit-*)› **obtain** *x xs* **where** *as* = *x*#*xs*
and (*-Entry-*) = *sourcenode x* **and** *valid-edge x*
and *targetnode x* −*xs*→* (*-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*→$_{\surd}$* (*-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* = (λ*s*. *False*)$_{\surd}$ **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* = (λ*s*. *False*)$_{\surd}$›
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*′ = (λ*s*. *True*)$_{\surd}$
by(*fastforce elim*:*Exit-path-Low-path*)
with ‹(*-Entry-*) = *sourcenode x*› ‹*valid-edge x*›
have (*-Entry-*) −*x*#*xs*′→* (*-Low-*) **by**(*fastforce intro*:*Cons-path*)
from ‹*valid-path as*› ‹*as* = *x*#*xs*› ‹*xs* = *xs*′@[*x*′]›
have *valid-path* (*x*#*xs*′)
by(*simp add*:*valid-path-def del*:*valid-path-aux.simps*)
(*rule valid-path-aux-split*,*simp*)
with ‹(*-Entry-*) −*x*#*xs*′→* (*-Low-*)› **have** (*-Entry-*) −*x*#*xs*′→$_{\surd}$* (*-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*′→$_{\surd}$* (*-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*′→$_{\surd}$* (*-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 ‹valid-edge y› have False by(rule Entry-target)
  thus ?thesis by simp
next
  case Exit
  with ‹targetnode y −ys→∗ (-Exit-)› have ys = []
    by −(drule path-Exit-source,auto)
  from Entry-Exit-edge obtain z where valid-edge z
    and sourcenode z = (-Entry-) and targetnode z = (-Exit-)
    and kind z = (λs. False)√ by blast
  from ‹valid-edge y› ‹valid-edge z› ‹(-Entry-) = sourcenode y›
    ‹sourcenode z = (-Entry-)› Exit ‹targetnode z = (-Exit-)›
  have y = z by(fastforce intro:edge-det)
  with ‹preds (kinds as′) [(cf′,undefined)]› ‹as′ = y#ys› ‹ys = []›
    ‹kind z = (λs. False)√›
  have False by(simp add:kinds-def)
  thus ?thesis by simp
qed simp
with ‹targetnode y −ys→∗ (-Exit-)› obtain y′ ys′ where ys = ys′@[y′]
  and targetnode y −ys′→∗ (-Low-) and kind y′ = (λs. True)√
  by(fastforce elim:Exit-path-Low-path)
with ‹(-Entry-) = sourcenode y› ‹valid-edge y›
have (-Entry-) −y#ys′→∗ (-Low-) by(fastforce intro:Cons-path)
from ‹valid-path as′› ‹as′ = y#ys› ‹ys = ys′@[y′]›
have valid-path (y#ys′)
  by(simp add:valid-path-def del:valid-path-aux.simps)
    (rule valid-path-aux-split,simp)
with ‹(-Entry-) −y#ys′→∗ (-Low-)› have (-Entry-) −y#ys′→√∗ (-Low-)
  by(simp add:vp-def)
from ‹as′ = y#ys› ‹ys = ys′@[y′]› have as′ = (y#ys′)@[y′] by simp
with ‹preds (kinds as′) [(cf′,undefined)]›
have preds (kinds (y#ys′)) [(cf′,undefined)]
  by(simp add:kinds-def preds-split)
from ‹[cf] ≈L [cf′]› ‹(-High-) ∉ ⌊HRB-slice S⌋CFG› ‹CFG-node (-Low-) ∈ S›
  ‹(-Entry-) −x#xs′→√∗ (-Low-)› ‹preds (kinds (x#xs′)) [(cf,undefined)]›
  ‹(-Entry-) −y#ys′→√∗ (-Low-)› ‹preds (kinds (y#ys′)) [(cf′,undefined)]›
have map fst (transfers (kinds (x#xs′)) [(cf,undefined)]) ≈L
  map fst (transfers (kinds (y#ys′)) [(cf′,undefined)])
  by(rule nonInterference-path-to-Low)
with ‹as = x#xs› ‹xs = xs′@[x′]› ‹kind x′ = (λs. True)√›
  ‹as′ = y#ys› ‹ys = ys′@[y′]› ‹kind y′ = (λs. True)√›
show ?thesis
  apply(cases transfers (map kind xs′) (transfer (kind x) [(cf,undefined)]))
  apply (auto simp add:kinds-def transfers-split)
  by((cases transfers (map kind ys′) (transfer (kind y) [(cf′,undefined)])),
     (auto simp add:kinds-def transfers-split))+
qed


end
```

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

locale *NonInterferenceInter* =
  NonInterferenceInterGraph sourcenode targetnode kind valid-edge Entry
    get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
    H L High Low +
  SemanticsProperty sourcenode targetnode kind valid-edge Entry get-proc
    get-return-edges procs Main Exit Def Use ParamDefs ParamUses sem identifies
  for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
  and *kind* :: *'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*
  and *valid-edge* :: *'edge* ⇒ *bool*
  and *Entry* :: *'node* (‹*'*(*'-Entry'-*)*'*›) **and** *get-proc* :: *'node* ⇒ *'pname*
  and *get-return-edges* :: *'edge* ⇒ *'edge set*
  and *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname*
  and *Exit*::*'node* (‹*'*(*'-Exit'-*)*'*›)
  and *Def* :: *'node* ⇒ *'var set* **and** *Use* :: *'node* ⇒ *'var set*
  and *ParamDefs* :: *'node* ⇒ *'var list* **and** *ParamUses* :: *'node* ⇒ *'var set list*
  and *sem* :: *'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *bool*
    (‹((*1*⟨*-*,*/-*⟩) ⇒*/* (*1*⟨*-*,*/-*⟩))› [*0*,*0*,*0*,*0*] *81*)
  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* ∧ *sourcenode a* = *n* ∧ *targetnode a* = (*-Low-*) ∧ *kind a* = ⇑*id*
begin

The following theorem needs the explicit edge from (*-High-*) to *n*. An approach using a *init* predicate for initial statements, being reachable from (*-High-*) via a $(\lambda s.\ True)_{\surd}$ 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 inital statement: the initial node and the node within the loop (because of loop unrolling).

theorem *nonInterference*:
  assumes $[cf_1] \approx_L [cf_2]$ **and** (*-High-*) ∉ ⌊*HRB-slice S*⌋$_{CFG}$
  and *CFG-node* (*-Low-*) ∈ *S*
  and *valid-edge a* **and** *sourcenode a* = (*-High-*) **and** *targetnode a* = *n*
  and *kind a* = $(\lambda s.\ True)_{\surd}$ **and** *n* ≙ *c* **and** *final c'*
  and $\langle c,[cf_1]\rangle \Rightarrow \langle c',s_1\rangle$ **and** $\langle c,[cf_2]\rangle \Rightarrow \langle c',s_2\rangle$
  shows $s_1 \approx_L s_2$
proof −
  from *High-target-Entry-edge* **obtain** *ax* **where** *valid-edge ax*
    and *sourcenode ax* = (*-Entry-*) **and** *targetnode ax* = (*-High-*)
    and *kind ax* = $(\lambda s.\ True)_{\surd}$ **by** *blast*

from ‹$n \triangleq c$› ‹$\langle c,[cf_1]\rangle \Rightarrow \langle c',s_1\rangle$›
obtain n_1 as_1 cfs_1 **where** $n -as_1\rightarrow_{\surd}* n_1$ **and** $n_1 \triangleq c'$
and *preds* (*kinds* as_1) [(cf_1,*undefined*)]
and *transfers* (*kinds* as_1) [(cf_1,*undefined*)] = cfs_1 **and** *map fst* $cfs_1 = s_1$
by(*fastforce dest*:*fundamental-property*)
from ‹$n -as_1\rightarrow_{\surd}* n_1$› ‹*valid-edge a*› ‹*sourcenode a* = (*-High-*)› ‹*targetnode a* = n›
‹*kind a* = $(\lambda s.\ True)_{\surd}$›
have (*-High-*) $-a\#as_1\rightarrow_{\surd}* n_1$ **by**(*fastforce intro*:*Cons-path simp*:*vp-def valid-path-def*)
from ‹*final c'*› ‹$n_1 \triangleq c'$›
obtain a_1 **where** *valid-edge* a_1 **and** *sourcenode* $a_1 = n_1$
and *targetnode* a_1 = (*-Low-*) **and** *kind* $a_1 = \Uparrow id$ **by**(*fastforce dest*:*final-edge-Low*)
hence $n_1 -[a_1]\rightarrow*$ (*-Low-*) **by**(*fastforce intro*:*path-edge*)
with ‹(*-High-*) $-a\#as_1\rightarrow_{\surd}* n_1$› **have** (*-High-*) $-(a\#as_1)@[a_1]\rightarrow*$ (*-Low-*)
by(*fastforce intro*!:*path-Append simp*:*vp-def*)
with ‹*valid-edge ax*› ‹*sourcenode ax* = (*-Entry-*)› ‹*targetnode ax* = (*-High-*)›
have (*-Entry-*) $-ax\#((a\#as_1)@[a_1])\rightarrow*$ (*-Low-*) **by** $-$(*rule Cons-path*)
moreover
from ‹(*-High-*) $-a\#as_1\rightarrow_{\surd}* n_1$› **have** *valid-path-aux* [] $(a\#as_1)$
by(*simp add*:*vp-def valid-path-def*)
with ‹*kind* $a_1 = \Uparrow id$› **have** *valid-path-aux* [] $((a\#as_1)@[a_1])$
by(*fastforce intro*:*valid-path-aux-Append*)
with ‹*kind ax* = $(\lambda s.\ True)_{\surd}$› **have** *valid-path-aux* [] $(ax\#((a\#as_1)@[a_1]))$
by *simp*
ultimately have (*-Entry-*) $-ax\#((a\#as_1)@[a_1])\rightarrow_{\surd}*$ (*-Low-*)
by(*simp add*:*vp-def valid-path-def*)
from ‹*valid-edge a*› ‹*kind a* = $(\lambda s.\ True)_{\surd}$› ‹*sourcenode a* = (*-High-*)›
‹*targetnode a* = n›
have *get-proc n* = *get-proc* (*-High-*)
by(*fastforce dest*:*get-proc-intra simp*:*intra-kind-def*)
with *get-proc-High* **have** *get-proc n* = *Main* **by** *simp*
from ‹*valid-edge* a_1› ‹*sourcenode* $a_1 = n_1$› ‹*targetnode* a_1 = (*-Low-*)› ‹*kind* $a_1 = \Uparrow id$›
have *get-proc* n_1 = *get-proc* (*-Low-*)
by(*fastforce dest*:*get-proc-intra simp*:*intra-kind-def*)
with *get-proc-Low* **have** *get-proc* n_1 = *Main* **by** *simp*
from ‹$n -as_1\rightarrow_{\surd}* n_1$› **have** $n -as_1\rightarrow_{sl}* n_1$
by(*cases* as_1)
(*auto dest*!:*vpa-Main-slpa intro*:‹*get-proc* n_1 = *Main*› ‹*get-proc n* = *Main*›
simp:*vp-def valid-path-def valid-call-list-def slp-def*
same-level-path-def simp del:*valid-path-aux.simps*)
then obtain *cfx r* **where** *cfx*:*transfers* (*map kind* as_1) [(cf_1,*undefined*)] = [(*cfx*,*r*)]
by(*fastforce elim*:*slp-callstack-length-equal simp*:*kinds-def*)
from ‹*kind ax* = $(\lambda s.\ True)_{\surd}$› ‹*kind a* = $(\lambda s.\ True)_{\surd}$›
‹*preds* (*kinds* as_1) [(cf_1,*undefined*)]› ‹*kind* $a_1 = \Uparrow id$› *cfx*
have *preds* (*kinds* $(ax\#((a\#as_1)@[a_1]))$) [(cf_1,*undefined*)]
by(*auto simp*:*kinds-def preds-split*)
from ‹$n \triangleq c$› ‹$\langle c,[cf_2]\rangle \Rightarrow \langle c',s_2\rangle$›

obtain n_2 as_2 cfs_2 **where** $n\ -as_2\rightarrow_{\surd}*\ n_2$ **and** $n_2 \triangleq c'$
and *preds* (*kinds* as_2) [(cf_2,*undefined*)]
and *transfers* (*kinds* as_2) [(cf_2,*undefined*)] = cfs_2 **and** *map fst* $cfs_2 = s_2$
by(*fastforce dest*:*fundamental-property*)
from ‹$n\ -as_2\rightarrow_{\surd}*\ n_2$› ‹*valid-edge a*› ‹*sourcenode a* = (*-High-*)› ‹*targetnode a* = *n*›
‹*kind a* = $(\lambda s.\ True)_{\surd}$›
have (*-High-*) $-a\#as_2\rightarrow_{\surd}*\ n_2$ **by**(*fastforce intro*:*Cons-path simp*:*vp-def valid-path-def*)
from ‹*final c'*› ‹$n_2 \triangleq c'$›
obtain a_2 **where** *valid-edge* a_2 **and** *sourcenode* $a_2 = n_2$
and *targetnode* a_2 = (*-Low-*) **and** *kind* $a_2 = \Uparrow id$ **by**(*fastforce dest*:*final-edge-Low*)
hence $n_2\ -[a_2]\rightarrow*$ (*-Low-*) **by**(*fastforce intro*:*path-edge*)
with ‹(*-High-*) $-a\#as_2\rightarrow_{\surd}*\ n_2$› **have** (*-High-*) $-(a\#as_2)@[a_2]\rightarrow*$ (*-Low-*)
by(*fastforce intro*!:*path-Append simp*:*vp-def*)
with ‹*valid-edge ax*› ‹*sourcenode ax* = (*-Entry-*)› ‹*targetnode ax* = (*-High-*)›
have (*-Entry-*) $-ax\#((a\#as_2)@[a_2])\rightarrow*$ (*-Low-*) **by** −(*rule Cons-path*)
moreover
from ‹(*-High-*) $-a\#as_2\rightarrow_{\surd}*\ n_2$› **have** *valid-path-aux* [] $(a\#as_2)$
by(*simp add*:*vp-def valid-path-def*)
with ‹*kind* $a_2 = \Uparrow id$› **have** *valid-path-aux* [] $((a\#as_2)@[a_2])$
by(*fastforce intro*:*valid-path-aux-Append*)
with ‹*kind ax* = $(\lambda s.\ True)_{\surd}$› **have** *valid-path-aux* [] $(ax\#((a\#as_2)@[a_2]))$
by *simp*
ultimately have (*-Entry-*) $-ax\#((a\#as_2)@[a_2])\rightarrow_{\surd}*$ (*-Low-*)
by(*simp add*:*vp-def valid-path-def*)
from ‹*valid-edge a*› ‹*kind a* = $(\lambda s.\ True)_{\surd}$› ‹*sourcenode a* = (*-High-*)›
‹*targetnode a* = *n*›
have *get-proc n* = *get-proc* (*-High-*)
by(*fastforce dest*:*get-proc-intra simp*:*intra-kind-def*)
with *get-proc-High* **have** *get-proc n* = *Main* **by** *simp*
from ‹*valid-edge* a_2› ‹*sourcenode* $a_2 = n_2$› ‹*targetnode* a_2 = (*-Low-*)› ‹*kind* $a_2 = \Uparrow id$›
have *get-proc* n_2 = *get-proc* (*-Low-*)
by(*fastforce dest*:*get-proc-intra simp*:*intra-kind-def*)
with *get-proc-Low* **have** *get-proc* n_2 = *Main* **by** *simp*
from ‹$n\ -as_2\rightarrow_{\surd}*\ n_2$› **have** $n\ -as_2\rightarrow_{sl}*\ n_2$
by(*cases* as_2)
(*auto dest*!:*vpa-Main-slpa intro*:‹*get-proc* n_2 = *Main*› ‹*get-proc n* = *Main*›
simp:*vp-def valid-path-def valid-call-list-def slp-def*
same-level-path-def simp del:*valid-path-aux.simps*)
then obtain *cfx' r'*
where *cfx'*:*transfers* (*map kind* as_2) [(cf_2,*undefined*)] = [(*cfx'*,*r'*)]
by(*fastforce elim*:*slp-callstack-length-equal simp*:*kinds-def*)
from ‹*kind ax* = $(\lambda s.\ True)_{\surd}$› ‹*kind a* = $(\lambda s.\ True)_{\surd}$›
‹*preds* (*kinds* as_2) [(cf_2,*undefined*)]› ‹*kind* $a_2 = \Uparrow id$› *cfx'*
have *preds* (*kinds* $(ax\#((a\#as_2)@[a_2]))$) [(cf_2,*undefined*)]
by(*auto simp*:*kinds-def preds-split*)
from ‹$[cf_1] \approx_L [cf_2]$› ‹(*-High-*) $\notin \lfloor$*HRB-slice S*$\rfloor_{CFG}$› ‹*CFG-node* (*-Low-*) $\in S$›
‹(*-Entry-*) $-ax\#((a\#as_1)@[a_1])\rightarrow_{\surd}*$ (*-Low-*)›

‹preds (kinds (ax#((a#as$_1$)@[a$_1$]))) [(cf$_1$,undefined)]›
‹(-Entry-) −ax#((a#as$_2$)@[a$_2$])→$_{\surd}$* (-Low-)›
‹preds (kinds (ax#((a#as$_2$)@[a$_2$]))) [(cf$_2$,undefined)]›
have *map fst (transfers (kinds (ax#((a#as$_1$)@[a$_1$]))) [(cf$_1$,undefined)]) ≈$_L$ map fst (transfers (kinds (ax#((a#as$_2$)@[a$_2$]))) [(cf$_2$,undefined)])*
by(*rule nonInterference-path-to-Low*)
with ‹*kind ax = (λs. True)*$_{\surd}$› ‹*kind a = (λs. True)*$_{\surd}$› ‹*kind a$_1$ = ⇑id*› ‹*kind a$_2$ = ⇑id*›
‹*transfers (kinds as$_1$) [(cf$_1$,undefined)] = cfs$_1$*› ‹*map fst cfs$_1$ = s$_1$*›
‹*transfers (kinds as$_2$) [(cf$_2$,undefined)] = cfs$_2$*› ‹*map fst cfs$_2$ = s$_2$*›
show *?thesis* **by**(*cases s$_1$*)(*cases s$_2$*,(*fastforce simp*:*kinds-def transfers-split*)+)+
qed

end

end

3 Framework Graph Lifting for Noninterference

theory *LiftingInter*
imports *NonInterferenceInter*
begin

In this section, we show how a valid CFG from the slicing framework in [8] can be lifted to 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 =*
'node LDCFG-node × (('var,'val,'ret,'pname) edge-kind) × 'node LDCFG-node

3.1.2 Lifting basic definitions using *'edge* and *'node*

inductive *lift-valid-edge* :: *('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒*
('edge ⇒ ('var,'val,'ret,'pname) edge-kind) ⇒ 'node ⇒ 'node ⇒
('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒
bool

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

where *lve-edge*:
⟦*valid-edge a*; *src a* ≠ *E* ∨ *trg a* ≠ *X*;
e = (*Node* (*src a*),*knd a*,*Node* (*trg a*))⟧
⟹ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Entry-edge*:
e = (*NewEntry*,(λ*s*. *True*)$_{\surd}$,*Node E*)
⟹ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Exit-edge*:
e = (*Node X*,(λ*s*. *True*)$_{\surd}$,*NewExit*)
⟹ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Entry-Exit-edge*:
e = (*NewEntry*,(λ*s*. *False*)$_{\surd}$,*NewExit*)
⟹ *lift-valid-edge valid-edge src trg knd E X e*

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

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

inductive-set *lift-get-return-edges* :: (*'edge* ⇒ *'edge set*) ⇒ (*'edge* ⇒ *bool*) ⇒
(*'edge* ⇒ *'node*) ⇒ (*'edge* ⇒ *'node*) ⇒ (*'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*)

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

3.1.3 Lifting the Def and Use sets

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

$'var\ set \Rightarrow 'var\ set \Rightarrow ('node\ LDCFG\text{-}node \times 'var)\ set$
for Def::$('node \Rightarrow 'var\ set)$ **and** E::$'node$ **and** X::$'node$
and H::$'var\ set$ **and** L::$'var\ set$

where *lift-Def-node*:
$V \in Def\ n \Longrightarrow (Node\ n, V) \in lift\text{-}Def\text{-}set\ Def\ E\ X\ H\ L$

| *lift-Def-High*:
$V \in H \Longrightarrow (Node\ E, V) \in lift\text{-}Def\text{-}set\ Def\ E\ X\ H\ L$

abbreviation *lift-Def* :: $('node \Rightarrow 'var\ set) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
$'var\ set \Rightarrow 'var\ set \Rightarrow 'node\ LDCFG\text{-}node \Rightarrow 'var\ set$
where $lift\text{-}Def\ Def\ E\ X\ H\ L\ n \equiv \{V.\ (n, V) \in lift\text{-}Def\text{-}set\ Def\ E\ X\ H\ L\}$

inductive-set *lift-Use-set* :: $('node \Rightarrow 'var\ set) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
$'var\ set \Rightarrow 'var\ set \Rightarrow ('node\ LDCFG\text{-}node \times 'var)\ set$
for Use::$'node \Rightarrow 'var\ set$ **and** E::$'node$ **and** X::$'node$
and H::$'var\ set$ **and** L::$'var\ set$

where
lift-Use-node:
$V \in Use\ n \Longrightarrow (Node\ n, V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L$

| *lift-Use-High*:
$V \in H \Longrightarrow (Node\ E, V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L$

| *lift-Use-Low*:
$V \in L \Longrightarrow (Node\ X, V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L$

abbreviation *lift-Use* :: $('node \Rightarrow 'var\ set) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
$'var\ set \Rightarrow 'var\ set \Rightarrow 'node\ LDCFG\text{-}node \Rightarrow 'var\ set$
where $lift\text{-}Use\ Use\ E\ X\ H\ L\ n \equiv \{V.\ (n, V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L\}$

fun *lift-ParamUses* :: $('node \Rightarrow 'var\ set\ list) \Rightarrow 'node\ LDCFG\text{-}node \Rightarrow 'var\ set\ list$
where $lift\text{-}ParamUses\ ParamUses\ (Node\ n) = ParamUses\ n$
| $lift\text{-}ParamUses\ ParamUses\ NewEntry = []$
| $lift\text{-}ParamUses\ ParamUses\ NewExit = []$

fun *lift-ParamDefs* :: $('node \Rightarrow 'var\ list) \Rightarrow 'node\ LDCFG\text{-}node \Rightarrow 'var\ list$
where $lift\text{-}ParamDefs\ ParamDefs\ (Node\ n) = ParamDefs\ n$
| $lift\text{-}ParamDefs\ ParamDefs\ NewEntry = []$
| $lift\text{-}ParamDefs\ ParamDefs\ NewExit = []$

3.2 The lifting lemmas

3.2.1 Lifting the CFG locales

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

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

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

lemma *lift-CFG*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses*
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit*
shows *CFG src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *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 *trg a* = *NewEntry*
thus *False* **by**(*fastforce elim*:*lift-valid-edge.cases*)
next
show *lift-get-proc get-proc Main NewEntry* = *Main* **by** *simp*
next
fix *a Q r p fs*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *knd a* = *Q*:*r*↪*pfs* **and** *src a* = *NewEntry*
thus *False* **by**(*fastforce elim*:*lift-valid-edge.cases*)
next
fix *a a'*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'*

```
    and src a = src a' and trg a = trg a'
  thus a = a'
  proof(induct rule:lift-valid-edge.induct)
    case lve-edge thus ?case by -(erule lift-valid-edge.cases,auto dest:edge-det)
  qed(auto elim:lift-valid-edge.cases)
next
  fix a Q r f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q:r↪_Main f
  thus False by(fastforce elim:lift-valid-edge.cases dest:Main-no-call-target)
next
  fix a Q' f'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q'↩_Main f'
  thus False by(fastforce elim:lift-valid-edge.cases dest:Main-no-return-source)
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q:r↪_p fs
  thus ∃ ins outs. (p, ins, outs) ∈ set procs
    by(fastforce elim:lift-valid-edge.cases intro:callee-in-procs)
next
  fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and intra-kind (knd a)
  thus lift-get-proc get-proc Main (src a) = lift-get-proc get-proc Main (trg a)
    by(fastforce elim:lift-valid-edge.cases intro:get-proc-intra
                 simp:get-proc-Entry get-proc-Exit)
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q:r↪_p fs
  thus lift-get-proc get-proc Main (trg a) = p
    by(fastforce elim:lift-valid-edge.cases intro:get-proc-call)
next
  fix a Q' p f'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q'↩_p 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↪_p fs
  then obtain ax where valid-edge ax and kind ax = Q:r↪_p fs
    and sourcenode ax ≠ Entry ∨ targetnode ax ≠ Exit
    and src a = Node (sourcenode ax) and trg a = Node (targetnode ax)
    by(fastforce elim:lift-valid-edge.cases)
  from ‹valid-edge ax› ‹kind ax = Q:r↪_p fs›
  have all:∀ a'. valid-edge a' ∧ targetnode a' = targetnode ax ⟶
```

```
        (∃ Qx rx fsx. kind a' = Qx:rx↪pfsx)
  by(auto dest:call-edges-only)
 { fix a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
   and trg a' = trg a
  hence ∃ Qx rx fsx. knd a' = Qx:rx↪pfsx
  proof(induct rule:lift-valid-edge.induct)
   case (lve-edge ax' e)
   note [simp] = ‹e = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))›
   from ‹trg e = trg a› ‹trg a = Node (targetnode ax)›
   have targetnode ax' = targetnode ax by simp
   with ‹valid-edge ax'› all have ∃ Qx rx fsx. kind ax' = Qx:rx↪pfsx by blast
   thus ?case by simp
  next
   case (lve-Entry-edge e)
   from ‹e = (NewEntry, (λs. True)√, Node Entry)› ‹trg e = trg a›
    ‹trg a = Node (targetnode ax)›
   have targetnode ax = Entry by simp
   with ‹valid-edge ax› have False by(rule Entry-target)
   thus ?case by simp
  next
   case (lve-Exit-edge e)
   from ‹e = (Node Exit, (λs. True)√, NewExit)› ‹trg e = trg a›
    ‹trg a = Node (targetnode ax)› have False by simp
   thus ?case by simp
  next
   case (lve-Entry-Exit-edge e)
   from ‹e = (NewEntry,(λs. False)√,NewExit)› ‹trg e = trg a›
    ‹trg a = Node (targetnode ax)› have False by simp
   thus ?case by simp
  qed }
 thus ∀ a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧
        trg a' = trg a ⟶ (∃ Qx rx fsx. knd a' = Qx:rx↪pfsx) by simp
next
 fix a Q' p f'
 assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q'↩pf'
 then obtain ax where valid-edge ax and kind ax = Q'↩pf'
  and sourcenode ax ≠ Entry ∨ targetnode ax ≠ Exit
  and src a = Node (sourcenode ax) and trg a = Node (targetnode ax)
  by(fastforce elim:lift-valid-edge.cases)
 from ‹valid-edge ax› ‹kind ax = Q'↩pf'›
 have all:∀ a'. valid-edge a' ∧ sourcenode a' = sourcenode ax ⟶
       (∃ Qx fx. kind a' = Qx↩pfx)
  by(auto dest:return-edges-only)
 { fix a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
   and src a' = src a
  hence ∃ Qx fx. knd a' = Qx↩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*↩$_p$*fx* **by** *blast*
thus *?case* **by** *simp*
next
case (*lve-Entry-edge e*)
from ‹*e* = (*NewEntry*, (λ*s*. *True*)$_\surd$, *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*)$_\surd$, *NewExit*)› ‹*src e* = *src a*›
‹*src a* = *Node* (*sourcenode ax*)› **have** *sourcenode ax* = *Exit* **by** *simp*
with ‹*valid-edge ax*› **have** *False* **by**(*rule Exit-source*)
thus *?case* **by** *simp*
next
case (*lve-Entry-Exit-edge e*)
from ‹*e* = (*NewEntry*,(λ*s*. *False*)$_\surd$,*NewExit*)› ‹*src e* = *src a*›
‹*src a* = *Node* (*sourcenode ax*)› **have** *False* **by** *simp*
thus *?case* **by** *simp*
qed }
thus ∀ *a′*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′* ∧
src a′ = *src a* ⟶ (∃ *Qx fx*. *knd a′* = *Qx*↩$_p$*fx*) **by** *simp*
next
fix *a Q r p fs*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *knd a* = *Q*:*r*↪$_p$*fs*
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*))›
‹*knd e* = *Q*:*r*↪$_p$*fs*›
have *kind ax* = *Q*:*r*↪$_p$*fs* **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 $a' \in$ *lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a*
and *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
thus *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*› ‹$a' \in$ *get-return-edges ax*› **have** *valid-edge* a'
by(*rule get-return-edges-valid*)
from ‹*valid-edge ax*› ‹$a' \in$ *get-return-edges ax*› **obtain** $Q\ r\ p\ fs$
where *kind ax* $= Q{:}r{\hookrightarrow_p}fs$ **by**(*fastforce dest!:only-call-get-return-edges*)
with ‹*valid-edge ax*› ‹$a' \in$ *get-return-edges ax*› **obtain** $Q'\ f'$
where *kind* $a' = Q'{\hookleftarrow_p}f'$ **by**(*fastforce dest!:call-return-edges*)
from ‹*valid-edge* a'› ‹*kind* $a' = Q'{\hookleftarrow_p}f'$› **have** *get-proc*(*sourcenode* a') $= p$
by(*rule get-proc-return*)
have *sourcenode* $a' \neq$ *Entry*
proof
assume *sourcenode* $a' =$ *Entry*
with *get-proc-Entry* ‹*get-proc*(*sourcenode* a') $= p$› **have** $p =$ *Main* **by** *simp*
with ‹*kind* $a' = Q'{\hookleftarrow_p}f'$› **have** *kind* $a' = Q'{\hookleftarrow_{Main}}f'$ **by** *simp*
with ‹*valid-edge* a'› **show** *False* **by**(*rule Main-no-return-source*)
qed
with ‹$e' =$ (*Node* (*sourcenode* a'), *kind* a', *Node* (*targetnode* a'))›
‹*valid-edge* a'›
show *?case* **by**(*fastforce intro:lve-edge*)
qed
next
fix $a\ a'$
assume $a' \in$ *lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a*
and *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
thus $\exists\ Q\ r\ p\ fs.$ *knd a* $= Q{:}r{\hookrightarrow_p}fs$
proof (*induct rule:lift-get-return-edges.induct*)
case (*lift-get-return-edgesI ax* a' e')
from ‹*valid-edge ax*› ‹$a' \in$ *get-return-edges ax*›
have $\exists\ Q\ r\ p\ fs.$ *kind ax* $= Q{:}r{\hookrightarrow_p}fs$
by(*rule only-call-get-return-edges*)
with ‹$a =$ (*Node* (*sourcenode ax*), *kind ax*, *Node* (*targetnode ax*))›
show *?case* **by** *simp*
qed
next
fix $a\ Q\ r\ p\ fs\ a'$
assume $a' \in$ *lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a* **and** *knd a* $= Q{:}r{\hookrightarrow_p}fs$
and *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
thus $\exists\ Q'\ f'.$ *knd* $a' = Q'{\hookleftarrow_p}f'$
proof (*induct rule:lift-get-return-edges.induct*)
case (*lift-get-return-edgesI ax* a' e')
from ‹$a =$ (*Node* (*sourcenode ax*), *kind ax*, *Node* (*targetnode ax*))›

```
    ‹knd a = Q:r↪pfs›
   have kind ax = Q:r↪pfs by simp
  with ‹valid-edge ax› ‹a' ∈ get-return-edges ax› have ∃ Q' f'. kind a' = Q'↩pf'
    by −(rule call-return-edges)
   with ‹e' = (Node (sourcenode a'), kind a', Node (targetnode a'))›
   show ?case by simp
 qed
next
 fix a Q' p f'
 assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q'↩pf'
 thus ∃!a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧
  (∃ Q r fs. knd a' = Q:r↪pfs) ∧ a ∈ lift-get-return-edges get-return-edges
  valid-edge sourcenode targetnode kind a'
 proof(induct rule:lift-valid-edge.induct)
  case (lve-edge a e)
  from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
   ‹knd e = Q'↩pf'› have kind a = Q'↩pf' by simp
  with ‹valid-edge a›
  have ∃!a'. valid-edge a' ∧ (∃ Q r fs. kind a' = Q:r↪pfs) ∧
   a ∈ get-return-edges a'
   by(rule return-needs-call)
  then obtain a' Q r fs where valid-edge a' and kind a' = Q:r↪pfs
   and a ∈ get-return-edges a'
   and imp:∀ x. valid-edge x ∧ (∃ Q r fs. kind x = Q:r↪pfs) ∧
   a ∈ get-return-edges 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 ‹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 ‹kind a' = Q:r↪pfs› have knd ?e' = Q:r↪pfs by simp
  moreover
  from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
   ‹valid-edge a'› ‹a ∈ get-return-edges a'›
  have e ∈ lift-get-return-edges get-return-edges valid-edge
   sourcenode targetnode kind ?e' by(fastforce intro:lift-get-return-edgesI)
  moreover
  { fix x
   assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x
    and ∃ Q r fs. knd x = Q:r↪pfs
    and e ∈ lift-get-return-edges get-return-edges valid-edge
```

```
    sourcenode targetnode kind x
  from ‹lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x›
    ‹∃ Q r fs. knd x = Q:r↪pfs› obtain y where valid-edge y
    and x = (Node (sourcenode y), kind y, Node (targetnode y))
    by(fastforce elim:lift-valid-edge.cases)
  with ‹e ∈ lift-get-return-edges get-return-edges valid-edge
    sourcenode targetnode kind x› ‹valid-edge a›
    ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
  have x = ?e'
  proof(induct rule:lift-get-return-edges.induct)
    case (lift-get-return-edgesI ax ax' e)
    from ‹valid-edge ax› ‹ax' ∈ get-return-edges ax› have valid-edge ax'
      by(rule get-return-edges-valid)
    from ‹e = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))›
      ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
    have sourcenode a = sourcenode ax' and targetnode a = targetnode ax'
      by simp-all
    with ‹valid-edge a› ‹valid-edge ax'› have [simp]:a = ax' by(rule edge-det)
    from ‹x = (Node (sourcenode ax), kind ax, Node (targetnode ax))›
     ‹∃ Q r fs. knd x = Q:r↪pfs› have ∃ Q r fs. kind ax = Q:r↪pfs by simp
    with ‹valid-edge ax› ‹ax' ∈ get-return-edges ax› imp
    have ax = a' by fastforce
    with ‹x = (Node (sourcenode ax), kind ax, Node (targetnode ax))›
    show ?thesis by simp
  qed }
 ultimately show ?case by(blast intro:ex1I)
qed simp-all
next
 fix a a'
 assume a' ∈ 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 ∃ a''. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'' ∧
   src a'' = trg a ∧ trg a'' = src a' ∧ knd a'' = (λcf. False)√
 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›
   obtain ax' where valid-edge ax' and sourcenode ax' = targetnode ax
     and targetnode ax' = sourcenode a' and kind ax' = (λcf. False)√
     by(fastforce dest:intra-proc-additional-edge)
   let ?ex = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))
   have targetnode ax ≠ Entry
   proof
     assume targetnode ax = Entry
     with ‹valid-edge ax› show False by(rule Entry-target)
   qed
  with ‹sourcenode ax' = targetnode ax› have sourcenode ax' ≠ Entry by simp
   with ‹valid-edge ax'›
   have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?ex
```

```
    by(fastforce intro:lve-edge)
   with ‹e' = (Node (sourcenode a'), kind a', Node (targetnode a'))›
    ‹a = (Node (sourcenode ax), kind ax, Node (targetnode ax))›
    ‹e' = (Node (sourcenode a'), kind a', Node (targetnode a'))›
    ‹sourcenode ax' = targetnode ax› ‹targetnode ax' = sourcenode a'›
    ‹kind ax' = (λcf. False)√›
   show ?case by simp
  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 ∃ a''. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'' ∧
    src a'' = src a ∧ trg a'' = trg a' ∧ knd a'' = (λcf. False)√
  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›
   obtain ax' where valid-edge ax' and sourcenode ax' = sourcenode ax
    and targetnode ax' = targetnode a' and kind ax' = (λcf. False)√
    by(fastforce dest:call-return-node-edge)
   let ?ex = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))
   from ‹valid-edge ax› ‹a' ∈ get-return-edges ax›
   obtain Q r p fs where kind ax = Q:r↪pfs
    by(fastforce dest!:only-call-get-return-edges)
   have sourcenode ax ≠ Entry
   proof
    assume sourcenode ax = Entry
    with ‹valid-edge ax› ‹kind ax = Q:r↪pfs› show False
     by(rule Entry-no-call-source)
   qed
  with ‹sourcenode ax' = sourcenode ax› have sourcenode ax' ≠ Entry by simp
   with ‹valid-edge ax'›
   have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?ex
    by(fastforce intro:lve-edge)
   with ‹e' = (Node (sourcenode a'), kind a', Node (targetnode a'))›
    ‹a = (Node (sourcenode ax), kind ax, Node (targetnode ax))›
    ‹e' = (Node (sourcenode a'), kind a', Node (targetnode a'))›
    ‹sourcenode ax' = sourcenode ax› ‹targetnode ax' = targetnode a'›
    ‹kind ax' = (λcf. False)√›
   show ?case by simp
  qed
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
   and knd a = Q:r↪pfs
  thus ∃!a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧
    src a' = src 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: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 sourcenode targetnode kind Entry Exit ?e'
     by(fastforce intro:lift-valid-edge.lve-edge)
    moreover
    from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
     ‹sourcenode a' = sourcenode a›
    have src ?e' = src e by simp
    moreover
    from ‹intra-kind(kind a')› have intra-kind (knd ?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
    qed }
  ultimately show ?case by(blast intro:ex1I)
 qed simp-all
next
 fix a Q' p f'
 assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q'↪_p f'
 thus ∃!a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧
  trg a' = trg 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))›
```

```
    ‹targetnode a' = targetnode a›
  have trg ?e' = trg e by simp
  moreover
  from ‹intra-kind(kind a')› have intra-kind (knd ?e') by simp
  moreover
  { fix x
    assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit x
      and trg x = trg 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› ‹trg x = trg 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 targetnode ax = targetnode 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 ‹trg x = trg e›
        ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
      have targetnode a = Entry by simp
      with ‹valid-edge a› have False by(rule Entry-target)
      thus ?case by simp
    next
      case (lve-Exit-edge ex)
      with ‹trg x = trg e›
        ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
      have False by simp
      thus ?case by simp
    next
      case (lve-Entry-Exit-edge ex)
      with ‹trg x = trg e›
        ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
      have False by simp
      thus ?case by simp
    qed }
  ultimately show ?case by(blast intro:ex1I)
 qed simp-all
next
 fix a a' Q1 r1 p fs1 Q2 r2 fs2
 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 knd a = Q1:r1↪pfs1 and knd a' = Q2:r2↪pfs2
 then obtain x x' where valid-edge x
   and a:a = (Node (sourcenode x),kind x,Node (targetnode x)) and valid-edge
```

```
x'
      and a':a' = (Node (sourcenode x'),kind x',Node (targetnode x'))
      by(auto elim!:lift-valid-edge.cases)
    with ‹knd a = Q₁:r₁↪ₚfs₁› ‹knd a' = Q₂:r₂↪ₚfs₂›
    have kind x = Q₁:r₁↪ₚfs₁ and kind x' = Q₂:r₂↪ₚfs₂ by simp-all
    with ‹valid-edge x› ‹valid-edge x'› have targetnode x = targetnode x'
      by(rule same-proc-call-unique-target)
    with a a' show trg a = trg a' by simp
  next
    from unique-callers show distinct-fst procs .
  next
    fix p ins outs
    assume (p, ins, outs) ∈ set procs
    from distinct-formal-ins[OF this] show distinct ins .
  next
    fix p ins outs
    assume (p, ins, outs) ∈ set procs
    from distinct-formal-outs[OF this] show distinct outs .
  qed
qed


lemma lift-CFG-wf:
  assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
  get-return-edges procs Main Exit Def Use ParamDefs ParamUses
  and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
  get-return-edges procs Main Exit
  shows CFG-wf src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-get-proc get-proc Main)
  (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
  procs Main (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
  (lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
proof −
  interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
    get-return-edges procs Main Exit Def Use ParamDefs ParamUses
    by(rule wf)
  interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc
    get-return-edges procs Main Exit
    by(rule pd)
  interpret CFG:CFG src trg knd
    lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
    lift-get-proc get-proc Main
    lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
    procs Main
    by(fastforce intro:lift-CFG wf pd)
  show ?thesis
  proof
    show lift-Def Def Entry Exit H L NewEntry = {} ∧
```

```
      lift-Use Use Entry Exit H L NewEntry = {}
    by(fastforce elim:lift-Use-set.cases lift-Def-set.cases)
  next
    fix a Q r p fs ins outs
    assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and knd a = Q:r↪pfs and (p, ins, outs) ∈ set procs
    thus length (lift-ParamUses ParamUses (src a)) = length ins
    proof(induct rule:lift-valid-edge.induct)
      case (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 assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    thus distinct (lift-ParamDefs ParamDefs (trg a))
    proof(induct rule:lift-valid-edge.induct)
      case (lve-edge a e)
      from ‹valid-edge a› have distinct (ParamDefs (targetnode a))
        by(rule distinct-ParamDefs)
      with ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
      show ?case by simp
    next
      case (lve-Entry-edge e)
      have ParamDefs Entry = []
      proof(rule ccontr)
        assume ParamDefs Entry ≠ []
        then obtain V Vs where ParamDefs Entry = V#Vs
          by(cases ParamDefs Entry) auto
        hence V ∈ set (ParamDefs Entry) by fastforce
        hence V ∈ Def Entry by(fastforce intro:ParamDefs-in-Def)
        with Entry-empty show False by simp
      qed
      with ‹e = (NewEntry, (λs. True)√, Node Entry)› show ?case by simp
    qed simp-all
  next
    fix a Q' p f' ins outs
    assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and knd a = Q'↩pf' and (p, ins, outs) ∈ set procs
    thus length (lift-ParamDefs ParamDefs (trg a)) = length outs
    proof(induct rule:lift-valid-edge.induct)
      case (lve-edge a e)
      from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
        ‹knd e = Q'↩pf'›
      have kind a = Q'↩pf' and trg e = Node (targetnode a) by simp-all
```

```
    with ‹valid-edge a› ‹(p,ins,outs) ∈ set procs›
    have length(ParamDefs (targetnode a)) = length outs
      by −(rule ParamDefs-return-target-length)
    with ‹trg e = Node (targetnode a)› show ?case by simp
  qed simp-all
 next
  fix n V
  assume CFG.CFG.valid-node src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
    and V ∈ set (lift-ParamDefs ParamDefs n)
   hence ((n = NewEntry) ∨ n = NewExit) ∨ (∃ m. n = Node m ∧ valid-node
m)
    by(auto elim:lift-valid-edge.cases simp:CFG.valid-node-def)
  thus V ∈ lift-Def Def Entry Exit H L n apply −
  proof(erule disjE)+
    assume n = NewEntry
    with ‹V ∈ set (lift-ParamDefs ParamDefs n)› show ?thesis by simp
  next
     assume n = NewExit
    with ‹V ∈ set (lift-ParamDefs ParamDefs n)› show ?thesis by simp
  next
    assume ∃ m. n = Node m ∧ valid-node m
    then obtain m where n = Node m and valid-node m by blast
    from ‹n = Node m› ‹V ∈ set (lift-ParamDefs ParamDefs n)›
    have V ∈ set (ParamDefs m) by simp
    with ‹valid-node m› have V ∈ Def m by(rule ParamDefs-in-Def)
    with ‹n = Node m› show ?thesis by(fastforce intro:lift-Def-node)
  qed
 next
  fix a Q r p fs ins outs V
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q:r↪pfs and (p, ins, outs) ∈ set procs and V ∈ set ins
  thus V ∈ lift-Def Def Entry Exit H L (trg 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:r↪pfs›
    have kind a = Q:r↪pfs by simp
   from ‹valid-edge a› ‹kind a = Q:r↪pfs› ‹(p, ins, outs) ∈ set procs› ‹V ∈ set
ins›
    have V ∈ Def (targetnode a) by(rule ins-in-Def)
    from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
    have trg e = Node (targetnode a) by simp
    with ‹V ∈ Def (targetnode a)› show ?case by(fastforce intro:lift-Def-node)
  qed simp-all
 next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and knd a = Q:r↪pfs
```

```
    thus lift-Def Def Entry Exit H L (src a) = {}
    proof(induct rule:lift-valid-edge.induct)
      case (lve-edge a e)
      show ?case
      proof(rule ccontr)
        assume lift-Def Def Entry Exit H L (src e) ≠ {}
        then obtain x where x ∈ lift-Def Def Entry Exit H L (src e) by blast
         from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))› ‹knd e =
Q:r↪pfs›
        have kind a = Q:r↪pfs by simp
        with ‹valid-edge a› have Def (sourcenode a) = {}
          by(rule call-source-Def-empty)
        have sourcenode a ≠ Entry
        proof
          assume sourcenode a = Entry
          with ‹valid-edge a› ‹kind a = Q:r↪pfs›
          show False by(rule Entry-no-call-source)
        qed
        from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
        have src e = Node (sourcenode a) by simp
        with ‹Def (sourcenode a) = {}› ‹x ∈ lift-Def Def Entry Exit H L (src e)›
          ‹sourcenode a ≠ Entry›
        show False by(fastforce elim:lift-Def-set.cases)
      qed
    qed simp-all
  next
    fix n V
    assume CFG.CFG.valid-node src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
      and V ∈ ⋃(set (lift-ParamUses ParamUses n))
     hence ((n = NewEntry) ∨ n = NewExit) ∨ (∃ m. n = Node m ∧ valid-node
m)
      by(auto elim:lift-valid-edge.cases simp:CFG.valid-node-def)
    thus V ∈ lift-Use Use Entry Exit H L n apply −
    proof(erule disjE)+
      assume n = NewEntry
      with ‹V ∈ ⋃(set (lift-ParamUses ParamUses n))› show ?thesis by simp
    next
      assume n = NewExit
      with ‹V ∈ ⋃(set (lift-ParamUses ParamUses n))› show ?thesis by simp
    next
      assume ∃ m. n = Node m ∧ valid-node m
      then obtain m where n = Node m and valid-node m by blast
      from ‹V ∈ ⋃(set (lift-ParamUses ParamUses n))› ‹n = Node m›
      have V ∈ ⋃(set (ParamUses m)) by simp
      with ‹valid-node m› have V ∈ Use m by(rule ParamUses-in-Use)
      with ‹n = Node m› show ?thesis by(fastforce intro:lift-Use-node)
    qed
  next
```

fix *a Q p f ins outs V*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *knd a = Q↩pf* **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 (*lve-edge a e*)
from ‹*e = (Node (sourcenode a), kind a, Node (targetnode a))*› ‹*knd e = Q↩pf*›
have *kind a = Q↩pf* **by** *simp*
from ‹*valid-edge a*› ‹*kind a = Q↩pf*› ‹*(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 (*lve-edge a e*)
from ‹*e = (Node (sourcenode a), kind a, Node (targetnode a))*›
‹*intra-kind (knd e)*› ‹*pred (knd e) s*›
have *intra-kind (kind a)* **and** *pred (kind a) s*
and *knd e = kind a* **and** *src e = Node (sourcenode a)* **by** *simp-all*
from ‹*V ∉ lift-Def Def Entry Exit H L (src e)*› ‹*src e = Node (sourcenode a)*›
have *V ∉ Def (sourcenode a)* **by** (*auto dest*: *lift-Def-node*)
from ‹*valid-edge a*› ‹*V ∉ Def (sourcenode a)*› ‹*intra-kind (kind a)*›
‹*pred (kind a) s*›
have *state-val (transfer (kind 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 (*lve-Entry-edge e*)
from ‹*e = (NewEntry, (λs. True)√, Node Entry)*› ‹*pred (knd e) s*›
show *?case* **by**(*cases s*) *auto*
next
case (*lve-Exit-edge e*)
from ‹*e = (Node Exit, (λs. True)√, NewExit)*› ‹*pred (knd e) s*›
show *?case* **by**(*cases s*) *auto*
next
case (*lve-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
a
      ∀ V∈lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
      intra-kind (knd a) pred (knd a) s pred (knd a) s'
    show ∀ V∈lift-Def Def Entry Exit H L (src a).
      state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
    proof
      fix V assume V ∈ lift-Def Def Entry Exit H L (src a)
      with assms
      show state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
      proof(induct rule:lift-valid-edge.induct)
        case (lve-edge a e)
        from ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
          ‹intra-kind (knd e)› have intra-kind (kind a) by simp
        show ?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)√
          proof(cases targetnode a = Exit)
            case True
            with ‹valid-edge a› ‹valid-edge a'› ‹sourcenode a = Entry›
              ‹sourcenode a' = Entry› ‹targetnode a' = Exit›
            have a = a' by(fastforce dest:edge-det)
            with ‹kind a' = (λs. False)√› show ?thesis by simp
          next
            case False
            with ‹valid-edge a› ‹valid-edge a'› ‹sourcenode a = Entry›
              ‹sourcenode a' = Entry› ‹targetnode a' = Exit›
              ‹intra-kind (kind a)› ‹kind a' = (λs. False)√›
            show ?thesis by(auto dest:deterministic simp:intra-kind-def)
          qed
          from True ‹V ∈ lift-Def Def Entry Exit H L (src e)› Entry-empty
            ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
          have V ∈ H by(fastforce elim:lift-Def-set.cases)
          from True ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
            ‹sourcenode a ≠ Entry ∨ targetnode a ≠ Exit›
          have ∀ V∈H. V ∈ lift-Use Use Entry Exit H L (src e)
            by(fastforce intro:lift-Use-High)
          with ‹∀ V∈lift-Use Use Entry Exit H L (src e).
            state-val s V = state-val s' V› ‹V ∈ H›
          have state-val s V = state-val s' V by simp
          with ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
```

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

and $\forall\, V{\in}$*lift-Use Use Entry Exit H L* (*src a*). *state-val s V* = *state-val s′ V*
and *length s* = *length s′*
thus *pred* (*knd a*) *s′*
proof(*induct rule*:*lift-valid-edge.induct*)
case (*lve-edge a e*)
from ‹*e* = (*Node* (*sourcenode a*), *kind a*, *Node* (*targetnode a*))› ‹*pred* (*knd e*) *s*›
have *pred* (*kind a*) *s* **and** *src e* = *Node* (*sourcenode a*) **by** *simp-all*
from ‹*src e* = *Node* (*sourcenode a*)›
‹$\forall\, V{\in}$*lift-Use Use Entry Exit H L* (*src e*). *state-val s V* = *state-val s′ V*›
have $\forall\, V \in$ *Use* (*sourcenode a*). *state-val s V* = *state-val s′ V*
by(*auto dest*:*lift-Use-node*)
from ‹*valid-edge a*› ‹*pred* (*kind 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 (*lve-Entry-edge e*)
thus *?case* **by**(*cases s′*) *auto*
next
case (*lve-Exit-edge e*)
thus *?case* **by**(*cases s′*) *auto*
next
case (*lve-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*↪$_p$*fs* **and** (*p*, *ins*, *outs*) ∈ *set procs*
thus *length fs* = *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*↪$_p$*fs*›
have *kind a* = *Q*:*r*↪$_p$*fs* **by** *simp*
from ‹*valid-edge a*› ‹*kind a* = *Q*:*r*↪$_p$*fs*› ‹(*p*, *ins*, *outs*) ∈ *set procs*›
show *?case* **by**(*rule CFG-call-edge-length*)
qed *simp-all*
next
fix *a Q r p fs a′ Q′ r′ p′ fs′ s s′*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *knd a* = *Q*:*r*↪$_p$*fs* **and** *knd a′* = *Q′*:*r′*↪$_{p'}$*fs′*
and *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′*
and *src a* = *src a′* **and** *pred* (*knd a*) *s* **and** *pred* (*knd a′*) *s*
from ‹*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*›
‹*knd a* = *Q*:*r*↪$_p$*fs*› ‹*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)*
and *kind x = Q:r↪pfs* **and** *pred (kind x) s*
by(*fastforce elim:lift-valid-edge.cases*)
from ‹*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'*›
‹*knd a' = Q':r'↪p'fs'*› ‹*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')*
and *kind x' = Q':r'↪p'fs'* **and** *pred (kind x') s*
by(*fastforce elim:lift-valid-edge.cases*)
from ‹*src a = Node (sourcenode x)*› ‹*src a' = Node (sourcenode x')*›
‹*src a = src a'*›
have *sourcenode x = sourcenode x'* **by** *simp*
from ‹*valid-edge x*› ‹*kind x = Q:r↪pfs*› ‹*valid-edge x'*› ‹*kind x' = Q':r'↪p'fs'*›
‹*sourcenode x = sourcenode x'*› ‹*pred (kind x) s*› ‹*pred (kind x') s*›
have *x = x'* **by**(*rule CFG-call-determ*)
with *a a'* **show** *a = a'* **by** *simp*
next
fix *a Q r p fs i ins outs s s'*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *knd a = Q:r↪pfs* **and** *i < length ins* **and** *(p, ins, outs) ∈ set procs*
and *pred (knd a) s* **and** *pred (knd a) s'*
and ∀ *V∈lift-ParamUses ParamUses (src a) ! i. state-val s V = state-val s' V*
thus *params fs (state-val s) ! i = CFG.params fs (state-val s') ! i*
proof(*induct rule:lift-valid-edge.induct*)
case (*lve-edge a e*)
from ‹*e = (Node (sourcenode a), kind a, Node (targetnode a))*› ‹*knd e = Q:r↪pfs*›
‹*pred (knd e) s*› ‹*pred (knd e) s'*›
have *kind a = Q:r↪pfs* **and** *pred (kind a) s* **and** *pred (kind a) s'*
and *src e = Node (sourcenode a)*
by *simp-all*
from ‹∀ *V∈lift-ParamUses ParamUses (src e) ! i. state-val s V = state-val s' V*›
‹*src e = Node (sourcenode a)*›
have ∀ *V ∈ (ParamUses (sourcenode a))!i. state-val s V = state-val s' V* **by** *simp*
with ‹*valid-edge a*› ‹*kind a = Q:r↪pfs*› ‹*i < length ins*›
‹*(p, ins, outs) ∈ set procs*› ‹*pred (kind a) s*› ‹*pred (kind a) s'*›
show *?case* **by**(*rule CFG-call-edge-params*)
qed *simp-all*
next
fix *a Q' p f' ins outs cf cf'*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *knd a = Q'↩pf'* **and** *(p, ins, outs) ∈ set procs*
thus *f' cf 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))*› ‹*knd e = Q'↩pf'*›

```
    have kind a = Q'↪pf' and trg e = Node (targetnode a) by simp-all
    from ‹valid-edge a› ‹kind a = Q'↪pf'› ‹(p, ins, outs) ∈ set procs›
    have f' cf cf' = cf'(ParamDefs (targetnode a) [:=] map cf outs)
      by(rule CFG-return-edge-fun)
    with ‹trg e = Node (targetnode a)› show ?case by simp
  qed simp-all
 next
  fix a a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
    and src a = src a' and trg a ≠ trg a'
    and intra-kind (knd a) and intra-kind (knd a')
  thus ∃ Q Q'. knd a = (Q)√ ∧ knd a' = (Q')√ ∧
              (∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s))
  proof(induct rule:lift-valid-edge.induct)
    case (lve-edge a e)
    from ‹lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'›
      ‹valid-edge a› ‹e = (Node (sourcenode a), kind a, Node (targetnode a))›
      ‹src e = src a'› ‹trg e ≠ trg a'› ‹intra-kind (knd e)› ‹intra-kind (knd a')›
    show ?case
    proof(induct rule:lift-valid-edge.induct)
      case lve-edge thus ?case by(auto dest:deterministic)
    next
      case (lve-Exit-edge e')
      from ‹e = (Node (sourcenode 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* **and** *src a* = *NewExit*
thus *False* **by**(*fastforce elim*:*lift-valid-edge.cases*)
next
show *lift-get-proc get-proc Main NewExit* = *Main* **by** *simp*
next
fix *a Q p f*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a* **and** *knd a* = $Q\hookleftarrow_p f$ **and** *trg a* = *NewExit*
thus *False* **by**(*fastforce elim*:*lift-valid-edge.cases*)
next
show $\exists$ *a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a* $\wedge$ *src a* = *NewEntry* $\wedge$ *trg a* = *NewExit* $\wedge$ *knd a* = $(\lambda s.\ False)_{\surd}$
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
qed

3.2.2 Lifting the SDG

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

```
  have CFG.CFG.valid-node src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) (Node m)
  proof
    assume m = sourcenode a
    show ?thesis
    proof(cases m = Entry)
      case True
      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› show ?thesis by(fastforce simp:CFGExit.valid-node-def)
    next
      case False
      with ‹m = sourcenode a› ‹valid-edge a›
      have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
        (Node (sourcenode a),kind a,Node(targetnode a))
        by(fastforce intro:lve-edge)
    with ‹m = sourcenode a› show ?thesis by(fastforce simp:CFGExit.valid-node-def)
    qed
  next
    assume m = targetnode a
    show ?thesis
    proof(cases m = Exit)
      case True
      have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
        (Node Exit,(λs. True)√,NewExit) by(fastforce intro:lve-Exit-edge)
      with ‹m = Exit› show ?thesis by(fastforce simp:CFGExit.valid-node-def)
    next
      case False
      with ‹m = targetnode a› ‹valid-edge a›
      have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
        (Node (sourcenode a),kind a,Node(targetnode a))
        by(fastforce intro:lve-edge)
    with ‹m = targetnode a› show ?thesis by(fastforce simp:CFGExit.valid-node-def)
    qed
  qed }
note lift-valid-node = this
{ fix n as n' cs m m'
  assume valid-path-aux cs as and m −as→* m' and ∀ c ∈ set cs. valid-edge c
    and m ≠ Entry ∨ m' ≠ Exit
  hence ∃ cs' es. CFG.CFG.valid-path-aux knd
    (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
    cs' es ∧
    list-all2 (λc c'. c' = (Node (sourcenode c),kind c,Node (targetnode c))) cs cs'
     ∧ CFG.CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node m) es (Node m')
  proof(induct arbitrary:m rule:vpa-induct)
    case (vpa-empty cs)
    from ‹m −[]→* m'› have [simp]:m = m' by fastforce
```

```
    from ‹m −[]→* m'› have valid-node m by(rule path-valid-node)
    obtain cs' where cs' =
      map (λc. (Node (sourcenode c),kind c,Node (targetnode c))) cs by simp
    hence list-all2
      (λc c'. c' = (Node (sourcenode c),kind c,Node (targetnode c))) cs cs'
      by(simp add:list-all2-conv-all-nth)
    with ‹valid-node m› show ?case
      apply(rule-tac x=cs' in exI)
      apply(rule-tac x=[] in exI)
      by(fastforce intro:CFGExit.empty-path lift-valid-node)
  next
    case (vpa-intra cs a as)
    note IH = ‹⋀m. ⟦m −as→* m'; ∀ c∈set 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))) 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)
    show ?case
    proof(cases sourcenode a = Entry ∧ targetnode a = Exit)
      case True
      with ‹m = sourcenode a› ‹m ≠ Entry ∨ m' ≠ Exit›
      have m' ≠ Exit by simp
      from True have targetnode a = Exit by simp
      with ‹targetnode a −as→* m'› have m' = Exit
        by −(drule path-Exit-source,auto)
      with ‹m' ≠ Exit› have False by simp
      thus ?thesis by simp
    next
      case False
      let ?e = (Node (sourcenode a),kind a,Node (targetnode a))
      from False ‹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 −as→* m'› ‹∀ c∈set cs. valid-edge c› this]
      obtain cs' es
        where valid-path:CFG.valid-path-aux knd
        (lift-get-return-edges get-return-edges valid-edge sourcenode
```

```
    targetnode kind) cs' es
    and list:list-all2
    (λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c))) cs cs'
    and path:CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node (targetnode a)) es (Node m') by blast
  from ‹intra-kind (kind a)› valid-path have CFG.valid-path-aux knd
    (lift-get-return-edges get-return-edges valid-edge sourcenode
    targetnode kind) cs' (?e#es) by(fastforce simp:intra-kind-def)
  moreover
  from path ‹m = sourcenode a›
    ‹lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e›
  have CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node m) (?e#es) (Node m') by(fastforce intro:CFGExit.Cons-path)
  ultimately show ?thesis using list by blast
 qed
next
 case (vpa-Call cs a as Q r p fs)
 note IH = ‹⋀m. ⟦m −as→* m'; ∀ c∈set (a # cs). valid-edge c;
   m ≠ Entry ∨ m' ≠ Exit⟧ ⟹
   ∃ cs' es. CFG.valid-path-aux knd
   (lift-get-return-edges get-return-edges valid-edge sourcenode
   targetnode kind) cs' es ∧
   list-all2 (λc c'. c' = (Node (sourcenode c), kind c, Node (targetnode c)))
   (a#cs) cs' ∧ CFG.path src trg
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   (Node m) es (Node m')›
 from ‹m −a # as→* m'› have m = sourcenode a and valid-edge a
   and targetnode a −as→* m' by(auto elim:path-split-Cons)
 from ‹∀ c∈set cs. valid-edge c› ‹valid-edge a›
 have ∀ c∈set (a # cs). valid-edge c by simp
 let ?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 ‹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 −as→* m'› ‹∀ c∈set (a # cs). valid-edge c› this]
```

obtain *cs′ es*
where *valid-path*:*CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) *cs′ es*
and *list*:*list-all2*
(λ*c c′*. *c′* = (*Node* (*sourcenode c*), *kind c*, *Node* (*targetnode c*))) (*a*#*cs*) *cs′*
and *path*:*CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node* (*targetnode a*)) *es* (*Node m′*) **by** *blast*
from *list* **obtain** *cx csx* **where** *cs′* = *cx*#*csx*
and *cx*:*cx* = (*Node* (*sourcenode a*), *kind a*, *Node* (*targetnode a*))
and *list′*:*list-all2*
(λ*c c′*. *c′* = (*Node* (*sourcenode c*), *kind c*, *Node* (*targetnode c*))) *cs csx*
by(*fastforce simp*:*list-all2-Cons1*)
from *valid-path cx* ‹*cs′* = *cx*#*csx*› ‹*kind a* = *Q*:*r*↪$_p$*fs*›
have *CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) *csx* (*?e*#*es*) **by** *simp*
moreover
from *path* ‹*m* = *sourcenode a*›
‹*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e*›
have *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node m*) (*?e*#*es*) (*Node m′*) **by**(*fastforce intro*:*CFGExit.Cons-path*)
ultimately show *?case* **using** *list′* **by** *blast*
next
case (*vpa-ReturnEmpty cs a as Q p f*)
note *IH* = ‹⋀*m*. ⟦*m* −*as*→∗ *m′*; ∀ *c*∈*set* []. *valid-edge c*; *m* ≠ *Entry* ∨ *m′* ≠ *Exit*⟧ ⟹
∃ *cs′ es*. *CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) *cs′ es* ∧
list-all2 (λ*c c′*. *c′* = (*Node* (*sourcenode c*), *kind c*, *Node* (*targetnode c*)))
[] *cs′* ∧ *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node m*) *es* (*Node m′*)›
from ‹*m* −*a* # *as*→∗ *m′*› **have** *m* = *sourcenode a* **and** *valid-edge a*
and *targetnode a* −*as*→∗ *m′* **by**(*auto elim*:*path-split-Cons*)
let *?e* = (*Node* (*sourcenode a*),*kind a*,*Node* (*targetnode a*))
have *targetnode a* ≠ *Exit*
proof
assume *targetnode a* = *Exit*
with ‹*valid-edge a*› ‹*kind a* = *Q*↩$_p$*f*› **show** *False* **by**(*rule Exit-no-return-target*)
qed
with ‹*valid-edge a*›
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e*
by(*fastforce intro*:*lve-edge*)
have *targetnode a* ≠ *Entry*
proof

assume *targetnode a = Entry*
with ‹*valid-edge a*› **show** *False* **by**(*rule Entry-target*)
qed
hence *targetnode a ≠ Entry ∨ m' ≠ Exit* **by** *simp*
from *IH*[*OF* ‹*targetnode a −as→∗ m'*› *- this*] **obtain** *es*
where *valid-path*:*CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) [] *es*
and *path*:*CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node* (*targetnode a*)) *es* (*Node m'*) **by** *auto*
from *valid-path* ‹*kind a = Q↩pf*›
have *CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) [] (*?e#es*) **by** *simp*
moreover
from *path* ‹*m = sourcenode a*›
‹*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e*›
have *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node m*) (*?e#es*) (*Node m'*) **by**(*fastforce intro*:*CFGExit.Cons-path*)
ultimately show *?case* **using** ‹*cs = []*› **by** *blast*
next
case (*vpa-ReturnCons cs a as Q p f c' cs'*)
note *IH* = ‹⋀*m*. ⟦*m −as→∗ m'*; ∀ *c*∈*set cs'*. *valid-edge c*; *m ≠ Entry ∨ m' ≠ Exit*⟧ ⟹
∃ *csx es*. *CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) *csx es* ∧
list-all2 (λ*c c'*. *c'* = (*Node* (*sourcenode c*), *kind c*, *Node* (*targetnode c*)))
cs' csx ∧ *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node m*) *es* (*Node m'*)›
from ‹*m −a # as→∗ m'*› **have** *m = sourcenode a* **and** *valid-edge a*
and *targetnode a −as→∗ m'* **by**(*auto elim*:*path-split-Cons*)
from ‹∀ *c*∈*set cs*. *valid-edge c*› ‹*cs = c' # cs'*›
have *valid-edge c'* **and** ∀ *c*∈*set cs'*. *valid-edge c* **by** *simp-all*
let *?e* = (*Node* (*sourcenode a*),*kind a*,*Node* (*targetnode a*))
have *targetnode a ≠ Exit*
proof
assume *targetnode a = Exit*
with ‹*valid-edge a*› ‹*kind a = Q↩pf*› **show** *False* **by**(*rule Exit-no-return-target*)
qed
with ‹*valid-edge a*›
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e*
by(*fastforce intro*:*lve-edge*)
have *targetnode a ≠ Entry*
proof
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'*› ‹∀ *c*∈*set cs'*. *valid-edge c*› *this*]
obtain *csx es*
where *valid-path*:*CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) *csx es*
and *list*:*list-all2*
(λ*c c'*. *c'* = (*Node* (*sourcenode c*), *kind c*, *Node* (*targetnode c*))) *cs' csx*
and *path*:*CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node* (*targetnode a*)) *es* (*Node m'*) **by** *blast*
from ‹*valid-edge c'*› ‹*a* ∈ *get-return-edges c'*›
have *?e* ∈ *lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind (*Node* (*sourcenode c'*),*kind c'*,*Node* (*targetnode c'*))
by(*fastforce intro*:*lift-get-return-edgesI*)
with *valid-path* ‹*kind a* = *Q*↩*pf*›
have *CFG.valid-path-aux knd*
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
((*Node* (*sourcenode c'*),*kind c'*,*Node* (*targetnode c'*))#*csx*) (*?e*#*es*)
by *simp*
moreover
from *list* ‹*cs* = *c'* # *cs'*›
have *list-all2*
(λ*c c'*. *c'* = (*Node* (*sourcenode c*), *kind c*, *Node* (*targetnode c*))) *cs*
((*Node* (*sourcenode c'*),*kind c'*,*Node* (*targetnode c'*))#*csx*)
by *simp*
moreover
from *path* ‹*m* = *sourcenode a*›
‹*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e*›
have *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node m*) (*?e*#*es*) (*Node m'*) **by**(*fastforce intro*:*CFGExit.Cons-path*)
ultimately show *?case* **using** ‹*kind a* = *Q*↩*pf*› **by** *blast*
qed }
hence *lift-valid-path*:⋀*m as m'*. ⟦*m* −*as*→√∗ *m'*; *m* ≠ *Entry* ∨ *m'* ≠ *Exit*⟧
⟹ ∃ *es*. *CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
(*Node 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 trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *n*
hence ((*n* = *NewEntry*) ∨ *n* = *NewExit*) ∨ (∃ *m*. *n* = *Node m* ∧ *valid-node*
m)
by(*auto elim*:*lift-valid-edge.cases simp*:*CFGExit.valid-node-def*)

thus $\exists$ *as. CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
NewEntry as n **apply** −
proof(*erule disjE*)+
assume *n = NewEntry*
hence *CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
NewEntry [] n
by(*fastforce intro*:*CFGExit.empty-path*
simp:*CFGExit.vp-def CFGExit.valid-path-def*)
thus *?thesis* **by** *blast*
next
assume *n = NewExit*
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
(*NewEntry*,$(\lambda s.\ False)_{\surd}$,*NewExit*) **by**(*fastforce intro*:*lve-Entry-Exit-edge*)
hence *CFG.CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry [(*NewEntry*,$(\lambda s.\ False)_{\surd}$,*NewExit*)] *NewExit*
by(*fastforce dest*:*CFGExit.path-edge*)
with ‹*n = NewExit*› **have** *CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
NewEntry [(*NewEntry*,$(\lambda s.\ False)_{\surd}$,*NewExit*)] *n*
by(*fastforce simp*:*CFGExit.vp-def CFGExit.valid-path-def*)
thus *?thesis* **by** *blast*
next
assume $\exists$ *m. n = Node m* $\wedge$ *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*,$(\lambda s.\ True)_{\surd}$,*Node Entry*) **by**(*fastforce intro*:*lve-Entry-edge*)
with ‹*m = Entry*› ‹*n = Node m*› **have** *CFG.CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry [(*NewEntry*,$(\lambda s.\ True)_{\surd}$,*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*
(*Node* (*sourcenode ax*),*kind ax*,*Node Exit*)

by(*auto intro*:*lift-valid-edge.lve-edge simp*:*inner-node-def*)
hence *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node* (*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 trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind)
(*Node* (*sourcenode ax*)) [(*Node* (*sourcenode ax*),*kind ax*,*Node Exit*)]
(*Node Exit*)
by(*fastforce simp*:*CFGExit.slp-def CFGExit.same-level-path-def*
intra-kind-def)
have *sourcenode ax* ≠ *Exit*
proof
assume *sourcenode ax* = *Exit*
with ‹*valid-edge ax*› **show** *False* **by**(*rule Exit-source*)
qed
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
(*NewEntry*,$(\lambda s.\ True)_{\surd}$,*Node Entry*) **by**(*fastforce intro*:*lve-Entry-edge*)
hence *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*NewEntry*) [(*NewEntry*,$(\lambda s.\ True)_{\surd}$,*Node Entry*)] (*Node Entry*)
by(*fastforce intro*:*CFGExit.Cons-path CFGExit.empty-path*
simp:*CFGExit.valid-node-def*)
hence *slp-edge′*:*CFG.CFG.same-level-path′ src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind)
(*NewEntry*) [(*NewEntry*,$(\lambda s.\ True)_{\surd}$,*Node Entry*)] (*Node Entry*)
by(*fastforce simp*:*CFGExit.slp-def CFGExit.same-level-path-def*)
from ‹*inner-node* (*sourcenode ax*)› **have** *valid-node* (*sourcenode ax*)
by(*rule inner-is-valid*)
then obtain *asx* **where** *Entry* $-asx\rightarrow_{\surd}*$ *sourcenode ax*
by(*fastforce dest*:*Entry-path*)
with ‹*sourcenode ax* ≠ *Exit*›
have ∃ *es*. *CFG.CFG.valid-path′ src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) (*Node Entry*) *es* (*Node* (*sourcenode ax*))
by(*fastforce intro*:*lift-valid-path*)
then obtain *es* **where** *CFG.CFG.valid-path′ src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) (*Node Entry*) *es* (*Node* (*sourcenode ax*)) **by** *blast*
with *slp-edge* **have** *CFG.CFG.valid-path′ src trg knd*

```
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (lift-get-return-edges get-return-edges valid-edge sourcenode
      targetnode kind)
     (Node Entry) (es@[(Node (sourcenode ax),kind ax,Node Exit)]) (Node Exit)
      by −(rule CFGExit.vp-slp-Append)
    with slp-edge′ have CFG.CFG.valid-path′ src trg knd
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (lift-get-return-edges get-return-edges valid-edge sourcenode
      targetnode kind) NewEntry
      ([(NewEntry,(λs. True)√,Node Entry)]@
      (es@[(Node (sourcenode ax),kind ax,Node Exit)])) (Node Exit)
      by(rule CFGExit.slp-vp-Append)
    with ‹m = Exit› ‹n = Node m› show ?thesis by simp blast
   next
    case inner
    have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (NewEntry,(λs. True)√,Node Entry) by(fastforce intro:lve-Entry-edge)
    hence CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (NewEntry) [(NewEntry,(λs. True)√,Node Entry)] (Node Entry)
      by(fastforce intro:CFGExit.Cons-path CFGExit.empty-path
              simp:CFGExit.valid-node-def)
    hence slp-edge:CFG.CFG.same-level-path′ src trg knd
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (lift-get-return-edges get-return-edges valid-edge sourcenode
      targetnode kind)
      (NewEntry) [(NewEntry,(λs. True)√,Node Entry)] (Node Entry)
      by(fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def)
    from ‹valid-node m› obtain as where Entry −as→√* m
      by(fastforce dest:Entry-path)
    with ‹inner-node m›
    have ∃ es. CFG.CFG.valid-path′ src trg knd
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (lift-get-return-edges get-return-edges valid-edge sourcenode
      targetnode kind) (Node Entry) es (Node m)
      by(fastforce intro:lift-valid-path simp:inner-node-def)
    then obtain es where CFG.CFG.valid-path′ src trg knd
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (lift-get-return-edges get-return-edges valid-edge sourcenode
      targetnode kind) (Node Entry) es (Node m) by blast
    with slp-edge have CFG.CFG.valid-path′ src trg knd
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (lift-get-return-edges get-return-edges valid-edge sourcenode
        targetnode kind) NewEntry ([(NewEntry,(λs. True)√,Node Entry)]@es)
(Node m)
      by(rule CFGExit.slp-vp-Append)
    with ‹n = Node m› show ?thesis by simp blast
   qed
  qed
```

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

(*Node* (*targetnode ax*))
by(*fastforce intro*:*CFGExit.Cons-path CFGExit.empty-path*
simp:*CFGExit.valid-node-def*)
with ‹*intra-kind* (*kind ax*)›
have *slp-edge*:*CFG.CFG.same-level-path′ src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind)
(*Node Entry*) [(*Node Entry*,*kind ax*,*Node* (*targetnode ax*))]
(*Node* (*targetnode ax*))
by(*fastforce simp*:*CFGExit.slp-def CFGExit.same-level-path-def*
intra-kind-def)
have *targetnode ax* ≠ *Entry*
proof
assume *targetnode ax* = *Entry*
with ‹*valid-edge ax*› **show** *False* **by**(*rule Entry-target*)
qed
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
(*Node Exit*,$(\lambda s.\ True)_{\surd}$,*NewExit*) **by**(*fastforce intro*:*lve-Exit-edge*)
hence *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node Exit*) [(*Node Exit*,$(\lambda s.\ True)_{\surd}$,*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 sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind)
(*Node Exit*) [(*Node Exit*,$(\lambda s.\ True)_{\surd}$,*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* $-asx\rightarrow_{\surd}{}^{*}$ *Exit*
by(*fastforce dest*:*Exit-path*)
with ‹*targetnode ax* ≠ *Entry*›
have ∃ *es*. *CFG.CFG.valid-path′ src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind) (*Node* (*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 sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode 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 sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode*
targetnode kind)
(*Node Entry*) ([(*Node Entry*,*kind ax*,*Node* (*targetnode ax*))]@*es*) (*Node*

Exit)
by(*rule CFGExit.slp-vp-Append*)
with *slp-edge'* **have** *CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*) (*Node Entry*)
(([(*Node Entry*,*kind ax*,*Node* (*targetnode ax*))]@*es*)@
[(*Node Exit*,(λ*s. True*)$_{\surd}$,*NewExit*)]) *NewExit*
by −(*rule CFGExit.vp-slp-Append*)
with ‹*m* = *Entry*› ‹*n* = *Node m*› **show** *?thesis* **by** *simp blast*
next
case *Exit*
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
(*Node Exit*,(λ*s. True*)$_{\surd}$,*NewExit*) **by**(*fastforce intro*:*lve-Exit-edge*)
with ‹*m* = *Exit*› ‹*n* = *Node m*› **have** *CFG.CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
n [(*Node Exit*,(λ*s. True*)$_{\surd}$,*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 sourcenode targetnode kind Entry Exit*
(*Node Exit*,(λ*s. True*)$_{\surd}$,*NewExit*) **by**(*fastforce intro*:*lve-Exit-edge*)
hence *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node Exit*) [(*Node Exit*,(λ*s. True*)$_{\surd}$,*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 sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
(*Node Exit*) [(*Node Exit*,(λ*s. True*)$_{\surd}$,*NewExit*)] *NewExit*
by(*fastforce simp*:*CFGExit.slp-def CFGExit.same-level-path-def*)
from ‹*valid-node m*› **obtain** *as* **where** *m* −*as*→$_{\surd}$* *Exit*
by(*fastforce dest*:*Exit-path*)
with ‹*inner-node m*›
have ∃ *es*. *CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*) (*Node m*) *es* (*Node Exit*)
by(*fastforce intro*:*lift-valid-path simp*:*inner-node-def*)
then obtain *es* **where** *CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*) (*Node m*) *es* (*Node Exit*) **by** *blast*
with *slp-edge* **have** *CFG.CFG.valid-path' src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)

```
      (lift-get-return-edges get-return-edges valid-edge sourcenode
     targetnode kind) (Node m) (es@[(Node Exit,(λs. True)√,NewExit)]) NewExit
      by −(rule CFGExit.vp-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 (src a) = p
```

```
      by −(rule CFGExit.get-proc-return)
    with CFGExit.get-proc-Exit lift-eq ‹n = src a› ‹n' = NewExit›
    have p = Main by simp
    with ‹knd a = Q↩_p f› have knd a = Q↩_Main f by simp
    with ‹lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a›
    have False by(rule CFGExit.Main-no-return-source)
    thus ?thesis by simp
  next
    fix a' Q' f' p'
    assume n' = src a'
      and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
      and knd a' = Q'↩_p' f'
    then obtain x' where valid-edge x' and src a' = Node (sourcenode x')
      and kind x' = Q'↩_p' f'
      by(fastforce elim:lift-valid-edge.cases)
    hence method-exit (sourcenode x') by(fastforce simp:method-exit-def)
    with ‹method-exit (sourcenode x)› lift-eq ‹n = src a› ‹n' = src a'›
      ‹src a = Node (sourcenode x)› ‹src a' = Node (sourcenode x')›
    have sourcenode x = sourcenode x' by(fastforce intro:method-exit-unique)
    with ‹src a = Node (sourcenode x)› ‹src a' = Node (sourcenode x')›
      ‹n = src a› ‹n' = src a'›
    show ?thesis by simp
  qed
 qed
qed


lemma lift-SDG:
 assumes SDG:SDG sourcenode targetnode kind valid-edge Entry get-proc
 get-return-edges procs Main Exit Def Use ParamDefs ParamUses
 and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
 shows SDG 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 SDG sourcenode targetnode kind valid-edge Entry get-proc
   get-return-edges procs Main Exit Def Use ParamDefs ParamUses
   by(rule SDG)
 have wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
   get-return-edges procs Main Exit Def Use ParamDefs ParamUses
   by(unfold-locales)
 have pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
   get-return-edges procs Main Exit
   by(unfold-locales)
 interpret wf':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
by(*fastforce intro*:*lift-CFGExit-wf wf pd*)
interpret *pd'*:*Postdomination src trg knd*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main NewExit
by(*fastforce intro*:*lift-Postdomination wf pd inner*)
show *?thesis* **by**(*unfold-locales*)
qed

3.2.3 Low-deterministic security via the lifted graph

lemma *Lift-NonInterferenceGraph*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *get-proc* **and** *get-return-edges* **and** *procs* **and** *Main*
and *Def* **and** *Use* **and** *ParamDefs* **and** *ParamUses* **and** *H* **and** *L*
defines *lve*:*lve* ≡ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lget-proc*:*lget-proc* ≡ *lift-get-proc get-proc Main*
and *lget-return-edges*:*lget-return-edges* ≡
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
and *lDef*:*lDef* ≡ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* ≡ *lift-Use Use Entry Exit H L*
and *lParamDefs*:*lParamDefs* ≡ *lift-ParamDefs ParamDefs*
and *lParamUses*:*lParamUses* ≡ *lift-ParamUses ParamUses*
assumes *SDG*:*SDG sourcenode targetnode kind valid-edge Entry get-proc*
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and *H* ∩ *L* = {} **and** *H* ∪ *L* = *UNIV*
shows *NonInterferenceInterGraph src trg knd lve NewEntry lget-proc*
lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L
(*Node Entry*) (*Node Exit*)
proof −
interpret *SDG sourcenode targetnode kind valid-edge Entry get-proc*
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(*rule SDG*)
interpret *SDG'*:*SDG src trg knd lve NewEntry lget-proc lget-return-edges*
procs Main NewExit lDef lUse lParamDefs lParamUses
by(*fastforce intro*:*lift-SDG SDG inner simp*:*lve lget-proc lget-return-edges lDef*
lUse lParamDefs lParamUses)
show *?thesis*
proof
fix *a* **assume** *lve a* **and** *src a* = *NewEntry*
thus *trg a* = *NewExit* ∨ *trg a* = *Node Entry*
by(*fastforce elim*:*lift-valid-edge.cases simp*:*lve*)

next
show $\exists\, a.\ lve\ a \wedge src\ a = NewEntry \wedge trg\ a = Node\ Entry \wedge knd\ a = (\lambda s.\ True)_{\surd}$
by(*fastforce intro*:*lve-Entry-edge simp*:*lve*)
next
fix *a* **assume** *lve a* **and** *trg a* = *Node Entry*
from ‹*lve a*›
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
by(*simp add*:*lve*)
from *this* ‹*trg a* = *Node Entry*›
show *src a* = *NewEntry*
proof(*induct rule*:*lift-valid-edge.induct*)
case (*lve-edge a e*)
from ‹*e* = (*Node* (*sourcenode a*), *kind a*, *Node* (*targetnode a*))›
‹*trg e* = *Node Entry*›
have *targetnode a* = *Entry* **by** *simp*
with ‹*valid-edge a*› **have** *False* **by**(*rule Entry-target*)
thus *?case* **by** *simp*
qed *simp-all*
next
fix *a* **assume** *lve a* **and** *trg a* = *NewExit*
thus *src a* = *NewEntry* ∨ *src a* = *Node Exit*
by(*fastforce elim*:*lift-valid-edge.cases simp*:*lve*)
next
show $\exists\, a.\ lve\ a \wedge src\ a = Node\ Exit \wedge trg\ a = NewExit \wedge knd\ a = (\lambda s.\ True)_{\surd}$
by(*fastforce intro*:*lve-Exit-edge simp*:*lve*)
next
fix *a* **assume** *lve a* **and** *src a* = *Node Exit*
from ‹*lve a*›
have *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
by(*simp add*:*lve*)
from *this* ‹*src a* = *Node Exit*›
show *trg a* = *NewExit*
proof(*induct rule*:*lift-valid-edge.induct*)
case (*lve-edge a e*)
from ‹*e* = (*Node* (*sourcenode a*), *kind a*, *Node* (*targetnode a*))›
‹*src e* = *Node Exit*›
have *sourcenode a* = *Exit* **by** *simp*
with ‹*valid-edge a*› **have** *False* **by**(*rule Exit-source*)
thus *?case* **by** *simp*
qed *simp-all*
next
from *lDef* **show** *lDef* (*Node Entry*) = *H*
by(*fastforce elim*:*lift-Def-set.cases intro*:*lift-Def-High*)
next
from *Entry-noteq-Exit lUse* **show** *lUse* (*Node Entry*) = *H*
by(*fastforce elim*:*lift-Use-set.cases intro*:*lift-Use-High*)
next
from *Entry-noteq-Exit lUse* **show** *lUse* (*Node Exit*) = *L*

```
    by(fastforce elim:lift-Use-set.cases intro:lift-Use-Low)
  next
    from ‹H ∩ L = {}› show H ∩ L = {} .
  next
    from ‹H ∪ L = UNIV› show H ∪ L = UNIV .
  qed
qed

end
```

References

[1] G. Barthe and L. P. Nieto. Secure information flow for a concurrent language with scheduling. *Journal of Computer Security*, 15(6):647–689, 2007.

[2] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *ESOP 2007*, volume 4421 of *LNCS*, pages 125–140. Springer, 2007.

[3] L. Beringer and M. Hofmann. Secure information flow and program logics. In *Archive of Formal Proofs*. http://isa-afp.org/entries/SIFPL.shtml, November 2008. Formal proof development.

[4] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[5] F. Kammüller. Formalizing non-interference for a simple bytecode language in Coq. *Formal Aspects of Computing*, 20(3):259–275, 2008.

[6] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order Symbolic Computation*, 14(1):59–91, 2001.

[7] G. Snelting and D. Wasserrab. A correctness proof for the Volpano/Smith security typing system. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/VolpanoSmith.shtml, September 2008. Formal proof development.

[8] D. Wasserrab. Towards certified slicing. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/Slicing.shtml, September 2008. Formal proof development.

[9] D. Wasserrab. Backing up slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley slicer. In *Archive of Formal Proofs*. http://isa-afp.org/entries/HRB-Slicing.shtml, September 2009. Formal proof development.

[10] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based noninterference and its modular proof. In *Proc. of PLAS '09*, pages 31–44. ACM, June 2009.