Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

December 14, 2021

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 Slicing guarantees IFC Noninterference

theory *NonInterferenceIntra* **imports**
  Slicing.Slice
  Slicing.CFGExit-wf
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 [8] accordingly in a new locale:

locale *NonInterferenceIntraGraph* =
BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *Exit* :: *'node* (*'('-Exit'-')*) +
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 *Entry-path-High-path*:
assumes (*-Entry-*) −*as*→∗ *n* **and** *inner-node n*
obtains *a′ as′* **where** *as* = *a′*#*as′* **and** (*-High-*) −*as′*→∗ *n*
and *kind a′* = (λ*s*. *True*)$_{\surd}$
proof(*atomize-elim*)
from ‹(*-Entry-*) −*as*→∗ *n*› ‹*inner-node n*›
show ∃ *a′ as′*. *as* = *a′*#*as′* ∧ (*-High-*) −*as′*→∗ *n* ∧ *kind a′* = (λ*s*. *True*)$_{\surd}$
proof(*induct n′*≡(*-Entry-*) *as n rule*:*path.induct*)
case (*Cons-path n″ as n′ a*)
from ‹*n″* −*as*→∗ *n′*› ‹*inner-node n′*› **have** *n″* ≠ (*-Exit-*)
by(*fastforce simp*:*inner-node-def*)
with ‹*valid-edge a*› ‹*targetnode a* = *n″*› ‹*sourcenode a* = (*-Entry-*)›
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′* = (λ*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*→∗ *n′*› ‹*n″* = (*-High-*)› ‹*kind a′* = (λ*s*. *True*)$_{\surd}$› **show** *?case* **by**
blast
qed *fastforce*
qed

lemma *Exit-path-Low-path*:
assumes *n* −*as*→∗ (*-Exit-*) **and** *inner-node n*
obtains *a′ as′* **where** *as* = *as′*@[*a′*] **and** *n* −*as′*→∗ (*-Low-*)
and *kind a′* = (λ*s*. *True*)$_{\surd}$
proof(*atomize-elim*)
from ‹*n* −*as*→∗ (*-Exit-*)›
show ∃ *as′ a′*. *as* = *as′*@[*a′*] ∧ *n* −*as′*→∗ (*-Low-*) ∧ *kind a′* = (λ*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']→* (-Exit-)›
    have n −as'→* sourcenode a' and valid-edge a' and targetnode a' = (-Exit-)
      by(auto elim:path-split-snoc)
    { assume sourcenode a' = (-Entry-)
      with ‹n −as'→* sourcenode a'› have n = (-Entry-)
        by(blast intro!:path-Entry-target)
      with ‹inner-node n› have False by(simp add:inner-node-def) }
    with ‹valid-edge a'› ‹targetnode a' = (-Exit-)› have sourcenode a' = (-Low-)
      by(blast dest!: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 = (λs. True)√
      by blast
    with ‹valid-edge a'› ‹targetnode a' = (-Exit-)› ‹sourcenode a' = (-Low-)›
    have a' = ax by(fastforce intro:edge-det)
     with ‹n −as'→* sourcenode a'› ‹sourcenode a' = (-Low-)› ‹kind ax = (λs.
True)√›
    show ?case by blast
  qed
qed


lemma not-Low-High: V ∉ L ⟹ V ∈ H
  using HighLowUNIV
  by fastforce

lemma not-High-Low: V ∉ H ⟹ V ∈ L
  using HighLowUNIV
  by fastforce
```

2.2 Low Equivalence

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

```
definition lowEquivalence :: 'state ⇒ 'state ⇒ bool (infixl ≈L 50)
  where s ≈L s' ≡ ∀ V ∈ L. state-val s V = state-val s' V
```

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

```
lemma relevant-vars-Entry:
  assumes V ∈ rv S (-Entry-) and (-High-) ∉ backward-slice S
  shows V ∈ L
proof −
```

```
  from ‹V ∈ rv S (-Entry-)› obtain as n' where (-Entry-) −as→* n'
    and n' ∈ backward-slice S and V ∈ Use n'
    and ∀ nx ∈ set(sourcenodes as). V ∉ Def nx by(erule rvE)
  from ‹(-Entry-) −as→* n'› have valid-node n' by(rule path-valid-node)
  thus ?thesis
  proof(cases n' rule:valid-node-cases)
    case Entry
    with ‹V ∈ Use n'› have False by(simp add:Entry-empty)
    thus ?thesis by simp
  next
    case Exit
    with ‹V ∈ Use n'› have False by(simp add:Exit-empty)
    thus ?thesis by simp
  next
    case inner
    with ‹(-Entry-) −as→* n'› obtain a' as' where as = a'#as'
      and (-High-) −as'→* n' by −(erule Entry-path-High-path)
    from ‹(-Entry-) −as→* n'› ‹as = a'#as'›
    have sourcenode a' = (-Entry-) by(fastforce elim:path.cases)
    show ?thesis
    proof(cases as' = [])
      case True
      with ‹(-High-) −as'→* n'› have n' = (-High-) by fastforce
      with ‹n' ∈ backward-slice S› ‹(-High-) ∉ backward-slice S›
      have False by simp
      thus ?thesis by simp
    next
      case False
      with ‹(-High-) −as'→* n'› have hd (sourcenodes as') = (-High-)
        by(rule path-sourcenode)
      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)
     with ‹hd (sourcenodes as') = (-High-)› ‹∀ nx ∈ set(sourcenodes as). V ∉ Def
nx›
      have V ∉ Def (-High-) by fastforce
      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-) ∉ backward-slice S
  shows ∀ V ∈ rv S (-Entry-). state-val s V = state-val s' V
proof
```

fix V **assume** $V \in rv\ S$ (*-Entry-*)
with ‹(*-High-*) $\notin$ *backward-slice* S› **have** $V \in L$ **by** $-$(*rule relevant-vars-Entry*)
with ‹$s \approx_L s'$› **show** *state-val* $s\ V$ = *state-val* $s'\ V$ **by**(*simp add:lowEquivalence-def*)
qed

lemma *rv-Low-Use-Low*:
assumes (*-Low-*) $\in S$
shows ⟦$n\ -as\rightarrow*$ (*-Low-*); $n\ -as'\rightarrow*$ (*-Low-*);
$\forall\ V \in rv\ S\ n$. *state-val* $s\ V$ = *state-val* $s'\ V$;
preds (*slice-kinds* $S\ as$) s; *preds* (*slice-kinds* $S\ as'$) s'⟧
$\Longrightarrow \forall\ V \in Use$ (*-Low-*). *state-val* (*transfers* (*slice-kinds* $S\ as$) s) V =
state-val (*transfers* (*slice-kinds* $S\ as'$) s') V
proof(*induct n as n*≡(*-Low-*) *arbitrary*:*as' s s' rule*:*path.induct*)
case *empty-path*
{ fix V **assume** $V \in Use$ (*-Low-*)
moreover
from ‹*valid-node* (*-Low-*)› **have** (*-Low-*) $-[]\rightarrow*$ (*-Low-*)
by(*fastforce intro*:*path.empty-path*)
moreover
from ‹*valid-node* (*-Low-*)› ‹(*-Low-*) $\in S$› **have** (*-Low-*) $\in$ *backward-slice* S
by(*fastforce intro*:*refl*)
ultimately have $V \in rv\ S$ (*-Low-*)
by(*fastforce intro*:*rvI simp*:*sourcenodes-def*) **}**
hence $\forall\ V \in Use$ (*-Low-*). $V \in rv\ S$ (*-Low-*) **by** *simp*
show *?case*
proof(*cases* $L = \{\}$)
case *True* **with** *UseLow* **show** *?thesis* **by** *simp*
next
case *False*
from ‹(*-Low-*) $-as'\rightarrow*$ (*-Low-*)› **have** $as' = []$
proof(*induct n*≡(*-Low-*) *as' n'*≡(*-Low-*) *rule*:*path.induct*)
case (*Cons-path n'' as a*)
from ‹*valid-edge a*› ‹*sourcenode a* = (*-Low-*)›
have *targetnode a* = (*-Exit-*) **by** $-$(*rule Exit-successor-of-Low*,*simp+*)
with ‹*targetnode a* = *n''*› ‹*n''* $-as\rightarrow*$ (*-Low-*)›
have (*-Low-*) = (*-Exit-*) **by** $-$(*rule path-Exit-source*,*fastforce*)
with *False* **have** *False* **by** $-$(*drule Low-neq-Exit*,*simp*)
thus *?case* **by** *simp*
qed *simp*
with ‹$\forall\ V \in Use$ (*-Low-*). $V \in rv\ S$ (*-Low-*)›
‹$\forall\ V{\in}rv\ S$ (*-Low-*). *state-val* $s\ V$ = *state-val* $s'\ V$›
show *?thesis* **by**(*auto simp*:*slice-kinds-def*)
qed
next
case (*Cons-path n'' as a n*)
note *IH* = ‹$\bigwedge as'\ s\ s'$. ⟦$n''\ -as'\rightarrow*$ (*-Low-*);
$\forall\ V{\in}rv\ S\ n''$. *state-val* $s\ V$ = *state-val* $s'\ V$;

```
 preds (slice-kinds S as) s; preds (slice-kinds S as') s'⟧
⟹ ∀ V∈Use (-Low-). state-val (transfers (slice-kinds S as) s) V =
                    state-val (transfers (slice-kinds S as') s') V›
show ?case
proof(cases L = {})
  case True with UseLow show ?thesis by simp
next
  case False
  show ?thesis
  proof(cases as')
    case Nil
    with ‹n −as'→* (-Low-)› have n = (-Low-) by fastforce
    with ‹valid-edge a› ‹sourcenode a = n› have targetnode a = (-Exit-)
      by −(rule Exit-successor-of-Low,simp+)
    from Low-source-Exit-edge obtain ax where valid-edge ax
      and sourcenode ax = (-Low-) and targetnode ax = (-Exit-)
      and kind ax = (λs. True)√ by blast
   from ‹valid-edge a› ‹sourcenode a = n› ‹n = (-Low-)› ‹targetnode a = (-Exit-)›
      ‹valid-edge ax› ‹sourcenode ax = (-Low-)› ‹targetnode ax = (-Exit-)›
    have a = ax by(fastforce dest:edge-det)
    with ‹kind ax = (λs. True)√› have kind a = (λs. True)√ by simp
    with ‹targetnode a = (-Exit-)› ‹targetnode a = n''› ‹n'' −as→* (-Low-)›
    have (-Low-) = (-Exit-) by −(rule path-Exit-source,auto)
    with False have False by −(drule Low-neq-Exit,simp)
    thus ?thesis by simp
  next
    case (Cons ax asx)
    with ‹n −as'→* (-Low-)› have n = sourcenode ax and valid-edge ax
      and targetnode ax −asx→* (-Low-) by(auto elim:path-split-Cons)
    show ?thesis
    proof(cases targetnode ax = n'')
      case True
      with ‹targetnode ax −asx→* (-Low-)› have n'' −asx→* (-Low-) by simp
     from ‹valid-edge ax› ‹valid-edge a› ‹n = sourcenode ax› ‹sourcenode a = n›
        True ‹targetnode a = n''› have ax = a by(fastforce intro:edge-det)
      from ‹preds (slice-kinds S (a#as)) s›
      have preds1:preds (slice-kinds S as) (transfer (slice-kind S a) s)
        by(simp add:slice-kinds-def)
      from ‹preds (slice-kinds S as') s'› Cons ‹ax = a›
      have preds2:preds (slice-kinds S asx)
        (transfer (slice-kind S a) s')
        by(simp add:slice-kinds-def)
      from ‹valid-edge a› ‹sourcenode a = n› ‹targetnode a = n''›
        ‹preds (slice-kinds S (a#as)) s› ‹preds (slice-kinds S as') s'›
        ‹ax = a› Cons ‹∀ V∈rv S n. state-val s V = state-val s' V›
      have ∀ V∈rv S n''. 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 IH[OF ‹n'' −asx→* (-Low-)› this preds1 preds2]
```

```
      Cons ‹ax = a› show ?thesis by(simp add:slice-kinds-def)
    next
      case False
      with ‹valid-edge a› ‹valid-edge ax› ‹sourcenode a = n› ‹n = sourcenode ax›
        ‹targetnode a = n''› ‹preds (slice-kinds S (a#as)) s›
        ‹preds (slice-kinds S as') s'› Cons
        ‹∀ V∈rv S n. state-val s V = state-val s' V›
      have False by −(rule rv-branching-edges-slice-kinds-False,auto)
      thus ?thesis by simp
    qed
  qed
 qed
qed
```

2.3 The Correctness Proofs

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

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

```
lemma nonInterference-path-to-Low:
  assumes s ≈_L s' and (-High-) ∉ backward-slice S and (-Low-) ∈ S
  and (-Entry-) −as→* (-Low-) and preds (kinds as) s
  and (-Entry-) −as'→* (-Low-) and preds (kinds as') s'
  shows transfers (kinds as) s ≈_L transfers (kinds as') s'
proof −
  from ‹(-Entry-) −as→* (-Low-)› ‹preds (kinds as) s› ‹(-Low-) ∈ S›
  obtain asx where preds (slice-kinds S asx) s
    and ∀ V ∈ Use (-Low-). state-val(transfers (slice-kinds S asx) s) V =
                           state-val(transfers (kinds as) s) V
    and slice-edges S as = slice-edges S asx
   and (-Entry-) −asx→* (-Low-) by(erule fundamental-property-of-static-slicing)
  from ‹(-Entry-) −as'→* (-Low-)› ‹preds (kinds as') s'› ‹(-Low-) ∈ S›
  obtain asx' where preds (slice-kinds S asx') s'
    and ∀ V ∈ Use (-Low-). state-val (transfers (slice-kinds S asx') s') V =
                           state-val (transfers (kinds as') s') V
    and slice-edges S as' = slice-edges S asx'
   and (-Entry-) −asx'→* (-Low-) by(erule fundamental-property-of-static-slicing)
  from ‹s ≈_L s'› ‹(-High-) ∉ backward-slice S›
  have ∀ V ∈ rv S (-Entry-). state-val s V = state-val s' V
    by(rule lowEquivalence-relevant-nodes-Entry)
  with ‹(-Entry-) −asx→* (-Low-)› ‹(-Entry-) −asx'→* (-Low-)› ‹(-Low-) ∈ S›
    ‹preds (slice-kinds S asx) s› ‹preds (slice-kinds S asx') s'›
  have ∀ V ∈ Use (-Low-). state-val (transfers (slice-kinds S asx) s) V =
                          state-val (transfers (slice-kinds S asx') s') V
    by −(rule rv-Low-Use-Low,auto)
```

```
  with ‹∀ V ∈ Use (-Low-). state-val(transfers (slice-kinds S asx) s) V =
                           state-val(transfers (kinds as) s) V›
    ‹∀ V ∈ Use (-Low-). state-val (transfers (slice-kinds S asx') s') V =
                           state-val (transfers (kinds as') s') V›
  show ?thesis by(auto simp:lowEquivalence-def UseLow)
qed


theorem nonInterference-path:
  assumes s ≈_L s' and (-High-) ∉ backward-slice S and (-Low-) ∈ S
  and (-Entry-) −as→* (-Exit-) and preds (kinds as) s
  and (-Entry-) −as'→* (-Exit-) and preds (kinds as') s'
  shows transfers (kinds as) s ≈_L transfers (kinds as') s'
proof −
  from ‹(-Entry-) −as→* (-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(simp,drule empty-path-nodes,drule Entry-noteq-Exit,simp)
    by(erule path-split-Cons)
  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 −(rule path-Exit-source,simp)
    from Entry-Exit-edge obtain z where valid-edge z
      and sourcenode z = (-Entry-) and targetnode z = (-Exit-)
      and kind z = (λs. False)_√ by blast
    from ‹valid-edge x› ‹valid-edge z› ‹(-Entry-) = sourcenode x›
      ‹sourcenode z = (-Entry-)› Exit ‹targetnode z = (-Exit-)›
    have x = z by(fastforce intro:edge-det)
    with ‹preds (kinds as) s› ‹as = x#xs› ‹xs = []› ‹kind z = (λs. False)_√›
    have False by(simp add:kinds-def)
    thus ?thesis by simp
  qed simp
  with ‹targetnode x −xs→* (-Exit-)› obtain x' xs' where xs = xs'@[x']
    and targetnode x −xs'→* (-Low-) and kind x' = (λs. True)_√
    by(fastforce elim:Exit-path-Low-path)
  with ‹(-Entry-) = sourcenode x› ‹valid-edge x›
  have (-Entry-) −x#xs'→* (-Low-) by(fastforce intro:Cons-path)
  from ‹as = x#xs› ‹xs = xs'@[x']› have as = (x#xs')@[x'] by simp
  with ‹preds (kinds as) s› have preds (kinds (x#xs')) s
    by(simp add:kinds-def preds-split)
  from ‹(-Entry-) −as'→* (-Exit-)› obtain y ys where as' = y#ys
```

```
  and (-Entry-) = sourcenode y and valid-edge y
  and targetnode y −ys→∗ (-Exit-)
  apply(cases as′ = [])
   apply(simp,drule empty-path-nodes,drule Entry-noteq-Exit,simp)
  by(erule path-split-Cons)
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 −(rule path-Exit-source,simp)
  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′) s′› ‹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 ‹as′ = y#ys› ‹ys = ys′@[y′]› have as′ = (y#ys′)@[y′] by simp
with ‹preds (kinds as′) s′› have preds (kinds (y#ys′)) s′
  by(simp add:kinds-def preds-split)
from ‹s ≈L s′› ‹(-High-) ∉ backward-slice S› ‹(-Low-) ∈ S›
  ‹(-Entry-) −x#xs′→∗ (-Low-)› ‹preds (kinds (x#xs′)) s›
  ‹(-Entry-) −y#ys′→∗ (-Low-)› ‹preds (kinds (y#ys′)) s′›
have transfers (kinds (x#xs′)) s ≈L transfers (kinds (y#ys′)) s′
  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 by(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 influ-

enced a low variable and the initial states were low equivalent, the reulting states are again low equivalent:

locale *NonInterferenceIntra* =
NonInterferenceIntraGraph sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice Exit H L High Low +
BackwardSlice-wf sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice sem identifies
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *sem* :: *'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *bool*
(((*1*⟨*-*,*/-*⟩) ⇒*/* (*1*⟨*-*,*/-*⟩)) [*0*,*0*,*0*,*0*] *81*)
and *identifies* :: *'node* ⇒ *'com* ⇒ *bool* (*-* ≜ *-* [*51*, *0*] *80*)
and *Exit* :: *'node* (*'('-Exit'-')*)
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 $s_1 \approx_L s_2$ **and** (*-High-*) ∉ *backward-slice S* **and** (*-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 ⟨*c*,*s*$_1$⟩ ⇒ ⟨*c'*,*s*$_1$'⟩ **and** ⟨*c*,*s*$_2$⟩ ⇒ ⟨*c'*,*s*$_2$'⟩
shows $s_1' \approx_L s_2'$
proof −
from *High-target-Entry-edge* **obtain** *ax* **where** *valid-edge ax*
and *sourcenode ax* = (*-Entry-*) **and** *targetnode ax* = (*-High-*)
and *kind ax* = $(\lambda s.\ True)_{\surd}$ **by** *blast*
from ‹*n* ≜ *c*› ‹⟨*c*,*s*$_1$⟩ ⇒ ⟨*c'*,*s*$_1$'⟩›
obtain n_1 *as*$_1$ **where** *n* −*as*$_1$→∗ n_1 **and** *transfers* (*kinds as*$_1$) $s_1 = s_1'$
and *preds* (*kinds as*$_1$) s_1 **and** n_1 ≜ *c'*
by(*fastforce dest*:*fundamental-property*)
from ‹*n* −*as*$_1$→∗ n_1› ‹*valid-edge a*› ‹*sourcenode a* = (*-High-*)› ‹*targetnode a* = *n*›
have (*-High-*) −*a*#*as*$_1$→∗ n_1 **by**(*rule Cons-path*)
from ‹*final c'*› ‹n_1 ≜ *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-*) $-a\#as_1\rightarrow* n_1$› **have** (*-High-*) $-(a\#as_1)@[a_1]\rightarrow*$ (*-Low-*)
by(*rule path-Append*)
with ‹*valid-edge ax*› ‹*sourcenode ax* = (*-Entry-*)› ‹*targetnode ax* = (*-High-*)›
have (*-Entry-*) $-ax\#((a\#as_1)@[a_1])\rightarrow*$ (*-Low-*) **by** −(*rule Cons-path*)
from ‹*kind ax* = $(\lambda s.\ True)_{\surd}$› ‹*kind a* = $(\lambda s.\ True)_{\surd}$› ‹*preds* (*kinds* as_1) s_1›
‹*kind* $a_1 = \Uparrow id$› **have** *preds* (*kinds* $(ax\#((a\#as_1)@[a_1]))$) s_1
by (*simp add*:*kinds-def preds-split*)
from ‹$n \triangleq c$› ‹$\langle c,s_2\rangle \Rightarrow \langle c',{s_2}'\rangle$›
obtain n_2 as_2 **where** $n -as_2\rightarrow* n_2$ **and** *transfers* (*kinds* as_2) $s_2 = {s_2}'$
and *preds* (*kinds* as_2) s_2 **and** $n_2 \triangleq c'$
by(*fastforce dest*:*fundamental-property*)
from ‹$n -as_2\rightarrow* n_2$› ‹*valid-edge a*› ‹*sourcenode a* = (*-High-*)› ‹*targetnode a* = *n*›
have (*-High-*) $-a\#as_2\rightarrow* n_2$ **by**(*rule Cons-path*)
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-*) $-a\#as_2\rightarrow* n_2$› **have** (*-High-*) $-(a\#as_2)@[a_2]\rightarrow*$ (*-Low-*)
by(*rule path-Append*)
with ‹*valid-edge ax*› ‹*sourcenode ax* = (*-Entry-*)› ‹*targetnode ax* = (*-High-*)›
have (*-Entry-*) $-ax\#((a\#as_2)@[a_2])\rightarrow*$ (*-Low-*) **by** −(*rule Cons-path*)
from ‹*kind ax* = $(\lambda s.\ True)_{\surd}$› ‹*kind a* = $(\lambda s.\ True)_{\surd}$› ‹*preds* (*kinds* as_2) s_2›
‹*kind* $a_2 = \Uparrow id$› **have** *preds* (*kinds* $(ax\#((a\#as_2)@[a_2]))$) s_2
by (*simp add*:*kinds-def preds-split*)
from ‹$s_1 \approx_L s_2$› ‹(*-High-*) $\notin$ *backward-slice S*› ‹(*-Low-*) $\in S$›
‹(*-Entry-*) $-ax\#((a\#as_1)@[a_1])\rightarrow*$ (*-Low-*)› ‹*preds* (*kinds* $(ax\#((a\#as_1)@[a_1]))$ s_1›
‹(*-Entry-*) $-ax\#((a\#as_2)@[a_2])\rightarrow*$ (*-Low-*)› ‹*preds* (*kinds* $(ax\#((a\#as_2)@[a_2]))$ s_2›
have *transfers* (*kinds* $(ax\#((a\#as_1)@[a_1]))$) $s_1 \approx_L$
transfers (*kinds* $(ax\#((a\#as_2)@[a_2]))$) s_2
by(*rule nonInterference-path-to-Low*)
with ‹*kind ax* = $(\lambda s.\ True)_{\surd}$› ‹*kind a* = $(\lambda s.\ True)_{\surd}$› ‹*kind* $a_1 = \Uparrow id$› ‹*kind* a_2 = $\Uparrow id$›
‹*transfers* (*kinds* as_1) $s_1 = {s_1}'$› ‹*transfers* (*kinds* as_2) $s_2 = {s_2}'$›
show *?thesis* **by**(*simp add*:*kinds-def transfers-split*)
qed

end

end

3 Framework Graph Lifting for Noninterference

theory *LiftingIntra*
imports *NonInterferenceIntra Slicing.CDepInstantiations*
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*,*'state*) *LDCFG-edge* =
'node LDCFG-node × (*'state edge-kind*) × *'node LDCFG-node*

3.1.2 Lifting *valid-edge*

inductive *lift-valid-edge* :: (*'edge* ⇒ *bool*) ⇒ (*'edge* ⇒ *'node*) ⇒ (*'edge* ⇒ *'node*) ⇒
(*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'node* ⇒ (*'edge*,*'node*,*'state*) *LDCFG-edge* ⇒
bool
for *valid-edge*::*'edge* ⇒ *bool* **and** *src*::*'edge* ⇒ *'node* **and** *trg*::*'edge* ⇒ *'node*
and *knd*::*'edge* ⇒ *'state 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*)

3.1.3 Lifting *Def* and *Use* sets

inductive-set *lift-Def-set* :: ('*node* ⇒ '*var set*) ⇒ '*node* ⇒ '*node* ⇒
'*var set* ⇒ '*var set* ⇒ ('*node LDCFG-node* × '*var*) *set*
for *Def*::('*node* ⇒ '*var set*) **and** *E*::'*node* **and** *X*::'*node*
and *H*::'*var set* **and** *L*::'*var set*

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

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

abbreviation *lift-Def* :: ('*node* ⇒ '*var set*) ⇒ '*node* ⇒ '*node* ⇒
'*var set* ⇒ '*var set* ⇒ '*node LDCFG-node* ⇒ '*var set*
where *lift-Def Def E X H L n* ≡ {*V*. (*n*,*V*) ∈ *lift-Def-set Def E X H L*}

inductive-set *lift-Use-set* :: ('*node* ⇒ '*var set*) ⇒ '*node* ⇒ '*node* ⇒
'*var set* ⇒ '*var set* ⇒ ('*node LDCFG-node* × '*var*) *set*
for *Use*::'*node* ⇒ '*var set* **and** *E*::'*node* **and** *X*::'*node*
and *H*::'*var set* **and** *L*::'*var set*

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

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

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

abbreviation *lift-Use* :: ('*node* ⇒ '*var set*) ⇒ '*node* ⇒ '*node* ⇒
'*var set* ⇒ '*var set* ⇒ '*node LDCFG-node* ⇒ '*var set*
where *lift-Use Use E X H L n* ≡ {*V*. (*n*,*V*) ∈ *lift-Use-set Use E X H L*}

3.2 The lifting lemmas

3.2.1 Lifting the basic locales

abbreviation *src* :: ('*edge*,'*node*,'*state*) *LDCFG-edge* ⇒ '*node LDCFG-node*
where *src a* ≡ *fst a*

abbreviation *trg* :: (*'edge*,*'node*,*'state*) *LDCFG-edge* ⇒ *'node LDCFG-node*
where *trg a* ≡ *snd*(*snd a*)

definition *knd* :: (*'edge*,*'node*,*'state*) *LDCFG-edge* ⇒ *'state edge-kind*
where *knd a* ≡ *fst*(*snd a*)

lemma *lift-CFG*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFG src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
proof −
interpret *CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
by(*rule wf*)
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
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*)
qed
qed

lemma *lift-CFG-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFG-wf src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val*
proof −
interpret *CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
by(*rule wf*)
interpret *CFG*:*CFG src trg knd*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
by(*fastforce intro*:*lift-CFG wf*)
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 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 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
     thus ?case by(fastforce intro:CFG-edge-no-Def-equal dest:lift-Def-node[of -
Def]
      simp:knd-def)
  qed(auto simp:knd-def)
 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
    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)
      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›
          show ?thesis by(auto dest:deterministic)
        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)√›
  show ?thesis by(fastforce simp:knd-def)
 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))›
  have ∀ V ∈ Def (sourcenode a). state-val (transfer (kind a) s) V =
    state-val (transfer (kind a) s') V
    by −(erule CFG-edge-transfer-uses-only-Use,auto simp:knd-def)
  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 add:knd-def)
 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
```

```
    qed(simp add:knd-def)
  qed
 next
  fix a s s'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and pred (knd a) s
    and ∀ V∈lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
  thus pred (knd a) s'
    by(induct rule:lift-valid-edge.induct,
       auto elim!:CFG-edge-Uses-pred-equal dest:lift-Use-node simp:knd-def)
 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 ∃ 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'›
    show ?case
    proof(induct rule:lift-valid-edge.induct)
      case lve-edge thus ?case by(auto dest:deterministic simp:knd-def)
    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 simp:knd-def)+
 qed
qed


lemma lift-CFGExit:
 assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
                       state-val Exit
 shows CFGExit src trg knd
       (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
       NewEntry NewExit
proof −
 interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
                      state-val Exit
   by(rule wf)
 interpret CFG:CFG src trg knd
```

lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
by(*fastforce intro*:*lift-CFG wf*)
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
from *lve-Entry-Exit-edge*
show ∃ *a*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a* ∧
src a = *NewEntry* ∧ *trg a* = *NewExit* ∧ *knd a* = $(\lambda s.\ False)_{\surd}$
by(*fastforce simp*:*knd-def*)
qed
qed

lemma *lift-CFGExit-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFGExit-wf src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
proof −
interpret *CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
by(*rule wf*)
interpret *CFGExit*:*CFGExit src trg knd*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
NewEntry NewExit
by(*fastforce intro*:*lift-CFGExit wf*)
interpret *CFG-wf*:*CFG-wf src trg knd*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
NewEntry lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val
by(*fastforce intro*:*lift-CFG-wf wf*)
show *?thesis*
proof
show *lift-Def Def Entry Exit H L NewExit* = {} ∧
lift-Use Use Entry Exit H L NewExit = {}
by(*fastforce elim*:*lift-Use-set.cases lift-Def-set.cases*)
qed
qed

3.2.2 Lifting *wod-backward-slice*

lemma *lift-wod-backward-slice*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *Def* **and** *Use* **and** *H* **and** *L*
defines *lve*:*lve* ≡ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lDef*:*lDef* ≡ *lift-Def Def Entry Exit H L*

and *lUse*:*lUse* ≡ *lift-Use Use Entry Exit H L*
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val* (*CFG-wf.wod-backward-slice src trg lve lDef lUse*) *NewExit H L* (*Node Entry*) (*Node Exit*)
proof −
interpret *CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
by(*rule wf*)
interpret *CFGExit-wf*:
CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit
by(*fastforce intro*:*lift-CFGExit-wf wf simp*:*lve lDef lUse*)
from *wf lve* **have** *CFG*:*CFG src trg lve NewEntry*
by(*fastforce intro*:*lift-CFG*)
from *wf lve lDef lUse* **have** *CFG-wf*:*CFG-wf src trg knd lve NewEntry lDef lUse state-val*
by(*fastforce intro*:*lift-CFG-wf*)
show *?thesis*
proof
fix *n S*
assume $n \in$ *CFG-wf.wod-backward-slice src trg lve lDef lUse S*
with *CFG-wf* **show** *CFG.valid-node src trg lve n*
by −(*rule CFG-wf.wod-backward-slice-valid-node*)
next
fix *n S* **assume** *CFG.valid-node src trg lve n* **and** $n \in S$
with *CFG-wf* **show** $n \in$ *CFG-wf.wod-backward-slice src trg lve lDef lUse S*
by −(*rule CFG-wf.refl*)
next
fix n' *S n V*
assume $n' \in$ *CFG-wf.wod-backward-slice src trg lve lDef lUse S*
and *CFG-wf.data-dependence src trg lve lDef lUse n V* n'
with *CFG-wf* **show** $n \in$ *CFG-wf.wod-backward-slice src trg lve lDef lUse S*
by −(*rule CFG-wf.dd-closed*)
next
fix *n S*
from *CFG-wf*
have ($\exists\, m.$ (*CFG.obs src trg lve n* (*CFG-wf.wod-backward-slice src trg lve lDef lUse S*)) = $\{m\}$) ∨ *CFG.obs src trg lve n* (*CFG-wf.wod-backward-slice src trg lve lDef lUse S*) = {}
by(*rule CFG-wf.obs-singleton*)
thus *finite* (*CFG.obs src trg lve n* (*CFG-wf.wod-backward-slice src trg lve lDef lUse S*))
by *fastforce*
next
fix *n S*
from *CFG-wf*

```
    have (∃ m. (CFG.obs src trg lve n
        (CFG-wf.wod-backward-slice src trg lve lDef lUse S)) = {m}) ∨
      CFG.obs src trg lve n (CFG-wf.wod-backward-slice src trg lve lDef lUse S) =
{}
      by(rule CFG-wf.obs-singleton)
    thus card (CFG.obs src trg lve n
                    (CFG-wf.wod-backward-slice src trg lve lDef lUse S)) ≤ 1
      by fastforce
  next
    fix a assume lve a and src a = NewEntry
    with lve show trg a = NewExit ∨ trg a = Node Entry
      by(fastforce elim:lift-valid-edge.cases)
  next
    from lve-Entry-edge lve
    show ∃ a. lve a ∧ src a = NewEntry ∧ trg a = Node Entry ∧ knd a = (λs.
True)√
      by(fastforce simp:knd-def)
  next
    fix a assume lve a and trg a = Node Entry
    with lve show src a = NewEntry by(fastforce elim:lift-valid-edge.cases)
  next
    fix a assume lve a and trg a = NewExit
    with lve show src a = NewEntry ∨ src a = Node Exit
      by(fastforce elim:lift-valid-edge.cases)
  next
    from lve-Exit-edge lve
    show ∃ a. lve a ∧ src a = Node Exit ∧ trg a = NewExit ∧ knd a = (λs. True)√
      by(fastforce simp:knd-def)
  next
    fix a assume lve a and src a = Node Exit
    with lve show trg a = NewExit by(fastforce elim:lift-valid-edge.cases)
  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
```

3.2.3 Lifting *PDG-BS* with *standard-control-dependence*

lemma *lift-Postdomination*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry 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 NewExit*
proof −
interpret *Postdomination sourcenode targetnode kind valid-edge Entry Exit*
by(*rule pd*)
interpret *CFGExit-wf*:*CFGExit-wf src trg knd*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val NewExit
by(*fastforce intro*:*lift-CFGExit-wf wf*)
from *wf* **have** *CFG*:*CFG src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
by(*rule lift-CFG*)
show *?thesis*
proof
fix *n* **assume** *CFG.valid-node src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *n*
show ∃ *as*. *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry as n
proof(*cases n*)
case *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*)
with *NewEntry* **have** *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry [] *n*
by(*fastforce intro*:*CFG.empty-path*[*OF CFG*] *simp*:*CFG.valid-node-def*[*OF CFG*])
thus *?thesis* **by** *blast*
next
case *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*)
with *NewExit* **have** *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry [(*NewEntry*,$(\lambda s.\ False)_{\surd}$,*NewExit*)] *n*
by(*fastforce intro*:*CFG.Cons-path*[*OF CFG*] *CFG.empty-path*[*OF CFG*]
simp:*CFG.valid-node-def*[*OF CFG*])
thus *?thesis* **by** *blast*
next
case (*Node m*)
with *Entry-Exit-edge* ‹*CFG.valid-node src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *n*›

have *valid-node m*
by(*auto elim*:*lift-valid-edge.cases*
simp:*CFG.valid-node-def*[*OF CFG*] *valid-node-def*)
thus *?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 *Entry Node* **have** *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*:*CFG.Cons-path*[*OF CFG*] *CFG.empty-path*[*OF CFG*]
simp:*CFG.valid-node-def*[*OF CFG*])
thus *?thesis* **by** *blast*
next
case *Exit*
from *inner* **obtain** *ax* **where** *valid-edge 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 *path*:*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*:*CFG.Cons-path*[*OF CFG*] *CFG.empty-path*[*OF CFG*]
simp:*CFG.valid-node-def*[*OF CFG*])
have *edge*:*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
(*NewEntry*,($\lambda s.$ *True*)$_{\surd}$,*Node Entry*) **by**(*fastforce intro*:*lve-Entry-edge*)
from ‹*inner-node* (*sourcenode ax*)› **have** *valid-node* (*sourcenode ax*)
by(*rule inner-is-valid*)
then obtain *asx* **where** *Entry* $-asx\rightarrow*$ *sourcenode ax*
by(*fastforce dest*:*Entry-path*)
from *this* ‹*valid-edge ax*› **have** $\exists$ *es. CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node Entry*) *es* (*Node* (*sourcenode ax*))
proof(*induct asx arbitrary*:*ax rule*:*rev-induct*)
case *Nil*
from ‹*Entry* $-[]\rightarrow*$ *sourcenode ax*› **have** *sourcenode ax* = *Entry* **by** *fastforce*
hence *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node Entry*) [] (*Node* (*sourcenode ax*))
apply *simp* **apply**(*rule CFG.empty-path*[*OF CFG*])
by(*auto intro*:*lve-Entry-edge simp*:*CFG.valid-node-def*[*OF CFG*])
thus *?case* **by** *blast*
next
case (*snoc x xs*)
note *IH* = ‹$\bigwedge ax.$ ⟦*Entry* $-xs\rightarrow*$ *sourcenode ax*; *valid-edge ax*⟧ $\Longrightarrow$

```
    ∃es. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) es (Node (sourcenode ax))›
  from ‹Entry −xs@[x]→∗ sourcenode ax›
  have Entry −xs→∗ sourcenode x and valid-edge x
    and targetnode x = sourcenode ax by(auto elim:path-split-snoc)
  { assume targetnode x = Exit
    with ‹valid-edge ax› ‹targetnode x = sourcenode ax›
    have False by −(rule Exit-source,simp+) }
  hence targetnode x ≠ Exit by clarsimp
  with ‹valid-edge x› ‹targetnode x = sourcenode ax›[THEN sym]
  have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (Node (sourcenode x),kind x,Node (sourcenode ax))
    by(fastforce intro:lift-valid-edge.lve-edge)
  hence path:CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node (sourcenode x)) [(Node (sourcenode x),kind x,Node (sourcenode
ax))]
    (Node (sourcenode ax))
    by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
            simp:CFG.valid-node-def[OF CFG])
  from IH[OF ‹Entry −xs→∗ sourcenode x› ‹valid-edge x›] obtain es
    where CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) es (Node (sourcenode x)) by blast
  with path have CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) (es@[(Node (sourcenode x),kind x,Node (sourcenode ax))])
    (Node (sourcenode ax))
    by −(rule CFG.path-Append[OF CFG])
  thus ?case by blast
qed
then obtain es where CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node (sourcenode ax)) by blast
with path have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) (es@ [(Node (sourcenode ax),kind ax,Node Exit)]) (Node
Exit)
  by −(rule CFG.path-Append[OF CFG])
with edge have CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry ((NewEntry,(λs. True)√,Node Entry)#
            (es@ [(Node (sourcenode ax),kind ax,Node Exit)])) (Node Exit)
  by(fastforce intro:CFG.Cons-path[OF CFG])
with Node Exit show ?thesis by fastforce
next
  case inner
  from ‹valid-node m› obtain as where Entry −as→∗ m
```

```
    by(fastforce dest:Entry-path)
  with inner have ∃ es. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) es (Node m)
  proof(induct arbitrary:m rule:rev-induct)
    case Nil
    from ‹Entry −[]→∗ m›
    have m = Entry by fastforce
    with lve-Entry-edge have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Entry) [] (Node m)
     by(fastforce intro:CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF
CFG])
    thus ?case by blast
  next
    case (snoc x xs)
    note IH = ‹⋀m. ⟦inner-node m; Entry −xs→∗ m⟧
      ⟹ ∃ es. CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Entry) es (Node m)›
    from ‹Entry −xs@[x]→∗ m› have Entry −xs→∗ sourcenode x
      and valid-edge x and m = targetnode x by(auto elim:path-split-snoc)
    with ‹inner-node m›
    have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (Node (sourcenode x),kind x,Node m)
      by(fastforce intro:lve-edge simp:inner-node-def)
    hence path:CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node (sourcenode x)) [(Node (sourcenode x),kind x,Node m)] (Node m)
      by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
                   simp:CFG.valid-node-def[OF CFG])
    from ‹valid-edge x› have valid-node (sourcenode x) by simp
    thus ?case
    proof(cases sourcenode x rule:valid-node-cases)
      case Entry
      with edge have CFG.path src trg
        (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
        (Node Entry) [(Node Entry,kind x,Node m)] (Node m)
        apply − apply(rule CFG.Cons-path[OF CFG])
        apply(rule CFG.empty-path[OF CFG])
        by(auto simp:CFG.valid-node-def[OF CFG])
      thus ?thesis by blast
    next
      case Exit
      with ‹valid-edge x› have False by(rule Exit-source)
      thus ?thesis by simp
    next
      case inner
      from IH[OF this ‹Entry −xs→∗ sourcenode x›] obtain es
```

```
          where CFG.path src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          (Node Entry) es (Node (sourcenode x)) by blast
        with path have CFG.path src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          (Node Entry) (es@[(Node (sourcenode x),kind x,Node m)]) (Node m)
          by −(rule CFG.path-Append[OF CFG])
        thus ?thesis by blast
      qed
    qed
    then obtain es where path:CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Entry) es (Node m) by blast
    have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (NewEntry,(λs. True)√,Node Entry) by(fastforce intro:lve-Entry-edge)
    from this path Node have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      NewEntry ((NewEntry,(λs. True)√,Node Entry)#es) n
      by(fastforce intro:CFG.Cons-path[OF CFG])
    thus ?thesis by blast
  qed
 qed
next
  fix n assume CFG.valid-node src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
  show ∃as. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    n as NewExit
  proof(cases n)
    case NewEntry
    have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (NewEntry,(λs. False)√,NewExit) by(fastforce intro:lve-Entry-Exit-edge)
    with NewEntry have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      n [(NewEntry,(λs. False)√,NewExit)] NewExit
      by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
                    simp:CFG.valid-node-def[OF CFG])
    thus ?thesis by blast
  next
    case NewExit
    have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (NewEntry,(λs. False)√,NewExit) by(fastforce intro:lve-Entry-Exit-edge)
    with NewExit have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      n [] NewExit
      by(fastforce intro:CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF
CFG])
    thus ?thesis by blast
  next
```

case (*Node m*)
with *Entry-Exit-edge* ‹*CFG.valid-node src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *n*›
have *valid-node m*
by(*auto elim*:*lift-valid-edge.cases*
simp:*CFG.valid-node-def*[*OF CFG*] *valid-node-def*)
thus *?thesis*
proof(*cases m rule*:*valid-node-cases*)
case *Entry*
from *inner* **obtain** *ax* **where** *valid-edge ax* **and** *inner-node* (*targetnode ax*)
and *sourcenode ax* = *Entry* **by**(*erule inner-node-Entry-edge*)
hence *edge*:*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*)
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 *path*:*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*:*CFG.Cons-path*[*OF CFG*] *CFG.empty-path*[*OF CFG*]
simp:*CFG.valid-node-def*[*OF CFG*])
from ‹*inner-node* (*targetnode ax*)› **have** *valid-node* (*targetnode ax*)
by(*rule inner-is-valid*)
then obtain *asx* **where** *targetnode ax* $-asx\rightarrow*$ *Exit* **by**(*fastforce dest*:*Exit-path*)
from *this* ‹*valid-edge ax*› **have** $\exists$ *es*. *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node* (*targetnode ax*)) *es* (*Node Exit*)
proof(*induct asx arbitrary*:*ax*)
case *Nil*
from ‹*targetnode ax* $-[]\rightarrow*$ *Exit*› **have** *targetnode ax* = *Exit* **by** *fastforce*
hence *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node* (*targetnode ax*)) [] (*Node Exit*)
apply *simp* **apply**(*rule CFG.empty-path*[*OF CFG*])
by(*auto intro*:*lve-Exit-edge simp*:*CFG.valid-node-def*[*OF CFG*])
thus *?case* **by** *blast*
next
case (*Cons x xs*)
note *IH* = ‹$\bigwedge$*ax*. ⟦*targetnode ax* $-xs\rightarrow*$ *Exit*; *valid-edge ax*⟧ $\Longrightarrow$
$\exists$ *es*. *CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
(*Node* (*targetnode ax*)) *es* (*Node Exit*)›
from ‹*targetnode ax* $-x\#xs\rightarrow*$ *Exit*›
have *targetnode x* $-xs\rightarrow*$ *Exit* **and** *valid-edge x*
and *sourcenode x* = *targetnode ax* **by**(*auto elim*:*path-split-Cons*)
{ assume *sourcenode x* = *Entry*
with ‹*valid-edge ax*› ‹*sourcenode x* = *targetnode ax*›
have *False* **by** −(*rule Entry-target*,*simp*+) **}**
hence *sourcenode x* ≠ *Entry* **by** *clarsimp*

```
      with ‹valid-edge x› ‹sourcenode x = targetnode ax›[THEN sym]
      have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
        (Node (targetnode ax),kind x,Node (targetnode x))
        by(fastforce intro:lift-valid-edge.lve-edge)
      from IH[OF ‹targetnode x −xs→∗ Exit› ‹valid-edge x›] obtain es
        where CFG.path src trg
        (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
        (Node (targetnode x)) es (Node Exit) by blast
      with edge have CFG.path src trg
        (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
        (Node (targetnode ax))
        ((Node (targetnode ax),kind x,Node (targetnode x))#es) (Node Exit)
        by(fastforce intro:CFG.Cons-path[OF CFG])
      thus ?case by blast
    qed
    then obtain es where CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node (targetnode ax)) es (Node Exit) by blast
    with edge have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Entry) ((Node Entry, kind ax, Node (targetnode ax))#es) (Node
Exit)
      by(fastforce intro:CFG.Cons-path[OF CFG])
    with path have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Entry) (((Node Entry,kind ax,Node (targetnode ax))#es)@
                    [(Node Exit, (λs. True)√, NewExit)]) NewExit
      by −(rule CFG.path-Append[OF CFG])
    with Node Entry show ?thesis by fastforce
  next
    case Exit
    have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (Node Exit,(λs. True)√,NewExit) by(fastforce intro:lve-Exit-edge)
    with Exit Node have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      n [(Node Exit,(λs. True)√,NewExit)] NewExit
      by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
                    simp:CFG.valid-node-def[OF CFG])
    thus ?thesis by blast
  next
    case inner
    from ‹valid-node m› obtain as where m −as→∗ Exit
      by(fastforce dest:Exit-path)
    with inner have ∃ es. CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node m) es (Node Exit)
    proof(induct as arbitrary:m)
      case Nil
      from ‹m −[]→∗ Exit›
```

```
      have m = Exit by fastforce
      with lve-Exit-edge have CFG.path src trg
        (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
        (Node m) [] (Node Exit)
      by(fastforce intro:CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF
CFG])
      thus ?case by blast
    next
      case (Cons x xs)
      note IH = ‹⋀m. ⟦inner-node m; m −xs→∗ Exit⟧
        ⟹ ∃es. CFG.path src trg
        (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
        (Node m) es (Node Exit)›
      from ‹m −x#xs→∗ Exit› have targetnode x −xs→∗ Exit
        and valid-edge x and m = sourcenode x by(auto elim:path-split-Cons)
      with ‹inner-node m›
      have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
        (Node m,kind x,Node (targetnode x))
        by(fastforce intro:lve-edge simp:inner-node-def)
      from ‹valid-edge x› have valid-node (targetnode x) by simp
      thus ?case
      proof(cases targetnode x rule:valid-node-cases)
        case Entry
        with ‹valid-edge x› have False by(rule Entry-target)
        thus ?thesis by simp
      next
        case Exit
        with edge have CFG.path src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          (Node m) [(Node m,kind x,Node Exit)] (Node Exit)
          apply − apply(rule CFG.Cons-path[OF CFG])
          apply(rule CFG.empty-path[OF CFG])
          by(auto simp:CFG.valid-node-def[OF CFG])
        thus ?thesis by blast
      next
        case inner
        from IH[OF this ‹targetnode x −xs→∗ Exit›] obtain es
          where CFG.path src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          (Node (targetnode x)) es (Node Exit) by blast
        with edge have CFG.path src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          (Node m) ((Node m,kind x,Node (targetnode x))#es) (Node Exit)
          by(fastforce intro:CFG.Cons-path[OF CFG])
        thus ?thesis by blast
      qed
    qed
    then obtain es where path:CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
```

```
      (Node m) es (Node Exit) by blast
    have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (Node Exit,(λs. True)√,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,(λs. True)√,NewExit)] NewExit
      by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
                   simp:CFG.valid-node-def[OF CFG])
    with path Node have CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      n (es@[(Node Exit, (λs. True)√, NewExit)]) NewExit
      by(fastforce intro:CFG.path-Append[OF CFG])
    thus ?thesis by blast
   qed
  qed
 qed
qed


lemma lift-PDG-scd:
 assumes PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
Exit
 (Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)
 and pd:Postdomination sourcenode targetnode kind valid-edge Entry Exit
 and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
 shows PDG src trg knd
 (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
 (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
 (Postdomination.standard-control-dependence src trg
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit)
proof −
 interpret PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
Exit
   Postdomination.standard-control-dependence sourcenode targetnode
                                              valid-edge Exit
   by(rule PDG)
 have wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
                    state-val Exit by(unfold-locales)
 from wf pd inner have pd':Postdomination src trg knd
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   NewEntry NewExit
   by(rule lift-Postdomination)
 from wf have CFG:CFG src trg
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
   by(rule lift-CFG)
 from wf have CFG-wf:CFG-wf src trg knd
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
   (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val
   by(rule lift-CFG-wf)
```

from *wf* **have** *CFGExit*:*CFGExit src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry NewExit
by(*rule lift-CFGExit*)
from *wf* **have** *CFGExit-wf*:*CFGExit-wf src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
by(*rule lift-CFGExit-wf*)
show *?thesis*
proof
fix *a* **assume** *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *trg a* = *NewEntry*
with *CFG* **show** *False* **by**(*rule CFG.Entry-target*)
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′*
with *CFG* **show** *a* = *a′* **by**(*rule CFG.edge-det*)
next
from *CFG-wf*
show *lift-Def Def Entry Exit H L NewEntry* = {} ∧
lift-Use Use Entry Exit H L NewEntry = {}
by(*rule CFG-wf.Entry-empty*)
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** *pred* (*knd a*) *s*
with *CFG-wf* **show** *state-val* (*transfer* (*knd a*) *s*) *V* = *state-val s V*
by(*rule CFG-wf.CFG-edge-no-Def-equal*)
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*
pred (*knd a*) *s pred* (*knd a*) *s′*
with *CFG-wf* **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*
by(*rule CFG-wf.CFG-edge-transfer-uses-only-Use*)
next
fix *a s s′*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *pred* (*knd a*) *s*
and ∀ *V*∈*lift-Use Use Entry Exit H L* (*src a*). *state-val s V* = *state-val s′ V*
with *CFG-wf* **show** *pred* (*knd a*) *s′* **by**(*rule CFG-wf.CFG-edge-Uses-pred-equal*)
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'
   with CFG-wf show ∃ Q Q'. knd a = (Q)√ ∧ knd a' = (Q')√ ∧
                            (∀ s. (Q s ⟶ ¬ Q' s) ∧ (Q' s ⟶ ¬ Q s))
    by(rule CFG-wf.deterministic)
  next
   fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
    and src a = NewExit
   with CFGExit show False by(rule CFGExit.Exit-source)
  next
   from CFGExit
   show ∃ a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
            src a = NewEntry ∧ trg a = NewExit ∧ knd a = (λs. False)√
    by(rule CFGExit.Entry-Exit-edge)
  next
   from CFGExit-wf
   show lift-Def Def Entry Exit H L NewExit = {} ∧
        lift-Use Use Entry Exit H L NewExit = {}
    by(rule CFGExit-wf.Exit-empty)
  next
   fix n n'
   assume scd:Postdomination.standard-control-dependence src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n n'
   show n' ≠ NewExit
   proof(rule ccontr)
    assume ¬ n' ≠ NewExit
    hence n' = NewExit by simp
    with scd pd' show False
     by(fastforce intro:Postdomination.Exit-not-standard-control-dependent)
   qed
  next
   fix n n'
   assume Postdomination.standard-control-dependence src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n n'
   thus ∃ as. CFG.path src trg
            (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
            n as n' ∧ as ≠ []
    by(fastforce simp:Postdomination.standard-control-dependence-def[OF pd'])
  qed
qed
```

```
lemma lift-PDG-standard-backward-slice:
  fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
  and Def and Use and H and L
  defines lve:lve ≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  and lDef:lDef ≡ lift-Def Def Entry Exit H L
  and lUse:lUse ≡ lift-Use Use Entry Exit H L
```

assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit*)
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val*
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*))
NewExit H L (*Node Entry*) (*Node Exit*)
proof −
interpret *PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit
by(*rule PDG*)
have *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit* **by**(*unfold-locales*)
interpret *wf'*:*CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit*
by(*fastforce intro*:*lift-CFGExit-wf wf simp*:*lve lDef lUse*)
from *PDG pd inner lve lDef lUse* **have** *PDG'*:*PDG src trg knd lve NewEntry lDef lUse state-val NewExit*
(*Postdomination.standard-control-dependence src trg lve NewExit*)
by(*fastforce intro*:*lift-PDG-scd*)
from *wf pd inner* **have** *pd'*:*Postdomination src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry NewExit
by(*rule lift-Postdomination*)
from *wf lve* **have** *CFG*:*CFG src trg lve NewEntry*
by(*fastforce intro*:*lift-CFG*)
from *wf lve lDef lUse*
have *CFG-wf*:*CFG-wf src trg knd lve NewEntry lDef lUse state-val*
by(*fastforce intro*:*lift-CFG-wf*)
from *wf lve* **have** *CFGExit*:*CFGExit src trg knd lve NewEntry NewExit*
by(*fastforce intro*:*lift-CFGExit*)
from *wf lve lDef lUse*
have *CFGExit-wf*:*CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit*
by(*fastforce intro*:*lift-CFGExit-wf*)
show *?thesis*
proof
fix *n S*
assume *n* ∈ *PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) *S*
with *PDG'* **show** *CFG.valid-node src trg lve n*
by(*rule PDG.PDG-BS-valid-node*)
next
fix *n S* **assume** *CFG.valid-node src trg lve n* **and** *n* ∈ *S*
thus *n* ∈ *PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) *S*

by(*fastforce intro*:*PDG.PDG-path-Nil*[*OF PDG'*] *simp*:*PDG.PDG-BS-def*[*OF PDG'*])
next
fix n' S n V
assume $n' \in$ *PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) S
and *CFG-wf.data-dependence src trg lve lDef lUse n V n'*
thus $n \in$ *PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) S
by(*fastforce intro*:*PDG.PDG-path-Append*[*OF PDG'*] *PDG.PDG-path-ddep*[*OF PDG'*]
PDG.PDG-ddep-edge[*OF PDG'*] *simp*:*PDG.PDG-BS-def*[*OF PDG'*]
split:*if-split-asm*)
next
fix n S
interpret *PDGx*:*PDG src trg knd lve NewEntry lDef lUse state-val NewExit*
Postdomination.standard-control-dependence src trg lve NewExit
by(*rule PDG'*)
interpret *pdx*:*Postdomination src trg knd lve NewEntry NewExit*
by(*fastforce intro*:*pd' simp*:*lve*)
have *scd*:*StandardControlDependencePDG src trg knd lve NewEntry lDef lUse state-val NewExit* **by**(*unfold-locales*)
from *StandardControlDependencePDG.obs-singleton*[*OF scd*]
have ($\exists\, m.$ *CFG.obs src trg lve n*
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) S) $= \{m\}$)
$\vee$
CFG.obs src trg lve n
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) S) $= \{\}$
by(*fastforce simp*:*StandardControlDependencePDG.PDG-BS-s-def*[*OF scd*])
thus *finite* (*CFG.obs src trg lve n*
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) S))
by *fastforce*
next
fix n S
interpret *PDGx*:*PDG src trg knd lve NewEntry lDef lUse state-val NewExit*
Postdomination.standard-control-dependence src trg lve NewExit
by(*rule PDG'*)
interpret *pdx*:*Postdomination src trg knd lve NewEntry NewExit*
by(*fastforce intro*:*pd' simp*:*lve*)
have *scd*:*StandardControlDependencePDG src trg knd lve NewEntry lDef lUse state-val NewExit* **by**(*unfold-locales*)
from *StandardControlDependencePDG.obs-singleton*[*OF scd*]
have ($\exists\, m.$ *CFG.obs src trg lve n*
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) S) $= \{m\}$)

$\vee$

CFG.obs src trg lve n
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) *S*) = {}
by(*fastforce simp*:*StandardControlDependencePDG.PDG-BS-s-def*[*OF scd*])
thus *card* (*CFG.obs src trg lve n*
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*) *S*)) $\leq$ *1*
by *fastforce*
next
fix *a* **assume** *lve a* **and** *src a* = *NewEntry*
with *lve* **show** *trg a* = *NewExit* $\vee$ *trg a* = *Node Entry*
by(*fastforce elim*:*lift-valid-edge.cases*)
next
from *lve-Entry-edge lve*
show $\exists$ *a. lve a* $\wedge$ *src a* = *NewEntry* $\wedge$ *trg a* = *Node Entry* $\wedge$ *knd a* = $(\lambda s.$ *True*$)_{\surd}$
by(*fastforce simp*:*knd-def*)
next
fix *a* **assume** *lve a* **and** *trg a* = *Node Entry*
with *lve* **show** *src a* = *NewEntry* **by**(*fastforce elim*:*lift-valid-edge.cases*)
next
fix *a* **assume** *lve a* **and** *trg a* = *NewExit*
with *lve* **show** *src a* = *NewEntry* $\vee$ *src a* = *Node Exit*
by(*fastforce elim*:*lift-valid-edge.cases*)
next
from *lve-Exit-edge lve*
show $\exists$ *a. lve a* $\wedge$ *src a* = *Node Exit* $\wedge$ *trg a* = *NewExit* $\wedge$ *knd a* = $(\lambda s.$ *True*$)_{\surd}$
by(*fastforce simp*:*knd-def*)
next
fix *a* **assume** *lve a* **and** *src a* = *Node Exit*
with *lve* **show** *trg a* = *NewExit* **by**(*fastforce elim*:*lift-valid-edge.cases*)
next
from *lDef* **show** *lDef* (*Node Entry*) = *H*
by(*fastforce elim*:*lift-Def-set.cases intro*:*lift-Def-High*)
next
from *Entry-noteq-Exit lUse* **show** *lUse* (*Node Entry*) = *H*
by(*fastforce elim*:*lift-Use-set.cases intro*:*lift-Use-High*)
next
from *Entry-noteq-Exit lUse* **show** *lUse* (*Node Exit*) = *L*
by(*fastforce elim*:*lift-Use-set.cases intro*:*lift-Use-Low*)
next
from ‹*H* $\cap$ *L* = {}› **show** *H* $\cap$ *L* = {} **.**
next
from ‹*H* $\cup$ *L* = *UNIV*› **show** *H* $\cup$ *L* = *UNIV* **.**
qed
qed

3.2.4 Lifting *PDG-BS* with *weak-control-dependence*

lemma *lift-StrongPostdomination*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
and *spd*:*StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *StrongPostdomination src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry NewExit*
proof −
interpret *StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
by(*rule spd*)
have *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry Exit*
by(*unfold-locales*)
interpret *pd'*:*Postdomination src trg knd*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
NewEntry NewExit
by(*fastforce intro*:*wf inner lift-Postdomination pd*)
interpret *CFGExit-wf*:*CFGExit-wf src trg knd*
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val NewExit
by(*fastforce intro*:*lift-CFGExit-wf wf*)
from *wf* **have** *CFG*:*CFG src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
by(*rule lift-CFG*)
show *?thesis*
proof
fix *n* **assume** *CFG.valid-node src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *n*
show *finite*
{*n'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧
src a' = *n* ∧ *trg a'* = *n'*}
proof(*cases n*)
case *NewEntry*
hence {*n'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *n* ∧ *trg a'* = *n'*} = {*NewExit*,*Node Entry*}
by(*auto elim*:*lift-valid-edge.cases intro*:*lift-valid-edge.intros*)
thus *?thesis* **by** *simp*
next
case *NewExit*
hence {*n'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *n* ∧ *trg a'* = *n'*} = {}
by *fastforce*
thus *?thesis* **by** *simp*
next
case (*Node m*)
with *Entry-Exit-edge* ‹*CFG.valid-node src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *n*›
have *valid-node m*
by(*auto elim*:*lift-valid-edge.cases*

simp:*CFG.valid-node-def*[*OF CFG*] *valid-node-def*)
hence *finite* {*m'*. ∃ *a'*. *valid-edge a'* ∧ *sourcenode a'* = *m* ∧ *targetnode a'* = *m'*}
by(*rule successor-set-finite*)
have {*m'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *Node m* ∧ *trg a'* = *Node m'*} ⊆ {*m'*. ∃ *a'*. *valid-edge a'* ∧ *sourcenode a'* = *m* ∧ *targetnode a'* = *m'*}
by(*fastforce elim*:*lift-valid-edge.cases*)
with ‹*finite* {*m'*. ∃ *a'*. *valid-edge a'* ∧ *sourcenode a'* = *m* ∧ *targetnode a'* = *m'*}›
have *finite* {*m'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *Node m* ∧ *trg a'* = *Node m'*}
by −(*rule finite-subset*)
hence *finite* (*Node* ‘ {*m'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *Node m* ∧ *trg a'* = *Node m'*})
by *fastforce*
hence *fin*:*finite* ((*Node* ‘ {*m'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *Node m* ∧ *trg a'* = *Node m'*}) ∪ {*NewEntry*,*NewExit*}) **by** *fastforce*
with *Node* **have** {*n'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *n* ∧ *trg a'* = *n'*} ⊆ (*Node* ‘ {*m'*. ∃ *a'*. *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'* ∧ *src a'* = *Node m* ∧ *trg a'* = *Node m'*}) ∪ {*NewEntry*,*NewExit*} **by** *auto* (*case-tac x*,*auto*)
with *fin* **show** *?thesis* **by** −(*rule finite-subset*)
qed
qed
qed

lemma *lift-PDG-wcd*:
assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*StrongPostdomination.weak-control-dependence sourcenode targetnode valid-edge Exit*)
and *spd*:*StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *PDG src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
(*StrongPostdomination.weak-control-dependence src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewExit*)
proof −
interpret *PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*

StrongPostdomination.weak-control-dependence sourcenode targetnode valid-edge Exit
by(*rule PDG*)
have *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit* **by**(*unfold-locales*)
from *wf spd inner* **have** *spd'*:*StrongPostdomination src trg knd* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry NewExit*
by(*rule lift-StrongPostdomination*)
from *wf* **have** *CFG*:*CFG src trg* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
by(*rule lift-CFG*)
from *wf* **have** *CFG-wf*:*CFG-wf src trg knd* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry* (*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val*
by(*rule lift-CFG-wf*)
from *wf* **have** *CFGExit*:*CFGExit src trg knd* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry NewExit*
by(*rule lift-CFGExit*)
from *wf* **have** *CFGExit-wf*:*CFGExit-wf src trg knd* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry* (*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
by(*rule lift-CFGExit-wf*)
show *?thesis*
proof
fix *a* **assume** *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a* **and** *trg a* = *NewEntry*
with *CFG* **show** *False* **by**(*rule CFG.Entry-target*)
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'*
with *CFG* **show** *a* = *a'* **by**(*rule CFG.edge-det*)
next
from *CFG-wf*
show *lift-Def Def Entry Exit H L NewEntry* = {} ∧ *lift-Use Use Entry Exit H L NewEntry* = {}
by(*rule CFG-wf.Entry-empty*)
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** *pred* (*knd a*) *s*
with *CFG-wf* **show** *state-val* (*transfer* (*knd a*) *s*) *V* = *state-val s V*
by(*rule CFG-wf.CFG-edge-no-Def-equal*)
next
fix *a s s'*
assume *assms*:*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*

a

$\forall\, V{\in}$*lift-Use Use Entry Exit H L* (*src a*). *state-val s V* = *state-val s′ V*
pred (*knd a*) *s pred* (*knd a*) *s′*
with *CFG-wf* **show** $\forall\, V{\in}$*lift-Def Def Entry Exit H L* (*src a*).
state-val (*transfer* (*knd a*) *s*) *V* = *state-val* (*transfer* (*knd a*) *s′*) *V*
by(*rule CFG-wf.CFG-edge-transfer-uses-only-Use*)
next
fix *a s s′*
assume *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *pred* (*knd a*) *s*
and $\forall\, V{\in}$*lift-Use Use Entry Exit H L* (*src a*). *state-val s V* = *state-val s′ V*
with *CFG-wf* **show** *pred* (*knd a*) *s′* **by**(*rule CFG-wf.CFG-edge-Uses-pred-equal*)
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′*
with *CFG-wf* **show** $\exists\, Q\ Q'.\ knd\ a = (Q)_{\surd} \wedge knd\ a' = (Q')_{\surd} \wedge$
$(\forall\, s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$
by(*rule CFG-wf.deterministic*)
next
fix *a* **assume** *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a*
and *src a* = *NewExit*
with *CFGExit* **show** *False* **by**(*rule CFGExit.Exit-source*)
next
from *CFGExit*
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(*rule CFGExit.Entry-Exit-edge*)
next
from *CFGExit-wf*
show *lift-Def Def Entry Exit H L NewExit* = {} $\wedge$
lift-Use Use Entry Exit H L NewExit = {}
by(*rule CFGExit-wf.Exit-empty*)
next
fix *n n′*
assume *wcd*:*StrongPostdomination.weak-control-dependence src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewExit n n′*
show *n′* ≠ *NewExit*
proof(*rule ccontr*)
assume ¬ *n′* ≠ *NewExit*
hence *n′* = *NewExit* **by** *simp*
with *wcd spd′* **show** *False*
by(*fastforce intro*:*StrongPostdomination.Exit-not-weak-control-dependent*)
qed
next
fix *n n′*
assume *StrongPostdomination.weak-control-dependence src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewExit n n′*

thus $\exists$ *as. CFG.path src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
n as n' $\wedge$ *as* $\neq$ []
by(*fastforce simp*:*StrongPostdomination.weak-control-dependence-def*[*OF spd'*])
qed
qed

lemma *lift-PDG-weak-backward-slice*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *Def* **and** *Use* **and** *H* **and** *L*
defines *lve*:*lve* $\equiv$ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lDef*:*lDef* $\equiv$ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* $\equiv$ *lift-Use Use Entry Exit H L*
assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*StrongPostdomination.weak-control-dependence sourcenode targetnode*
valid-edge Exit)
and *spd*:*StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and *H* $\cap$ *L* = {} **and** *H* $\cup$ *L* = *UNIV*
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val*
(*PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*))
NewExit H L (*Node Entry*) (*Node Exit*)
proof −
interpret *PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
StrongPostdomination.weak-control-dependence sourcenode targetnode
valid-edge Exit
by(*rule PDG*)
have *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*
state-val Exit **by**(*unfold-locales*)
interpret *wf'*:*CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit*
by(*fastforce intro*:*lift-CFGExit-wf wf simp*:*lve lDef lUse*)
from *PDG spd inner lve lDef lUse* **have** *PDG'*:*PDG src trg knd*
lve NewEntry lDef lUse state-val NewExit
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*)
by(*fastforce intro*:*lift-PDG-wcd*)
from *wf spd inner* **have** *spd'*:*StrongPostdomination src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry NewExit
by(*rule lift-StrongPostdomination*)
from *wf lve* **have** *CFG*:*CFG src trg lve NewEntry*
by(*fastforce intro*:*lift-CFG*)
from *wf lve lDef lUse*
have *CFG-wf*:*CFG-wf src trg knd lve NewEntry lDef lUse state-val*

by(*fastforce intro*:*lift-CFG-wf*)
from *wf lve* **have** *CFGExit*:*CFGExit src trg knd lve NewEntry NewExit*
by(*fastforce intro*:*lift-CFGExit*)
from *wf lve lDef lUse*
have *CFGExit-wf*:*CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit*
by(*fastforce intro*:*lift-CFGExit-wf*)
show *?thesis*
proof
fix *n S*
assume *n* ∈ *PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*) *S*
with *PDG′* **show** *CFG.valid-node src trg lve n*
by(*rule PDG.PDG-BS-valid-node*)
next
fix *n S* **assume** *CFG.valid-node src trg lve n* **and** *n* ∈ *S*
thus *n* ∈ *PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*) *S*
by(*fastforce intro*:*PDG.PDG-path-Nil*[*OF PDG′*] *simp*:*PDG.PDG-BS-def*[*OF PDG′*])
next
fix *n′ S n V*
assume *n′* ∈ *PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*) *S*
and *CFG-wf.data-dependence src trg lve lDef lUse n V n′*
thus *n* ∈ *PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*) *S*
by(*fastforce intro*:*PDG.PDG-path-Append*[*OF PDG′*] *PDG.PDG-path-ddep*[*OF PDG′*]
PDG.PDG-ddep-edge[*OF PDG′*] *simp*:*PDG.PDG-BS-def*[*OF PDG′*]
split:*if-split-asm*)
next
fix *n S*
interpret *PDGx*:*PDG src trg knd lve NewEntry lDef lUse state-val NewExit*
StrongPostdomination.weak-control-dependence src trg lve NewExit
by(*rule PDG′*)
interpret *spdx*:*StrongPostdomination src trg knd lve NewEntry NewExit*
by(*fastforce intro*:*spd′ simp*:*lve*)
have *wcd*:*WeakControlDependencePDG src trg knd lve NewEntry*
lDef lUse state-val NewExit **by**(*unfold-locales*)
from *WeakControlDependencePDG.obs-singleton*[*OF wcd*]
have (∃ *m*. *CFG.obs src trg lve n*
(*PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*) *S*) = {*m*}) ∨
CFG.obs src trg lve n
(*PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*) *S*) = {}

```
    by(fastforce simp:WeakControlDependencePDG.PDG-BS-w-def[OF wcd])
  thus finite (CFG.obs src trg lve n
     (PDG.PDG-BS src trg lve lDef lUse
      (StrongPostdomination.weak-control-dependence src trg lve NewExit) S))
    by fastforce
 next
  fix n S
  interpret PDGx:PDG src trg knd lve NewEntry lDef lUse state-val NewExit
   StrongPostdomination.weak-control-dependence src trg lve NewExit
    by(rule PDG')
  interpret spdx:StrongPostdomination src trg knd lve NewEntry NewExit
    by(fastforce intro:spd' simp:lve)
  have wcd:WeakControlDependencePDG src trg knd lve NewEntry
   lDef lUse state-val NewExit by(unfold-locales)
  from WeakControlDependencePDG.obs-singleton[OF wcd]
  have (∃ m. CFG.obs src trg lve n
    (PDG.PDG-BS src trg lve lDef lUse
     (StrongPostdomination.weak-control-dependence src trg lve NewExit) S) =
{m}) ∨
    CFG.obs src trg lve n
    (PDG.PDG-BS src trg lve lDef lUse
     (StrongPostdomination.weak-control-dependence src trg lve NewExit) S) =
{}
    by(fastforce simp:WeakControlDependencePDG.PDG-BS-w-def[OF wcd])
  thus card (CFG.obs src trg lve n
    (PDG.PDG-BS src trg lve lDef lUse
     (StrongPostdomination.weak-control-dependence src trg lve NewExit) S)) ≤
1
    by fastforce
 next
  fix a assume lve a and src a = NewEntry
  with lve show trg a = NewExit ∨ trg a = Node Entry
    by(fastforce elim:lift-valid-edge.cases)
 next
  from lve-Entry-edge lve
  show ∃ a. lve a ∧ src a = NewEntry ∧ trg a = Node Entry ∧ knd a = (λs.
True)√
    by(fastforce simp:knd-def)
 next
  fix a assume lve a and trg a = Node Entry
  with lve show src a = NewEntry by(fastforce elim:lift-valid-edge.cases)
 next
  fix a assume lve a and trg a = NewExit
  with lve show src a = NewEntry ∨ src a = Node Exit
    by(fastforce elim:lift-valid-edge.cases)
 next
  from lve-Exit-edge lve
  show ∃ a. lve a ∧ src a = Node Exit ∧ trg a = NewExit ∧ knd a = (λs. True)√
    by(fastforce simp:knd-def)
```

next
fix *a* **assume** *lve a* **and** *src a* = *Node Exit*
with *lve* **show** *trg a* = *NewExit* **by**(*fastforce elim*:*lift-valid-edge.cases*)
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

4 Information Flow for While

theory *NonInterferenceWhile* **imports**
Slicing.SemanticsWellFormed
Slicing.StaticControlDependences
LiftingIntra
begin

locale *SecurityTypes* =
fixes *H* :: *vname set*
fixes *L* :: *vname set*
assumes *HighLowDistinct*: *H* ∩ *L* = {}
and *HighLowUNIV*: *H* ∪ *L* = *UNIV*
begin

4.1 Lifting *labels-nodes* and Defining *final*

fun *labels-LDCFG-nodes* :: *cmd* ⇒ *w-node LDCFG-node* ⇒ *cmd* ⇒ *bool*
where *labels-LDCFG-nodes prog* (*Node n*) *c* = *labels-nodes prog n c*
| *labels-LDCFG-nodes prog n c* = *False*

lemmas *WCFG-path-induct*[*consumes 1*, *case-names empty-path Cons-path*]
= *CFG.path.induct*[*OF While-CFG-aux*]

lemma *lift-valid-node*:
assumes *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
shows *CFG.valid-node src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind* (*-Entry-*) (*-Exit-*))
(*Node n*)
proof −
from ‹*CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*›
obtain *a* **where** *valid-edge prog a* **and** *n* = *sourcenode a* ∨ *n* = *targetnode a*
by(*fastforce simp*:*While-CFG.valid-node-def*)
from ‹*n* = *sourcenode a* ∨ *n* = *targetnode a*›
show *?thesis*
proof
assume *n* = *sourcenode a*
show *?thesis*
proof(*cases sourcenode a* = *Entry*)
case *True*
have *lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind Entry Exit*
(*NewEntry*,(λ*s*. *True*)$_{\surd}$,*Node Entry*)
by(*fastforce intro*:*lve-Entry-edge*)
with *While-CFGExit-wf-aux*[*of prog*] ‹*n* = *sourcenode a*› *True* **show** *?thesis*
by(*fastforce simp*:*CFG.valid-node-def*[*OF lift-CFG*])
next
case *False*
with ‹*valid-edge prog a*› ‹*n* = *sourcenode a* ∨ *n* = *targetnode a*›
have *lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind Entry Exit*
(*Node* (*sourcenode a*),*kind a*,*Node* (*targetnode a*))
by(*fastforce intro*:*lve-edge*)
with *While-CFGExit-wf-aux*[*of prog*] ‹*n* = *sourcenode a*› **show** *?thesis*
by(*fastforce simp*:*CFG.valid-node-def*[*OF lift-CFG*])
qed
next
assume *n* = *targetnode a*
show *?thesis*
proof(*cases targetnode a* = *Exit*)
case *True*
have *lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind Entry Exit*
(*Node Exit*,(λ*s*. *True*)$_{\surd}$,*NewExit*)
by(*fastforce intro*:*lve-Exit-edge*)
with *While-CFGExit-wf-aux*[*of prog*] ‹*n* = *targetnode a*› *True* **show** *?thesis*
by(*fastforce simp*:*CFG.valid-node-def*[*OF lift-CFG*])
next
case *False*
with ‹*valid-edge prog a*› ‹*n* = *sourcenode a* ∨ *n* = *targetnode a*›
have *lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind Entry Exit*
(*Node* (*sourcenode a*),*kind a*,*Node* (*targetnode a*))
by(*fastforce intro*:*lve-edge*)
with *While-CFGExit-wf-aux*[*of prog*] ‹*n* = *targetnode a*› **show** *?thesis*
by(*fastforce simp*:*CFG.valid-node-def*[*OF lift-CFG*])
qed

```
  qed
qed



lemma lifted-CFG-fund-prop:
  assumes labels-LDCFG-nodes prog n c and ⟨c,s⟩ →* ⟨c',s'⟩
  shows ∃ n' as. CFG.path src trg
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
  n as n' ∧ transfers (CFG.kinds knd as) s = s' ∧
  preds (CFG.kinds knd as) s ∧ labels-LDCFG-nodes prog n' c'
proof −
  from ‹labels-LDCFG-nodes prog n c› obtain nx where n = Node nx
    and labels-nodes prog nx c by(cases n) auto
  from ‹labels-nodes prog nx c› ‹⟨c,s⟩ →* ⟨c',s'⟩›
  obtain n' as where prog ⊢ nx −as→* n' and transfers (CFG.kinds kind as) s
= s'
    and preds (CFG.kinds kind as) s and labels-nodes prog n' c'
    by(auto dest:While-semantics-CFG-wf.fundamental-property)
  from ‹labels-nodes prog n' c'› have labels-LDCFG-nodes prog (Node n') c'
    by simp
  from ‹prog ⊢ nx −as→* n'› ‹transfers (CFG.kinds kind as) s = s'›
    ‹preds (CFG.kinds kind as) s› ‹n = Node nx›
    ‹labels-nodes prog nx c› ‹labels-nodes prog n' c'›
  have ∃ es. CFG.path src trg
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
    (Node nx) es (Node n') ∧ transfers (CFG.kinds knd es) s = s' ∧
    preds (CFG.kinds knd es) s
  proof(induct arbitrary:n s c rule:WCFG-path-induct)
    case (empty-path n nx)
    from ‹CFG.valid-node sourcenode targetnode (valid-edge prog) n›
    have valid-node:CFG.valid-node src trg
     (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
      (Node n)
      by(rule lift-valid-node)
    have CFG.kinds knd
      ([]::(w-node LDCFG-node × state edge-kind × w-node LDCFG-node) list) =
[]
      by(simp add:CFG.kinds-def[OF lift-CFG[OF While-CFGExit-wf-aux]])
    with ‹transfers (CFG.kinds kind []) s = s'› ‹preds (CFG.kinds kind []) s›
      valid-node
    show ?case
      by(fastforce intro:CFG.empty-path[OF lift-CFG[OF While-CFGExit-wf-aux]]
                   simp:While-CFG.kinds-def)
  next
    case (Cons-path n'' as n' a nx)
    note IH = ‹⋀n s c. ⟦transfers (CFG.kinds kind as) s = s';
      preds (CFG.kinds kind as) s; n = LDCFG-node.Node n'';
      labels-nodes prog n'' c; labels-nodes prog n' c'⟧
```

```
  ⟹ ∃ es. CFG.path src trg
 (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
  (LDCFG-node.Node n'') es (LDCFG-node.Node n') ∧
  transfers (CFG.kinds knd es) s = s' ∧ preds (CFG.kinds knd es) s›
from ‹transfers (CFG.kinds kind (a # as)) s = s'›
have transfers (CFG.kinds kind as) (transfer (kind a) s) = s'
  by(simp add:While-CFG.kinds-def)
from ‹preds (CFG.kinds kind (a # as)) s›
have preds (CFG.kinds kind as) (transfer (kind a) s)
  and pred (kind a) s by(simp-all add:While-CFG.kinds-def)
show ?case
proof(cases sourcenode a = (-Entry-))
  case True
  with ‹sourcenode a = nx› ‹labels-nodes prog nx c› have False by simp
  thus ?thesis by simp
next
  case False
  with ‹valid-edge prog a›
  have edge:lift-valid-edge (valid-edge prog) sourcenode targetnode kind
    Entry Exit (Node (sourcenode a),kind a,Node (targetnode a))
    by(fastforce intro:lve-edge)
  from ‹prog ⊢ n'' −as→∗ n'›
  have CFG.valid-node sourcenode targetnode (valid-edge prog) n''
    by(rule While-CFG.path-valid-node)
  then obtain c'' where labels-nodes prog n'' c''
  proof(cases rule:While-CFGExit.valid-node-cases)
    case Entry
    with ‹targetnode a = n''› ‹valid-edge prog a› have False by fastforce
    thus ?thesis by simp
  next
    case Exit
    with ‹prog ⊢ n'' −as→∗ n'› have n' = (-Exit-) by fastforce
    with ‹labels-nodes prog n' c'› have False by fastforce
    thus ?thesis by simp
  next
    case inner
    then obtain l'' where [simp]:n'' = (- l'' -) by(cases n'') auto
    with ‹valid-edge prog a› ‹targetnode a = n''› have l'' < #:prog
      by(fastforce intro:WCFG-targetlabel-less-num-nodes simp:valid-edge-def)
    then obtain c'' where labels prog l'' c''
      by(fastforce dest:less-num-inner-nodes-label)
    with that show ?thesis by fastforce
  qed
  from IH[OF ‹transfers (CFG.kinds kind as) (transfer (kind a) s) = s'›
    ‹preds (CFG.kinds kind as) (transfer (kind a) s)› - this
    ‹labels-nodes prog n' c'›]
  obtain es where CFG.path src trg
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
    (-Entry-) (-Exit-)) (LDCFG-node.Node n'') es (LDCFG-node.Node n')
```

```
      and transfers (CFG.kinds knd es) (transfer (kind a) s) = s'
      and preds (CFG.kinds knd es) (transfer (kind a) s) by blast
    with ‹targetnode a = n''› ‹sourcenode a = nx› edge
    have path:CFG.path src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode
      kind (-Entry-) (-Exit-))
        (LDCFG-node.Node nx) ((Node (sourcenode a),kind a,Node (targetnode
a))#es)
      (LDCFG-node.Node n')
     by(fastforce intro:CFG.Cons-path[OF lift-CFG[OF While-CFGExit-wf-aux]])
   from edge have knd (Node (sourcenode a),kind a,Node (targetnode a)) = kind
a
      by(simp add:knd-def)
    with ‹transfers (CFG.kinds knd es) (transfer (kind a) s) = s'›
      ‹preds (CFG.kinds knd es) (transfer (kind a) s)› ‹pred (kind a) s›
    have transfers
      (CFG.kinds knd ((Node (sourcenode a),kind a,Node (targetnode a))#es)) s
= s'
      and preds
      (CFG.kinds knd ((Node (sourcenode a),kind a,Node (targetnode a))#es)) s
      by(auto simp:CFG.kinds-def[OF lift-CFG[OF While-CFGExit-wf-aux]])
    with path show ?thesis by blast
  qed
 qed
 with ‹n = Node nx› ‹labels-LDCFG-nodes prog (Node n') c'›
 show ?thesis by fastforce
qed



fun final :: cmd ⇒ bool
 where final Skip = True
 | final c = False


lemma final-edge:
 labels-nodes prog n Skip ⟹ prog ⊢ n −⇑id→ (-Exit-)
proof(induct prog arbitrary:n)
 case Skip
 from ‹labels-nodes Skip n Skip› have n = (- 0 -)
  by(cases n)(auto elim:labels.cases)
 thus ?case by(fastforce intro:WCFG-Skip)
next
 case (LAss V e)
 from ‹labels-nodes (V:=e) n Skip› have n = (- 1 -)
  by(cases n)(auto elim:labels.cases)
 thus ?case by(fastforce intro:WCFG-LAssSkip)
next
 case (Seq c1 c2)
```

```
  note IH2 = ‹⋀n. labels-nodes c2 n Skip ⟹ c2 ⊢ n −⇑id→ (-Exit-)›
  from ‹labels-nodes (c1;; c2) n Skip› obtain l where n = (- l -)
    and l ≥ #:c1 and labels-nodes c2 (- l − #:c1 -) Skip
    by(cases n)(auto elim:labels.cases)
  from IH2[OF ‹labels-nodes c2 (- l − #:c1 -) Skip›]
  have c2 ⊢ (- l − #:c1 -) −⇑id→ (-Exit-) .
  with ‹l ≥ #:c1› have c1;;c2 ⊢ (- l − #:c1 -) ⊕ #:c1 −⇑id→ (-Exit-) ⊕ #:c1
    by(fastforce intro:WCFG-SeqSecond)
  with ‹n = (- l -)› ‹l ≥ #:c1› show ?case by(simp add:id-def)
next
  case (Cond b c1 c2)
  note IH1 = ‹⋀n. labels-nodes c1 n Skip ⟹ c1 ⊢ n −⇑id→ (-Exit-)›
  note IH2 = ‹⋀n. labels-nodes c2 n Skip ⟹ c2 ⊢ n −⇑id→ (-Exit-)›
  from ‹labels-nodes (if (b) c1 else c2) n Skip›
  obtain l where n = (- l -) and disj:(l ≥ 1 ∧ labels-nodes c1 (- l − 1 -) Skip) ∨
    (l ≥ #:c1 + 1 ∧ labels-nodes c2 (- l − #:c1 − 1 -) Skip)
    by(cases n) (fastforce elim:labels.cases)+
  from disj show ?case
  proof
    assume 1 ≤ l ∧ labels-nodes c1 (- l − 1 -) Skip
    hence 1 ≤ l and labels-nodes c1 (- l − 1 -) Skip by simp-all
    from IH1[OF ‹labels-nodes c1 (- l − 1 -) Skip›]
    have c1 ⊢ (- l − 1 -) −⇑id→ (-Exit-) .
    with ‹1 ≤ l› have if (b) c1 else c2 ⊢ (- l − 1 -) ⊕ 1 −⇑id→ (-Exit-) ⊕ 1
      by(fastforce intro:WCFG-CondThen)
    with ‹n = (- l -)› ‹1 ≤ l› show ?case by(simp add:id-def)
  next
    assume #:c1 + 1 ≤ l ∧ labels-nodes c2 (- l − #:c1 − 1 -) Skip
    hence #:c1 + 1 ≤ l and labels-nodes c2 (- l − #:c1 − 1 -) Skip by simp-all
    from IH2[OF ‹labels-nodes c2 (- l − #:c1 − 1 -) Skip›]
    have c2 ⊢ (- l − #:c1 − 1 -) −⇑id→ (-Exit-) .
    with ‹#:c1 + 1 ≤ l› have if (b) c1 else c2 ⊢ (- l − #:c1 − 1 -) ⊕ (#:c1 + 1)
      −⇑id→ (-Exit-) ⊕ (#:c1 + 1)
      by(fastforce intro:WCFG-CondElse)
    with ‹n = (- l -)› ‹#:c1 + 1 ≤ l› show ?case by(simp add:id-def)
  qed
next
  case (While b c)
  from ‹labels-nodes (while (b) c) n Skip› have n = (- 1 -)
    by(cases n)(auto elim:labels.cases)
  thus ?case by(fastforce intro:WCFG-WhileFalseSkip)
qed
```

4.2 Semantic Non-Interference for Weak Order Dependence

```
lemmas WODNonInterferenceGraph =
  lift-wod-backward-slice[OF While-CFGExit-wf-aux HighLowDistinct HighLowU-
NIV]
```

lemma *WODNonInterference*:
NonInterferenceIntra src trg knd
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
NewEntry (*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *id*
(*CFG-wf.wod-backward-slice src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*))
reds (*labels-LDCFG-nodes prog*)
NewExit H L (*LDCFG-node.Node* (*-Entry-*)) (*LDCFG-node.Node* (*-Exit-*)) *final*
proof −
interpret *NonInterferenceIntraGraph src trg knd*
lift-valid-edge (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*)
NewEntry lift-Def (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*
lift-Use (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L id*
CFG-wf.wod-backward-slice src trg
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*)
NewExit H L LDCFG-node.Node (*-Entry-*) *LDCFG-node.Node* (*-Exit-*)
by(*rule WODNonInterferenceGraph*)
interpret *BackwardSlice-wf src trg knd*
lift-valid-edge (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*)
NewEntry lift-Def (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*
lift-Use (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L id*
CFG-wf.wod-backward-slice src trg
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *reds labels-LDCFG-nodes prog*
proof(*unfold-locales*)
fix *n c s c′ s′*
assume *labels-LDCFG-nodes prog n c* **and** ⟨*c*,*s*⟩ →∗ ⟨*c′*,*s′*⟩
thus ∃ *n′ as. CFG.path src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind* (*-Entry-*) (*-Exit-*))
n as n′ ∧ *transfers* (*CFG.kinds knd as*) *s* = *s′* ∧
preds (*CFG.kinds knd as*) *s* ∧ *labels-LDCFG-nodes prog n′ c′*
by(*rule lifted-CFG-fund-prop*)
qed
show *?thesis*
proof(*unfold-locales*)
fix *c n*

assume *final c* **and** *labels-LDCFG-nodes prog n c*
from ‹*final c*› **have** [*simp*]:*c = Skip* **by**(*cases c*) *auto*
from ‹*labels-LDCFG-nodes prog n c*› **obtain** *nx* **where** [*simp*]:*n = Node nx*
and *labels-nodes prog nx Skip* **by**(*cases n*) *auto*
from ‹*labels-nodes prog nx Skip*› **have** *prog* ⊢ *nx* −⇑*id*→ (*-Exit-*)
by(*rule final-edge*)
then obtain *a* **where** *valid-edge prog a* **and** *sourcenode a = nx*
and *kind a* = ⇑*id* **and** *targetnode a* = (*-Exit-*)
by(*auto simp*:*valid-edge-def*)
with ‹*labels-nodes prog nx Skip*›
show ∃ *a. lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode*
kind (*-Entry-*) (*-Exit-*) *a* ∧
src a = n ∧ *trg a = LDCFG-node.Node* (*-Exit-*) ∧ *knd a* = ⇑*id*
by(*rule-tac x*=(*Node nx*,⇑*id*,*Node* (*-Exit-*)) **in** *exI*)
(*auto intro*!:*lve-edge simp*:*knd-def valid-edge-def*)
qed
qed

4.3 Semantic Non-Interference for Standard Control Dependence

lemma *inner-node-exists*:∃ *n. CFGExit.inner-node sourcenode targetnode*
(*valid-edge prog*) (*-Entry-*) (*-Exit-*) *n*
proof −
have *prog* ⊢ (*-Entry-*) −(λ*s. True*)$_{\surd}$→ (*-0-*) **by**(*rule WCFG-Entry*)
hence *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) (*-0-*)
by(*auto simp*:*While-CFG.valid-node-def valid-edge-def*)
thus *?thesis* **by**(*auto simp*:*While-CFGExit.inner-node-def*)
qed

lemmas *SCDNonInterferenceGraph* =
lift-PDG-standard-backward-slice[*OF WStandardControlDependence.PDG-scd*
WhilePostdomination-aux - HighLowDistinct HighLowUNIV]

lemma *SCDNonInterference*:
NonInterferenceIntra src trg knd
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
NewEntry (*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *id*
(*PDG.PDG-BS src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*Postdomination.standard-control-dependence src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*

```
        (-Entry-) (-Exit-)) NewExit))
    reds (labels-LDCFG-nodes prog)
    NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final
proof −
  from inner-node-exists obtain n where CFGExit.inner-node sourcenode tar-
getnode
    (valid-edge prog) (-Entry-) (-Exit-) n by blast
  then interpret NonInterferenceIntraGraph src trg knd
    lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-)
    NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
    lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
    PDG.PDG-BS src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
        (-Entry-) (-Exit-))
      (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
      (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
      (Postdomination.standard-control-dependence src trg
        (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
          (-Entry-) (-Exit-)) NewExit)
    NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node.Node (-Exit-)
  by(fastforce intro:SCDNonInterferenceGraph)
  interpret BackwardSlice-wf src trg knd
    lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-)
    NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
    lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
    PDG.PDG-BS src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
        (-Entry-) (-Exit-))
      (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
      (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
      (Postdomination.standard-control-dependence src trg
        (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
          (-Entry-) (-Exit-)) NewExit) reds labels-LDCFG-nodes prog
  proof(unfold-locales)
    fix n c s c' s'
    assume labels-LDCFG-nodes prog n c and ⟨c,s⟩ →* ⟨c',s'⟩
    thus ∃ n' as. CFG.path src trg
     (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
      n as n' ∧ transfers (CFG.kinds knd as) s = s' ∧
      preds (CFG.kinds knd as) s ∧ labels-LDCFG-nodes prog n' c'
      by(rule lifted-CFG-fund-prop)
  qed
  show ?thesis
  proof(unfold-locales)
    fix c n
    assume final c and labels-LDCFG-nodes prog n c
    from ‹final c› have [simp]:c = Skip by(cases c) auto
```

```
    from ‹labels-LDCFG-nodes prog n c› obtain nx where [simp]:n = Node nx
      and labels-nodes prog nx Skip by(cases n) auto
    from ‹labels-nodes prog nx Skip› have prog ⊢ nx −⇑id→ (-Exit-)
      by(rule final-edge)
    then obtain a where valid-edge prog a and sourcenode a = nx
      and kind a = ⇑id and targetnode a = (-Exit-)
      by(auto simp:valid-edge-def)
    with ‹labels-nodes prog nx Skip›
    show ∃ a. lift-valid-edge (valid-edge prog) sourcenode targetnode
      kind (-Entry-) (-Exit-) a ∧
      src a = n ∧ trg a = LDCFG-node.Node (-Exit-) ∧ knd a = ⇑id
      by(rule-tac x=(Node nx,⇑id,Node (-Exit-)) in exI)
        (auto intro!:lve-edge simp:knd-def valid-edge-def)
  qed
qed
```

4.4 Semantic Non-Interference for Weak Control Dependence

```
lemmas WCDNonInterferenceGraph =
  lift-PDG-weak-backward-slice[OF WWeakControlDependence.PDG-wcd
  WhileStrongPostdomination-aux - HighLowDistinct HighLowUNIV]


lemma WCDNonInterference:
  NonInterferenceIntra src trg knd
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-))
    NewEntry (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
    (lift-Use (Uses prog) (-Entry-) (-Exit-) H L) id
    (PDG.PDG-BS src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
        (-Entry-) (-Exit-))
      (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
      (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
      (StrongPostdomination.weak-control-dependence src trg
        (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
          (-Entry-) (-Exit-)) NewExit))
    reds (labels-LDCFG-nodes prog)
    NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final
proof −
  from inner-node-exists obtain n where CFGExit.inner-node sourcenode tar-
getnode
    (valid-edge prog) (-Entry-) (-Exit-) n by blast
  then interpret NonInterferenceIntraGraph src trg knd
    lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-)
    NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
    lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
    PDG.PDG-BS src trg
```

```
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-))
    (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
    (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
    (StrongPostdomination.weak-control-dependence src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
        (-Entry-) (-Exit-)) NewExit)
   NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node.Node (-Exit-)
  by(fastforce intro:WCDNonInterferenceGraph)
interpret BackwardSlice-wf src trg knd
  lift-valid-edge (valid-edge prog) sourcenode targetnode kind
    (-Entry-) (-Exit-)
  NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
  lift-Use (Uses prog) (-Entry-) (-Exit-) H L id
  PDG.PDG-BS src trg
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-))
    (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
    (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
    (StrongPostdomination.weak-control-dependence src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
        (-Entry-) (-Exit-)) NewExit) reds labels-LDCFG-nodes prog
proof(unfold-locales)
  fix n c s c' s'
  assume labels-LDCFG-nodes prog n c and ⟨c,s⟩ →* ⟨c',s'⟩
  thus ∃ n' as. CFG.path src trg
   (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
    n as n' ∧ transfers (CFG.kinds knd as) s = s' ∧
    preds (CFG.kinds knd as) s ∧ labels-LDCFG-nodes prog n' c'
    by(rule lifted-CFG-fund-prop)
qed
show ?thesis
proof(unfold-locales)
  fix c n
  assume final c and labels-LDCFG-nodes prog n c
  from ‹final c› have [simp]:c = Skip by(cases c) auto
  from ‹labels-LDCFG-nodes prog n c› obtain nx where [simp]:n = Node nx
    and labels-nodes prog nx Skip by(cases n) auto
  from ‹labels-nodes prog nx Skip› have prog ⊢ nx −⇑id→ (-Exit-)
    by(rule final-edge)
  then obtain a where valid-edge prog a and sourcenode a = nx
    and kind a = ⇑id and targetnode a = (-Exit-)
    by(auto simp:valid-edge-def)
  with ‹labels-nodes prog nx Skip›
  show ∃ a. lift-valid-edge (valid-edge prog) sourcenode targetnode
    kind (-Entry-) (-Exit-) a ∧
    src a = n ∧ trg a = LDCFG-node.Node (-Exit-) ∧ knd a = ⇑id
    by(rule-tac x=(Node nx,⇑id,Node (-Exit-)) in exI)
      (auto intro!:lve-edge simp:knd-def valid-edge-def)
```

qed
qed

end

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.