Slicing Guarantees Information Flow
Noninterference

Daniel Wasserrab
September 19, 2015

Abstract

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

1 Introduction

Information Flow Control (IFC) encompasses algorithms which determine 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/StaticIntra/Slice
  ../Slicing/Basic/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 fulfill one basic property: executing the program in two different initial states that may differ in the values of their \( H \)-variables yields two final states that again only differ in the values of their \( H \)-variables; thus the values of the \( H \)-variables did not influence those of the \( L \)-variables.

Every per-based approach makes certain assumptions: (i) all \( H \)-variables are defined at the beginning of the program, (ii) all \( L \)-variables are observed (or used in our terms) at the end and (iii) every variable is either \( H \) or \( L \). This security label is fixed for a variable and cannot be altered during a program run. Thus, we have to extend the prerequisites of the slicing framework in [8] accordingly in a new locale:

```isar
locale NonInterferenceIntraGraph = 
  BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val 
  backward-slice +
  CFGExit-uf 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 (}(\textquoteleft Entry\textquoteright\textquotesingle)) 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 (}(\textquoteleft Exit\textquoteright\textquotesingle)) +
  fixes \( H \) :: 'var set 
  fixes \( L \) :: 'var set 
  fixes High :: 'node (}(\textquoteleft High\textquoteright\textquotesingle)) 
  fixes Low :: 'node (}(\textquoteleft Low\textquoteright\textquotesingle)) 
  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. \text{True})\sqrt{ } \text{/} 
  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. \text{True})\sqrt{ } \text{/} 
  and Exit-successor-of-Low: 
  [valid-edge a; sourcenode a = (-Low-)] ⇒ targetnode a = (-Exit-) 
  and Def\text{High}: Def (-High-) = \( H \) 
  and Use\text{High}: Use (-High-) = \( H \)
```

2
and $UseLow$: $Use\ (-Low\ ) = L$
and $HighLowDistinct$: $H \cap L = \{\}$
and $HighLowUNIV$: $H \cup L = UNIV$

begin

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

lemma $Entry-path-High-path$:
assumes $(-Entry\ ) \ -as\rightarrow\ n$ and inner-node $n$
obtains $a' \ where \ as = a'\ #\ as' \ and \ (-High\ ) \ -as'\rightarrow\ n$
and $kind\ a' = (λs. True)$ \hfill √
proof(atomize-elim)
from $(-Entry\ ) \ -as\rightarrow\ n$ \inner-node $n$
show $∃ a' as'. as = a'\ #\ as' \land (-High\ ) \ -as'\rightarrow\ n \land kind\ a' = (λs. True)$ √
proof(induct $n\ ≡ (-Entry\ ) \ as\ n$ rule: path.induct)
case Nil
from $n'' \ -as\rightarrow\ n'\ \inner-node\ n\ \\\ \ have\ n'' \neq (-Exit\ )$
by (fastforce simp: inner-node-def)
with $\langle valid-edge\ a\ \\\\ \ \langle targetnode\ a = n''\ \\\\ \source node\ a = (-Entry\ )\ \rangle\ \\\\ have\ n'' = (-High\ )$ by $(-drule\ Entry-edge-Exit-or-High,\ auto)$
from $High-target-Entry-edge$
obtain $a'\ \ where\ valid-edge\ a'\ \and\ targetnode\ a' = (-Entry\ )$
and $source node\ a' = (-High\ )\ \and\ kind\ a' = (λs. True)$ √
by blast
with $\langle valid-edge\ a\ \langle source node\ a = (-Entry\ )\ \\\\ \langle targetnode\ a = n''\ \rangle\ \\\\ have\ a = a'$ by (auto dest: edge-det)
with $\langle n'' \ -as\rightarrow\ n'\ \\\\ (n'' = (-High\ )\ \\\\ \langle kind\ a' = (λs. True)$ √ show ?case by blast
qed fastforce

qed

lemma $Exit-path-Low-path$:
assumes $n \ -as\rightarrow\ (-Exit\ )$ and inner-node $n$
obtains $a' \ where\ as = as'@\langle a\ \rangle\ \and\ n \ -as'\rightarrow\ (-Low\ )$
and $kind\ a' = (λs. True)$ √
proof(atomize-elim)
from $\langle n \ -as\rightarrow\ (-Exit\ )\ \rangle$
show $∃ as\ a'. as = as'@\langle a\ \rangle \land n \ -as'\rightarrow\ (-Low\ ) \land kind\ a' = (λs. True)$ √
proof(induct as rule: rev.induct)
case Nil

with \((\text{inner-node } n)\) show \(?\text{case}\) by fastforce

next

case \((\text{snoc } a' \text{ as}')\)

from \(\langle n \rightarrow \text{as}' \text{ as}' \rangle\) (*\(-\text{Exit}\):)

have \(n \rightarrow \text{as}' \text{ as}'\) \and valid-edge \(a'\) and targetnode \(a' = (-\text{Exit}-)\)

by (auto elim:path-split-snoc)

\{ assume source-node \(a' = (-\text{Entry}-)\)

with \(\langle n \rightarrow \text{as}' \rangle\) have \(n = (-\text{Entry}-)\)

by (blast intro!:path-Entry-target)

with \(\langle \text{inner-node } n \rangle\) have \(\text{False}\) by (simp add:inner-node-def)

\}

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

have \(\text{source-node } a' = (-\text{Low}-)\)

by (blast dest!:Exit-edge-Entry-or-Low) from \(\text{Low-source-Exit-edge}\)

obtain \(ax\) where valid-edge \(ax\) and source-node \(ax = (-\text{Low}-)\) and targetnode \(ax = (-\text{Exit}-)\) and kind \(ax = (\lambda s. \text{True})\)

by blast

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

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

with \(\langle n \rightarrow \text{as}' \rangle\) \langle \text{source-node } a' = (-\text{Low}-)\rangle \langle \text{kind } ax = (\lambda s. \text{True})\rangle

show \(?\text{case}\) by blast

qed

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

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

2.2 Low Equivalence

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

**definition** lowEquivalence :: \('s\text{state} \Rightarrow \text{state} \Rightarrow \text{bool} (\text{infixl} \approx_L 50)\)

where \(s \approx_L s' \equiv \forall V \in L. \text{state-val } s V = \text{state-val } s' V\)

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

**lemma** relevant-vars-Entry:

assumes \(V \in \text{rv } S (-\text{Entry}-)\) and \((-\text{High-}) \notin \text{backward-slice } S\)

shows \(V \in L\)

proof –
from \( V \in rv S \), \((-Entry-)\) obtain as \( n' \) where \((-Entry-) \to^* n' \)
and \( n' \in \text{backward-slice } S \) and \( V \in \text{Use } n' \)
and \( \forall nx \in \text{set(sourcenodes as)}. \ V \notin \text{Def } nx \) by (erule \( \text{rvE} \))
from \((-Entry-) \to^* n' \) have valid-node \( n' \) by (rule \( \text{path-valid-node} \))
thus \(?thesis\)
proof (cases \( n' \) rule: valid-node-cases)
  case \( \text{Entry} \)
  with \( V \in \text{Use } n' \) have False by (simp add: Entry-empty)
  thus \(?thesis\) by simp
next
  case \( \text{Exit} \)
  with \( V \in \text{Use } n' \) have False by (simp add: Exit-empty)
  thus \(?thesis\) by simp
next
  case \( \text{inner} \)
  with \((-Entry-) \to^* n' \) obtain \( a' \) as \( as = a'a'\)
  and \((-High-) \to^* n' \) by -(erule Entry-path-High-path)
  from \((-Entry-) \to^* n' \) \( \langle as = a'a' \rangle \)
  have sourcenode \( a' = (-Entry-) \) by (fastforce elim: path.cases)
  show \(?thesis\)
  proof (cases \( as' = [] \))
    case \( \text{True} \)
    with \((-High-) \to^* n' \) have \( n' = (-High-) \) by fastforce
    with \( n' \in \text{backward-slice } S \) \((-High-) \notin \text{backward-slice } S \)
    have False by simp
    thus \(?thesis\) by simp
  next
    case \( \text{False} \)
    with \((-High-) \to^* n' \) have \( \text{hd (sourcenodes as')} = (-High-) \)
    by (rule \( \text{path-sourcenode} \))
    from \( \text{False} \) have \( \text{hd (sourcenodes as')} \in \text{set (sourcenodes as')} \)
    by (fastforce intro:hd-in-set simp:sourcenodes-def)
    with \( \langle as = a'a' \rangle \) have \( \text{hd (sourcenodes as')} \in \text{set (sourcenodes as)} \)
    by (simp add:sourcenodes-def)
    with \( \langle \text{hd (sourcenodes as')} = (-High-) \rangle \) \( \forall nx \in \text{set (sourcenodes as)}. \ V \notin \text{Def } nx \)
    have \( V \notin \text{Def } (-High-) \) by fastforce
    hence \( V \notin H \) by (simp add:DefHigh)
    thus \(?thesis\) by (rule not-High-Low)
  qed
next
next

lemma \( \text{lowEquivalence-relevant-nodes-Entry} \):
  assumes \( s \approx_L s' \) and \( (-High-) \notin \text{backward-slice } S \)
  shows \( \forall V \in rv S \), \((-Entry-)\), \text{state-val } s \ V = \text{state-val } s' \ V \)
proof
fix V assume V ∈ rv S (-Entry-)
with ⟨(-High-) ∉ backward-slice S⟩ have V ∈ L by -(rule relevant-vars-Entry)
with s ≈ s′ show state-val s V = state-val s′ V by(simp add:lowEquivalence-def)
qed

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

show \(?\text{case}\)
proof \((\text{cases} \, L = \{\})\)
  case True with UseLow show \(?\text{thesis} \, by \, \text{simp}\)
next
  case False
  show \(?\text{thesis}\)
  proof \((\text{cases} \, as')\)
    case Nil
    with \(\langle n - as' \rightarrow* \, (-\text{Low}-) \rangle \, \text{have} \, n = (-\text{Low}-) \, \text{by} \, \text{fastforce}\)
    with \(\langle \text{valid-edge} \, a \, \langle \text{source-node} \, a = n \rangle \, \text{have} \, \text{target-node} \, a = (-\text{Exit}-)\rangle\)
    by \(-\text{(rule} \text{Exit-successor-of-Low}, \text{simp}+)\)
    from Low-source-Exit-edge obtain ax where \(\text{valid-edge} \, ax\)
      and source-node ax = (-Low-) and target-node ax = (-Exit-)
    and kind ax = (\(\lambda s. \, \text{True}\))\, by \text{blast}\)
    from \(\langle \text{valid-edge} \, a \, \langle \text{source-node} \, a = n \rangle \, \langle n = (-\text{Low}-) \rangle \, \langle \text{target-node} \, a = (-\text{Exit}-)\rangle\)
      \(\langle \text{valid-edge} \, ax \, \langle \text{source-node} \, ax = (-\text{Low}-) \rangle \, \langle \text{target-node} \, ax = (-\text{Exit}-)\rangle\)
    have \(a = ax \, \text{by} \text{(fastforce dest:edge-det)}\)
    with \(\langle \text{kind} \, ax = (\lambda s. \, \text{True}) \rangle\, \text{have} \, \text{kind} \, a = (\lambda s. \, \text{True})\, \text{by} \, \text{simp}\)
    with \(\langle \text{target-node} \, a = (-\text{Exit}) \rangle \, \langle \text{target-node} \, a = n'' \rangle \, \langle n'' - as' \rightarrow* (-\text{Low}-)\rangle\)
    have \((-\text{Low}-) = (-\text{Exit}) \, \text{by} \, -(\text{rule} \, \text{path-Exit-source}, \text{auto})\)
    with False have False by \(-\text{(drule} \, \text{Low-neq-Exit}, \text{simp})\)
    thus \(?\text{thesis} \, \text{by} \, \text{simp}\)
next
  case \(\langle \text{Cons} \, ax \, ax \rangle\)
  with \(\langle n - as' \rightarrow* (-\text{Low}-) \rangle \, \text{have} \, n = \text{source-node} \, ax \, \text{and} \, \text{valid-edge} \, ax\)
    and target-node ax = as' \rightarrow* (-\text{Low}-) \, \text{by} \, (\text{auto elim:path-split-Cons})\)
  show \(?\text{thesis}\)
  proof \((\text{cases} \, \text{target-node} \, ax = n'')\)
    case True
    with \(\langle \text{target-node} \, ax = as' \rightarrow* (-\text{Low}-) \rangle \, \text{have} \, n'' - as' \rightarrow* (-\text{Low}-) \, \text{by} \, \text{simp}\)
    from \(\langle \text{valid-edge} \, ax \rangle \, \langle \text{valid-edge} \, a \, \langle n = \text{source-node} \, ax \rangle \, \langle \text{source-node} \, a = n \rangle\)
    \(\text{True} \langle \text{target-node} \, a = n'' \rangle \, \text{have} \, ax = a \, \text{by} \text{(fastforce intro:edge-det)}\)
    from \(\langle \text{preds} \, \langle \text{slice-kinds} \, S \, (a\#as) \rangle \, s \rangle\)
    have \(\text{preds1:preds} \, \langle \text{slice-kinds} \, S \, as \rangle \, \langle \text{transfer} \, \langle \text{slice-kind} \, S \, a \rangle \, s \rangle\)
    \(\text{by} \langle \text{simp add:slice-kinds-def} \rangle\)
    from \(\langle \text{preds} \, \langle \text{slice-kinds} \, S \, as' \rangle \, s' \, \text{Cons} \, \langle ax = a \rangle \rangle\)
    have \(\text{preds2:preds} \, \langle \text{slice-kinds} \, S \, ax \rangle\)
    \(\langle \text{transfer} \, \langle \text{slice-kind} \, S \, a \rangle \, s' \rangle\)
    \(\text{by} \langle \text{simp add:slice-kinds-def} \rangle\)
    from \(\langle \text{valid-edge} \, a \rangle \, \langle \text{source-node} \, a = n \rangle \, \langle \text{target-node} \, a = n'' \rangle\)
    \(\langle \text{preds} \, \langle \text{slice-kinds} \, S \, (a\#as) \rangle \, s \rangle \, \langle \text{preds} \, \langle \text{slice-kinds} \, S \, as' \rangle \, s' \rangle\)
    \(\langle ax = a \rangle \, \text{Cons} \, \langle \forall V \in rv \, S \, n', \, \text{state-val} \, s \, V = \text{state-val} \, s' \, V \rangle\)
    have \(\forall V \in rv \, S \, n'', \, \text{state-val} \, \langle \text{transfer} \, \langle \text{slice-kind} \, S \, a \rangle \, s \rangle \, V = \text{state-val} \, \langle \text{transfer} \, \langle \text{slice-kind} \, S \, a \rangle \, s' \rangle \, V\)
    by \(-\text{(rule rv-edge-slice-kinds,auto)}\)
    from \(IH \langle \text{OF} \, \langle n'' - as' \rightarrow* (-\text{Low}-) \rangle \, \text{this} \, \text{preds1} \, \text{preds2} \rangle\)
Cons \(\langle ax = a \rangle\) show \(?thesis by (simp add: slice-kinds-def)\)

next

case False

with \(\langle valid-edge ax \rangle \langle source-node a = n \rangle \langle n = source-node ax \rangle\)
\(\langle target-node a = n' \rangle \langle preds (slice-kinds S (a#as)) s \rangle\)
\(\langle preds (slice-kinds S as') s' \rangle\) Cons
\(\forall V \in rv S n. \ state-val s V = state-val s' V\)

have \(False by -(rule rv-branching-edges-slice-kinds-False,auto)\)

thus \(?thesis by simp\)

qed

qed

qed

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 \((-Entry-) \langle as\rangle \langle -Exit-\rangle\) paths in the control flow graph (CFG), which agree to a complete program execution:

**lemma nonInterference-path-to-Low:**

assumes \(s \approx_L s'\) and \((-High-) \notin backward-slice S\) and \((-Low-) \in S\)
and \((-Entry-) \langle as\rangle \langle -Low-\rangle\) and \(preds (kinds as) s\)
and \((-Entry-) \langle as\rangle \langle -Low-\rangle\) and \(preds (kinds as') s'\)
shows \(transfers (kinds as) s \approx_L transfers (kinds as') s'\)

**proof**

from \((-Entry-) \langle as\rangle \langle -Low-\rangle\) \(preds (kinds as) s\)

obtain \(as\) where \(preds (slice-kinds S asx) s\)

and \(\forall V \in Use (-Low-). \ state-val (transfers (slice-kinds S asx) s) V = state-val (transfers (kinds as) s) V\)

and \(slice-edges S as = slice-edges S asx\)

and \((-Entry-) \langle as\rangle \langle -Low-\rangle\) by \(erule fundamental-property-of-static-slicing\)

from \((-Entry-) \langle as\rangle \langle -Low-\rangle\) \(preds (kinds as') s'\) \((-Low-) \in S\)

obtain \(as'\) where \(preds (slice-kinds S asx') s'\)

and \(\forall V \in Use (-Low-). \ state-val (transfers (slice-kinds S asx') s') V = state-val (transfers (kinds as') s') V\)

and \(slice-edges S as' = slice-edges S asx'\)

and \((-Entry-) \langle as\rangle \langle -Low-\rangle\) by \(erule fundamental-property-of-static-slicing\)

from \((s \approx_L s' \langle -High-\rangle \notin backward-slice S)\)

have \(\forall V \in rv S (-Entry-). \ state-val s V = state-val s' V\)

by \(rule lowEquivalence-relevant-nodes-Entry\)

with \((-Entry-) \langle as\rangle \langle -Low-\rangle\) \((-Entry-) \langle as\rangle \langle -Low-\rangle\) \((-Low-) \in S\)

\(\langle preds (slice-kinds S asx) s \rangle \langle preds (slice-kinds S asx') s' \rangle\)

have \(\forall V \in Use (-Low-). \ state-val (transfers (slice-kinds S asx) s) V = state-val (transfers (slice-kinds S asx') s') V\)

by \(-rule rv-Low-Use-Low,auto\)
with \( \forall V \in \text{Use} \ (-\text{Low}). \text{state-val}(\text{transfers}(\text{slice-kinds} S \ \text{as} z) \ s) \ V = \text{state-val}(\text{transfers}(\text{kinds} \ \text{as}) \ s) \ V \)
\( \forall V \in \text{Use} \ (-\text{Low}). \text{state-val}(\text{transfers}(\text{slice-kinds} S \ \text{as} z') \ s') \ V = \text{state-val}(\text{transfers}(\text{kinds} \ \text{as}') \ s') \ V \)

show \(?thesis\) by\((\text{auto simp:lowEquivalence-def UseLow})\)

qed

\textbf{theorem} \text{nonInterference-path}:
\textbf{assumes} \( s \approx_L s' \ and \ (-\text{High}) \notin \ \text{backward-slice} \ S \ and \ (-\text{Low}) \in S \)
\textbf{and} \((-\text{Entry}) - as \rightarrow^* (-\text{Exit}) \ and \ \text{preds}(\text{kinds}(\text{as})) \ s \)
\textbf{and} \((-\text{Entry}) - as' \rightarrow^* (-\text{Exit}) \ and \ \text{preds}(\text{kinds}(\text{as}')) \ s' \)
\textbf{shows} \text{transfers}(\text{kinds} \ \text{as}) \ s \approx_L \text{transfers}(\text{kinds} \ \text{as}') \ s' \)

\textbf{proof} –
from \((-\text{Entry}) - as \rightarrow^* (-\text{Exit})) \ obtain \ x \ xs \ where \ as = x \#xs \)
\textbf{and} \((-\text{Entry}) = \text{sourcenode} \ x \ and \ \text{valid-edge} \ x \)
\textbf{and} \text{targetnode} \ x = -xs \rightarrow^* (-\text{Exit}) \)
\textbf{apply}\((\text{cases} \ as = [])\)
\textbf{apply}\((\text{simp, drule empty-path-nodes, drule Entry-noteq-Exit, simp})\)
\textbf{by}\((\text{erule path-split-Cons})\)
from \valid-edge \ x \ have \ \text{valid-node}(\text{targetnode} \ x) \ by \ \text{simp} \)
\textbf{hence} \ ?thesis \ by \ \text{simp} \)

\textbf{next}
\textbf{case} \ Exit
\textbf{with} \valid-edge \ x \ have \ False \ by\((\text{rule Entry-target})\)
\textbf{thus} \ ?thesis \ by \ \text{simp} \)

\textbf{qed simp}

\textbf{with} \valid-node \ x \ -xs \rightarrow^* (-\text{Exit})) \ obtain \ x' \ xs' \ where \ xs = xxs'@\{x'\} \)
\textbf{and} \text{targetnode} \ x = -xs' \rightarrow^* (-\text{Low}) \ and \ \text{kind} \ x' = (\lambda s. \ False) \)
\textbf{by}\((\text{fastforce elim:Exit-path-Low-path})\)
\textbf{with} \((-\text{Entry}) = \text{sourcenode} \ x \ (\text{valid-edge} \ x)\)
\textbf{have} \(-xs' \rightarrow^* (-\text{Low}) \ by\((\text{fastforce intro:Cons-path})\)
\textbf{from} \(as = x \#xs' \ \ (xs = xxs'@\{x'\}) \ have \ as = (x \#xs')@\{x'\} \ by \ \text{simp} \)
\textbf{with} \(\text{preds}(\text{kinds} \ \text{as}) \ s \) \ have \ \text{preds}(\text{kinds} \ (x \#xs')) \ s \)
\textbf{by}\((\text{simp add: kinds-def preds-split})\)
\textbf{from} \(-\text{Entry}) - as \rightarrow^* (-\text{Exit}) \ obtain \ y \ ys \ where \ as' = y \#ys \)
and \((-\text{Entry})\) = source node \(y\) and valid edge \(y\)
and target node \(y - ys\rightarrow (\text{-Exit})\)
apply(cases \(as' = [\])
apply(simp, drule empty-path-nodes, drule Entry-noteq-Exit, simp)
by(erule path-split-Cons)
from \(\text{valid-edge} y\) have valid node (target node \(y\)) by simp
hence inner node (target node \(y\))
proof(cases rule:valid-node-cases)
case Entry
with \(\text{valid-edge} y\) have \(\text{False}\) by(rule Entry-target)
thus \(\text{thesis}\) by simp
next
case Exit
with \(\text{target node} y - ys\rightarrow (\text{-Exit})\) have \(ys = []\)
by -(rule path-Exit-source, simp)
from Entry-Exit-edge obtain \(z\) where \(\text{valid-edge} z\)
and source node \(z = (\text{-Entry})\) and target node \(z = (\text{-Exit})\)
and kind \(z = (\lambda s. \text{False})\) by blast
from \(\text{valid-edge} y\) \((\text{valid-edge} z) (\text{-Entry}) = \text{source node} y\)
(source node \(z = (\text{-Entry})\) Exit (target node \(z = (\text{-Exit})\))
have \(y = z\) by(fastforce intro:edge-det)
with \(\text{preds} (\text{kinds} as') s'\) \((as' = y ys) (ys = [])\) \((\text{kind} z = (\lambda s. \text{False})\) \(\checkmark\))
have \(\text{False}\) by(simp add:kinds-def)
thus \(\text{thesis}\) by simp
qed simp
with \(\text{target node} y - ys\rightarrow (\text{-Exit})\) obtain \(y' ys'\) where \(ys = ys'@y'\)
and target node \(y - ys'\rightarrow (\text{-Low})\) and \(\text{kind} y' = (\lambda s. \text{True})\) \(\checkmark\)
by(fastforce elim:Exit-path-Low-path)
with \((\text{-Entry}) = \text{source node} y\) \((\text{valid-edge} y)\)
have \((\text{-Entry}) - y' ys'\rightarrow (\text{-Low})\) by(fastforce intro:Cons-path)
from \((as' = y' ys) (ys = ys'@y')\) have \(as' = (y' ys')@y'\) by simp
with \(\text{preds} (\text{kinds} as') s'\) have \(\text{preds} (\text{kinds} (y' ys')) s'\)
by(simp add:kinds-def preds-split)
from \((s \approx_L s') (\text{-High}) \notin \text{backward slice} S) (\text{-Low}) \in S\)
\((\text{-Entry}) - x' xs'\rightarrow (\text{-Low})\) \((\text{preds} (\text{kinds} (x' xs')) s)\)
\((\text{-Entry}) - y' ys'\rightarrow (\text{-Low})\) \((\text{preds} (\text{kinds} (y' ys')) s'\)
have transfers \((\text{kinds} (x' xs')) s \approx_L \text{transfers} (\text{kinds} (y' ys')) s'\)
by(rule nonInterference-path-to-Low)
with \((as = x' xs) (xs = xs'@x') (\text{kind} x' = (\lambda s. \text{True})\) \(\checkmark\)
\((as' = y' ys) (ys = ys'@y') (\text{kind} y' = (\lambda s. \text{True})\) \(\checkmark\) 
show \(\text{thesis}\) 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 resulting states are again low equivalent:

\[
\text{locale NonInterferenceIntra} = \\
\text{NonInterferenceIntraGraph sourcenode targetnode kind valid-edge Entry} \\
\text{Def Use state-val backward-slice Exit H L High Low +} \\
\text{BackwardSlice-ref sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice sem identifies} \\
\text{for sourcenode :: 'edge ⇒ 'node and targetnode :: 'edge ⇒ 'node} \\
\text{and kind :: 'edge ⇒ 'state edge-kind and valid-edge :: 'edge ⇒ bool} \\
\text{and Entry :: 'node ('('-Entry'')) and Def :: 'node ⇒ 'var set} \\
\text{and Use :: 'node ⇒ 'var set and state-val :: 'state ⇒ 'var ⇒ 'val} \\
\text{and backward-slice :: 'node set ⇒ 'node set} \\
\text{and sem :: 'com ⇒ 'state ⇒ 'com ⇒ 'state ⇒ bool} \\
\text{(((\langle1,/-\rangle) ⇒ (\langle1,/-\rangle)) 0,0,0 81)} \\
\text{and identifies :: 'node ⇒ 'com ⇒ bool (- ≈ [51, 0] 80)} \\
\text{and Exit :: 'node ('('-Exit''))} \\
\text{and H :: 'var set and L :: 'var set} \\
\text{and High :: 'node ('('-High'')) and Low :: 'node ('('-Low'')) +} \\
\text{fixes final :: 'com ⇒ bool} \\
\text{assumes final-edge-Low: [final c; n ≈ e]} \\
\text{⇒ ∃ a. valid-edge a ∧ sourcenode a = n ∧ targetnode a = (-Low-) ∧ kind a = \uparrow id} \\
\text{begin} \\
\text{The following theorem needs the explicit edge from ('-High-) to n. An} \\
\text{approach using a 'init' predicate for initial statements, being reachable from} \\
\text{('-High-) via a (\lambda s. True), edge, does not work as the same statement could } \\
\text{be identified by several nodes, some initial, some not. E.g., in the program} \\
\text{while (True) Skip};;\text{Skip two nodes identify this initial statement: the initial} \\
\text{node and the node within the loop (because of loop unrolling).} \\
\text{theorem nonInterference:} \\
\text{assumes s1 ≈L s2 and ('-High-) \notin backward-slice S and ('-Low-) \in S} \\
\text{and valid-edge a and sourcenode a = ('-High-) and targetnode a = n} \\
\text{and kind a = (\lambda s. True), and n \approx c and final c'} \\
\text{and \langle c,s_1 \rangle ⇒ \langle c',s_1' \rangle and \langle c,s_2 \rangle ⇒ \langle c',s_2' \rangle} \\
\text{shows s_1' \approxL s_2'} \\
\text{proof –} \\
\text{from High-target-Entry-edge obtain ax where valid-edge ax} \\
\text{and sourcenode ax = ('-Entry-) and targetnode ax = ('-High-)} \\
\text{and kind ax = (\lambda s. True), by blast} \\
\text{from \langle n \approx c; (c,s_1) ⇒ (c',s_1') \rangle} \\
\text{obtain n_1 as_1 where n \rightarrow as_1⇒* n_1 and transfers (kinds as_1) s_1 = s_1'} \\
\text{and predset (kinds as_1) s_1 and n_1 \approx c'} \\
\text{by (fastforce dest: fundamental-property)} \\
\text{from \langle n \rightarrow as_1⇒* n_1 \rangle (valid-edge a) (sourcenode a = ('-High-) \langle targetnode a = n \rangle} \\
\text{have ('-High-) \rightarrow a \# \rightarrow as_1⇒* n_1 by (rule Cons-path)} \\
\text{from \langle final c'; n_3 \approx c' \rangle} \\
\text{obtain a_1 where valid-edge a_1 and sourcenode a_1 = n_1} \]
and targetnode $a_1 = \langle -\text{Low-} \rangle$ and $\text{kind } a_1 = \uparrow \text{id}$ by (fastforce dest: final-edge-Low)

hence $\langle -\text{Low-} \rangle$ by (fastforce intro: path-edge)

with $\langle -\text{High-} \rangle - a\#a_1\rightarrow^* 1_1$ have $\langle -\text{High-} \rangle - (a\#a_1)@[a_1]\rightarrow^* \langle -\text{Low-} \rangle$

by (rule path-Append)

with $\langle \text{valid-edge } ax \rangle$ (source node $ax = \langle -\text{Entry-} \rangle$) (target node $a = \langle -\text{High-} \rangle$

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

from $\langle \text{kind } ax = (\lambda s. \text{True}) \rangle \langle \text{kind } a = (\lambda s. \text{True}) \rangle \langle \text{preds (kinds as}_1 s_1 \rangle$

$\langle \text{kind } a_1 = \uparrow \text{id} \rangle$ have $\langle \text{preds (kinds as}_2 s_2 = s_2' \rangle$

by (simp add: kinds-def preds-split)

from $(n \triangleq c) \langle \langle c, s_2 \rangle \Rightarrow \langle c', s_2' \rangle \rangle$

obtain $n_2 a_2$ where $n - a_2\rightarrow^* n_2$ and transfers (kinds as}_2 s_2 = s_2'$

and $\langle \text{preds (kinds as}_2 s_2 \rangle$ $n_2 \triangleq c'$

by (fastforce dest: fundamental-property)

from $(n - a_2\rightarrow^* n_2) \langle \text{valid-edge } a \rangle$ (source node $a = \langle -\text{Entry-} \rangle$) (target node $a = n$

have $\langle -\text{Entry-} \rangle - a\#a_2\rightarrow^* n_2$ by (rule Cons-path)

from $(\langle \text{final c'} \rangle \langle n_2 \triangleq c' \rangle \langle \langle c', s_2' \rangle \rangle$

obtain $a_2$ where valid-edge $a_2$ and source node $a_2 = n_2$

and target node $a_2 = \langle -\text{Low-} \rangle$ and $\langle \text{kind } a_2 = \uparrow \text{id} \rangle$ by (fastforce dest: final-edge-Low)

hence $n_2 - a_2\rightarrow^* \langle -\text{Low-} \rangle$ by (fastforce intro: path-edge)

with $\langle -\text{High-} \rangle - a\#a_2\rightarrow^* n_2$ have $\langle -\text{High-} \rangle - (a\#a_2)@[a_2]\rightarrow^* \langle -\text{Low-} \rangle$

by (rule path-Append)

with $\langle \text{valid-edge } ax \rangle$ (source node $ax = \langle -\text{Entry-} \rangle$) (target node $a = \langle -\text{High-} \rangle$

have $\langle -\text{Entry-} \rangle - ax\#((a\#a_2)@[a_2])\rightarrow^* \langle -\text{Low-} \rangle$ by -(rule Cons-path)

from $\langle \text{kind } ax = (\lambda s. \text{True}) \rangle \langle \text{kind } a = (\lambda s. \text{True}) \rangle \langle \text{preds (kinds as}_2 s_2 \rangle$

$\langle \text{kind } a_2 = \uparrow \text{id} \rangle$ have $\langle \text{preds (kinds as}_2 s_2 = s_2' \rangle$

by (simp add: kinds-def preds-split)

from $(s_1 \approx_L s_2) \langle \langle -\text{Low-} \rangle \notin \text{backward-slice } S \rangle \langle \langle -\text{Low-} \rangle \in S \rangle$

$\langle \langle \text{Entry-} \rangle - ax\#((a\#a_2)@[a_1])\rightarrow^* \langle -\text{Low-} \rangle \langle \text{preds (kinds ax#((a#as1)@[a_1])) s_1 \rangle$

$\langle \langle \text{Entry-} \rangle - ax\#((a\#a_2)@[a_2])\rightarrow^* \langle -\text{Low-} \rangle \langle \text{preds (kinds ax#((a#as2)@[a_2])) s_2 \rangle$

have transfers (kinds ax#((a#as1)@[a_1])) $s_1 \approx_L$

transfers (kinds ax#((a#as2)@[a_2])) $s_2$

by (rule nonInterference-path-to-Low)

with $\langle \text{kind } ax = (\lambda s. \text{True}) \rangle \langle \text{kind } a = (\lambda s. \text{True}) \rangle \langle \text{kind } a_1 = \uparrow \text{id} \rangle \langle \text{kind } a_2$

$= \uparrow \text{id} \rangle$

transfers (kinds as}_1 s_1 = s_1' \langle \text{transfers (kinds as}_2 s_2 = s_2' \rangle$

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

qed

end

end

3 Framework Graph Lifting for Noninterference

theory LiftingIntra

imports NonInterferenceIntra ../Slicing/StaticIntra/CDepInstantiations

12
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 (try a))] => lift-valid-edge valid-edge src trg knd E X e
  | lve-Entry-edge:
    e = (NewEntry,(λs. True)√,Node E)
    => lift-valid-edge valid-edge src trg knd E X e
  | lve-Exit-edge:
    e = (Node X,(λs. True)√,NewExit)
    => lift-valid-edge valid-edge src trg knd E X e
  | lve-Entry-Exit-edge:
    e = (NewEntry,(λs. False)√,NewExit)
    => lift-valid-edge valid-edge src trg knd E X e

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

\textbf{inductive-set lift-Def-set} :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒ 'var set ⇒ ('node LDCFG-node × 'var) set

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

\textbf{and} H::'var set and L::'var set

where lift-Def-node:

\[ V \in \text{Def } n \implies (\text{Node } n, V) \in \text{lift-Def-set Def E X H L} \]

| lift-Def-High:

\[ V \in H \implies (\text{Node } E, V) \in \text{lift-Def-set Def E X H L} \]

\textbf{abbreviation lift-Def} :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒ 'var set ⇒ ('node LDCFG-node ⇒ 'var set)

\textbf{where} lift-Def Def E X H L n ≡ \{ V. (n, V) \in \text{lift-Def-set Def E X H L} \}

\textbf{inductive-set lift-Use-set} :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒ 'var set ⇒ ('node LDCFG-node × 'var) set

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

\textbf{and} H::'var set and L::'var set

where lift-Use-node:

\[ V \in \text{Use } n \implies (\text{Node } n, V) \in \text{lift-Use-set Use E X H L} \]

| lift-Use-High:

\[ V \in H \implies (\text{Node } E, V) \in \text{lift-Use-set Use E X H L} \]

| lift-Use-Low:

\[ V \in L \implies (\text{Node } X, V) \in \text{lift-Use-set Use E X H L} \]

\textbf{abbreviation lift-Use} :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒ 'var set ⇒ ('node LDCFG-node ⇒ 'var set)

\textbf{where} lift-Use Use E X H L n ≡ \{ V. (n, V) \in \text{lift-Use-set Use E X H L} \}

3.2 The lifting lemmas

3.2.1 Lifting the basic locales

\textbf{abbreviation src} :: ('edge, 'node, 'state) LDCFG-edge ⇒ 'node LDCFG-node

\textbf{where} src a ≡ fst a

\textbf{abbreviation trg} :: ('edge, 'node, 'state) LDCFG-edge ⇒ 'node LDCFG-node

\textbf{where} trg a ≡ snd(snd a)

\textbf{definition knd} :: ('edge, 'node, 'state) LDCFG-edge ⇒ 'state edge-kind

\textbf{where} knd a ≡ fst(snd a)
lemma lift-CFG:
assumes \( \text{wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use} \)
state-val Exit
shows CFG src trg
\((\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \text{ NewEntry}\)
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

lemma lift-CFG-wf:
assumes \( \text{wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use} \)
state-val Exit
shows CFG-wf src trg knd
\((\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \text{ NewEntry} \)
\((\text{lift-Def Def Entry Exit H L}) \text{ (lift-Use Use Entry Exit H L) state-val}\)
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 \notin lift-Def Def Entry Exit H L (src a) and pred (kind a) s
thus state-val (transfer (kind 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 [auf - Def])
simp: kind-def\}
qed (auto simp: kind-def)
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 (kind a) s pred (kind a) s'
show \forall V \in lift-Def Def Entry Exit H L (src a).
  state-val (transfer (kind a) s) V = state-val (transfer (kind a) s') V
proof
fix V assume V \in lift-Def Entry Exit H L (src a)
with assms
show state-val (transfer (kind a) s) V = state-val (transfer (kind a) s') V
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  show \{thesis\}
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' = (\lambda s. False) \\checkmark by blast
have \exists Q. kind a = (Q) \\checkmark
proof (cases targetnode a = Exit)
case True
  with \langle valid-edge a \rangle \langle valid-edge a' \rangle \langle sourcenode a = Entry \rangle
    \langle sourcenode a' = Entry \rangle \langle targetnode a' = Exit \rangle
  have a = a' \by (fastforce dest: edge-det)
  with \langle kind a' = (\lambda s. False) \\checkmark \rangle show \{thesis\} by simp
next
  case False
  with \langle valid-edge a \rangle \langle valid-edge a' \rangle \langle sourcenode a = Entry \rangle
    \langle sourcenode a' = Entry \rangle \langle targetnode a' = Exit \rangle
  show \{thesis\} by (auto dest: deterministic)
qed
from True \langle V \in lift-Def Entry Exit H L (src e) \rangle Entry-empty
\langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle
have V \in H \by (fastforce elim: lift-Def-set.cases)
from True \langle e = (Node (sourcenode a), kind a, Node (targetnode a)) \rangle
  \langle sourcenode a \neq Entry \lor targetnode a \neq Exit \rangle
have \forall V \in H. V \in lift-Use Entry Exit H L (src e)
by \((\text{fastforce intro:lift-Use-High})\) 
with \(\forall V \in \text{lift-Use \ Use Entry Exit H L} \text{ (src e)}\).
state-val s V = state-val s' V \(\forall V \in H\)
have state-val s V = state-val s' V by simp
with \(e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)})\)
\(\exists Q. \text{kind a} = (Q)\)
show \(#\text{thesis}#\ by(\text{fastforce simp:knd-def})
next
\text{case} False \{ fix V' assume V' \(\in\) Use (sourcenode a)
with \(e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)})\)
have V' \(\in\) lift-Use \ Use Entry Exit H L \(\text{src e}\)
by(\text{fastforce intro:lift-Use-node})
\}\nwith \(\forall V \in \text{lift-Use \ Use Entry Exit H L} \text{ (src e)}\).
state-val s V = state-val s' V by fastforce
from \(\text{valid-edge a \ this \ \(}\langle\text{pred (kind e)}\rangle \ s \ \langle\text{pred (kind e)}\rangle \ s'\)
\(e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)})\)
have \(\forall V \in \text{Def (sourcenode a)}\).
state-val (transfer (kind a) s) V = state-val (transfer (kind a) s') V
by \(!\\text{erule CFG-edge-transfer-uses-only-Use,auto simp:knd-def}\)
from \(V \in \text{lift-Def \ Def Entry Exit H L} \text{ (src e)}\) False
\(e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)})\)
have V \(\in\) Def (sourcenode a) by(\text{fastforce elim:lift-Def-set.cases})
with \(\forall V \in \text{Def (sourcenode a)}\).
state-val (transfer (kind a) s) V = state-val (transfer (kind a) s') V
\(e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)})\)
show \(#\text{thesis}#\ by(\text{simp add:knd-def})
qed
next
\text{case} (\text{lve-Entry-edge e})
from \(V \in \text{lift-Def \ Def Entry Exit H L} \text{ (src e)}\)
\(e = (\text{NewEntry, (\lambda s. True)}), \text{Node Entry})\)
have False by(\text{fastforce elim:lift-Def-set.cases})
thus \(#\text{case}#\ by simp
next
\text{case} (\text{lve-Exit-edge e})
from \(V \in \text{lift-Def \ Def Entry Exit H L} \text{ (src e)}\)
\(e = (\text{Node Exit, (\lambda s. True)}), \text{NewExit})\)
have False
by(\text{fastforce elim:lift-Def-set.cases intro!:Entry-noteq-Exit simp:Exit-empty})
thus \(#\text{case}#\ by simp
qed(simp add:knd-def)
qed
next
\text{fix} a s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \(\text{pred} (\text{kind} a) \ s\)
and \(\forall V \in \text{lift-Use} \ \text{Use} \ \text{Exit} \ H \ L \ (\text{src} \ a). \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V\)
thus \(\text{pred} (\text{kind} a) \ s'\)
by (induct rule: lift-valid-edge.induct, auto elim!: \text{CFG-edge-Uses-pred-equal} \ \text{dest}: \text{lift-Use-node} \ \text{simp}: \text{kind-def})

next
fix \(a \ a'\)
assume lift-valid-edge valid-edge source-node target-node kind Entry Exit a
and lift-valid-edge valid-edge source-node target-node kind Entry Exit a'
and src \(a = \text{src} \ a'\) and \(\text{try} a \neq \text{try} a'\)
thus \(\exists Q \ Q'. \ \text{kind} a = (Q) \sqrt{} \land \text{kind} a' = (Q') \sqrt{} \land\)
\((\forall s. (Q \ s \rightarrow \neg Q' \ s) \land (Q' \ s \rightarrow \neg Q \ s))\)
proof (induct rule: lift-valid-edge.induct)
case (lve-edge \(a \ e\))
from \(\langle \text{lift-valid-edge valid-edge source-node target-node kind Entry Exit} \ a' \rangle\)
\(\langle \text{valid-edge} \ a \ \rangle \langle e = (\text{Node} \ \text{source-node} \ a), \ \text{kind} \ a, \ \text{Node} \ (\text{target-node} \ a) \rangle\)
\(\langle \text{src} \ e = \text{src} \ a' \rangle \langle \text{try} e \neq \text{try} a' \rangle\)
show ?case
proof (induct rule: lift-valid-edge.induct)
case lve-edge thus ?case by (auto dest: deterministic simp: kind-def)
next
case (lve-Exit-edge \(e'\))
from \(\langle e = (\text{Node} \ \text{source-node} \ a), \ \text{kind} \ a, \ \text{Node} \ (\text{target-node} \ a) \rangle\)
\(\langle e' = (\text{Node} Exit, (\lambda s. \text{True}) \sqrt{}, \text{NewExit}) \ \rangle \langle \text{src} \ e = \text{src} \ e' \rangle\)
have source-node \(a = \text{Exit} \) by simp
with \(\text{valid-edge} \ a\) have False by (rule Exit-source)
thus ?case by simp
qed auto
qed (fastforce elim: lift-valid-edge.cases simp: kind-def)+
qed

lemma lift-CFGExit:
assumes wf: \text{CFGExit-wf} source-node target-node kind valid-edge Entry Def Use
shows \text{CFGExit} \ \text{src} \ \text{trg} \ \text{kind}
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
\text{NewEntry} \ \text{NewExit}
proof -
interpret \text{CFGExit-wf} source-node target-node kind valid-edge Entry Def Use
state-val Exit
by (rule wf)
interpret CFG: \text{CFG} \ \text{src} \ \text{trg} \ \text{kind}
\text{lift-valid-edge valid-edge source-node target-node kind Entry Exit NewEntry}
by (fastforce intro: lift-CFG wf)
show ?thesis
proof
fix \(a\) assume \text{lift-valid-edge valid-edge source-node target-node kind Entry Exit} \ a
and \( \text{src } a = \text{NewExit} \)
thus \( \text{False} \) by (fastforce elim:lift-valid-edge.cases)
next
from \( \text{lve-Entry-Exit-edge} \)
show \( \exists \ a. \ \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \wedge \)
\( \text{src } a = \text{NewEntry} \wedge \text{try } a = \text{NewExit} \wedge \text{knd } a = (\lambda x. \text{False}) \checkmark \)
by (fastforce simp:knd-def)
qed

\textbf{lemma lift-CFGExit-wf:}
\textbf{assumes } wf::CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
\textbf{shows } CFGExit-wf src trg knd
(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \text{NewEntry}
(\text{lift-Def Def Entry Exit H L}) (\text{lift-Use Use Entry Exit H L}) state-val \text{NewExit}
\textbf{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-uf:CFG-uf 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-uf wf)
\textbf{show } \textbf{?thesis}
\textbf{proof}
show lift-Def Def Entry Exit H L NewExit = \{\} \wedge
lift-Use Use Entry Exit H L NewExit = \{}
by (fastforce elim:lift-Use-set.cases lift-Def-set.cases)
qed

\textbf{3.2.2 Lifting} \text{wod-backward-slice}

\textbf{lemma lift-wod-backward-slice:}
\textbf{fixes } valid-edge \text{ and sourcenode and targetnode and kind and Entry and Exit and Def and Use and H and L}
\textbf{defines } \text{lve:lve } \equiv \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}
\text{and } \text{lDef:lDef } \equiv \text{lift-Def Def Entry Exit H L}
\text{and } \text{lUse:lUse } \equiv \text{lift-Use Use Entry Exit H L}
\textbf{assumes } wf::CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
\textbf{and } H \cap L = \{\} \text{ and } H \cup L = \text{UNIV}
\textbf{shows } NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val
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-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 ∈ 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 ∈ S
  with CFG-wf show n ∈ CFG-wf.wod-backward-slice src trg lve lDef lUse S
    by -(rule CFG-wf.refl)
next
  fix n' S n V
  assume n' ∈ 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 ∈ 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 (∃ 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 \( \text{card} (\text{CFG}_{\text{obs}} \text{ src trg} \text{ lve} \ n) \)

\[
\text{CFG}_{\text{wod-backward-slice}} \text{ src trg} \text{ lve} \ l\text{Def} \ l\text{Use} \ S) \leq 1
\]

by fastforce

next

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

with \( \text{lve} \) show \( \text{try} \ a = \text{NewExit} \lor \text{try} \ a = \text{Node Entry} \)

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

next

from \( \text{lve-Entry-edge} \ \text{lve} \)

show \( \exists \, a. \ \text{lve} \ a \land \text{src} \ a = \text{NewEntry} \land \text{try} \ a = \text{Node Entry} \land \text{knd} \ a = (\lambda s. \text{True}) \)

by (fastforce simp:knd-def)

next

fix \( a \) assume \( \text{lve} \ a \) and \( \text{try} \ a = \text{Node Entry} \)

with \( \text{lve} \) show \( \text{src} \ a = \text{NewEntry} \) by (fastforce elim:lift-valid-edge.cases)

next

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

with \( \text{lve} \) show \( \text{src} \ a = \text{NewEntry} \lor \text{src} \ a = \text{Node Exit} \)

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

next

from \( \text{lve-Exit-edge} \ \text{lve} \)

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

by (fastforce simp:knd-def)

next

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

with \( \text{lve} \) show \( \text{try} \ a = \text{NewExit} \) by (fastforce elim:lift-valid-edge.cases)

next

from \( \text{lDef} \) show \( \text{lDef} (\text{Node Entry}) = H \)

by (fastforce elim:lift-Def-set.cases intro:lift-Def-High)

next

from \( \text{Entry-noteq-Exit} \ \text{lUse} \) show \( \text{lUse} (\text{Node Entry}) = H \)

by (fastforce elim:lift-Use-set.cases intro:lift-Use-High)

next

from \( \text{Entry-noteq-Exit} \ \text{lUse} \) show \( \text{lUse} (\text{Node Exit}) = L \)

by (fastforce elim:lift-Use-set.cases intro:lift-Use-Low)

next

from \( H \cap L = \{\} \) show \( H \cap L = \{\} \).

next

from \( H \cup L = \text{UNIV} \) show \( H \cup L = \text{UNIV} \).

qed

qed

3.2.3 Lifting PDG-BS with standard-control-dependence

lemma lift-Postdomination:

assumes \( \text{wf} : \text{CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit} \)

and \( \text{pd} : \text{Postdomination sourcenode targetnode kind valid-edge Entry Exit} \)

and \( \text{inner} : \text{CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx} \)
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 uf)
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,(λs. False)ₙ,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,(λs. False)ₙ,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,(λs. False)ₙ,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

  22
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry, (\lambda s. True), 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), 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 \langle inner \rangle have valid-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), Node Entry) by (fastforce intro: lve-Entry-edge)
  from \langle \langle Entry - ax @ \rangle \rangle have valid-node (sourcenode ax)
  by (rule inner-is-valid)
  then obtain asx where valid-edge ax and inner-node (sourcenode ax)
    by (fastforce intro: Entry-path)

  from this \langle valid-edge ax \rangle have \langle Entry - ax @ \rangle have valid-node (sourcenode ax)
  by (fastforce dest: Entry-path)

  from this \langle valid-edge ax \rangle 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 \langle Entry - [] @ \rangle 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 = \langle Entry - x s (xs) @ \rangle have \exists es. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node (sourcenode ax));
  have Entry - x s (xs) @ ?thesis by blast

note IH = \langle Entry - x s (xs) @ \rangle have \exists es. CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node (sourcenode ax));
  from \langle Entry - x s (xs) @ \rangle have Entry - x s (xs) @ source-node x and valid-edge x
\[
\text{and } \text{targetnode } x = \text{sourcenode } ax \text{ by (auto elim:path-split-snoc)}
\]

{ \text{assume targetnode } x = \text{Exit} \\
\text{with (valid-edge } ax) \text{ (targetnode } x = \text{sourcenode } ax) \\
\text{have False by } \text{(rule Exit-source, simp +)} \}

\text{hence targetnode } x \neq \text{Exit by clarsimp} \\
\text{with (valid-edge } x) \text{ (targetnode } x = \text{sourcenode } ax) \text{ [THEN sgm]}

\text{have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \\
\text{(Node (sourcenode } x), \text{kind } x, \text{Node (sourcenode } ax)) \\
\text{by (fastforce intro:lift-valid-edge, lve-edge)}

\text{hence path:CFG.path src trg} \\
\text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \\
\text{(Node (sourcenode } x)) \text{ [THEN sym]} \\
\text{have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \\
\text{(Node Entry)} \text{ es (Node (sourcenode } x)) \text{ by blast}

\text{with path have CFG.path src trg} \\
\text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \\
\text{(Node Entry)} \text{ es (Node (sourcenode } x)) \text{ by blast}

\text{thus } ?\text{case by blast}

\text{then obtain es where CFG.path src trg} \\
\text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \\
\text{(Node Entry)} \text{ es (Node (sourcenode } ax)) \text{ by blast}

\text{with path have CFG.path src trg} \\
\text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \\
\text{(Node Entry)} \text{ es (Node (sourcenode } ax)) \text{ by blast}

\text{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 try
(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 try
  (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 try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node (sourcenode x)) es (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 try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node (sourcenode x), kind x, 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 try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (Node Entry) es (Node (sourcenode x)) by blast
  with path have CFG.path src try
  (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

then obtain es where path:CFG.path src try
(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 try
(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

next

fix n assume CFG.valid-node src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
show ∃ as. CFG.path src try
(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 try
(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 try
(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 try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n)
have valid-node m
by (auto elim:lift-valid-edge.cases)
\text{simp:} CFG.\text{valid-node-def}[OF CFG \text{ valid-node-def}]

\text{thus} \ \ ?\text{thesis}

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

\text{case Entry}

\text{from inner obtain } ax \ \text{where} \ \text{valid-edge } ax \ \text{and} \ \text{inner-node} (\text{targetnode } ax) \ \text{and} \ \text{sourcenode } ax = \text{Entry} \ \text{by}(\text{erule inner-node-Entry-edge})

\text{hence} \ \text{edge:} \text{lift-valid-edge} \text{ valid-edge sourcenode targetnode kind } \text{Entry Exit} \\
(Node \text{ Entry,kind } ax, Node (\text{targetnode } ax)) \\
\text{by}(\text{auto intro:} \text{lift-valid-edge lve-edge simp:inner-node-def})

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

\text{hence} \ \text{path:} \text{CFG.path src try} \\
(\text{lift-valid-edge} \text{ valid-edge sourcenode targetnode kind } \text{Entry Exit}) \\
(Node \text{ Exit}) [(Node \text{ Exit,} (\lambda s. \text{True}) \ \text{NewExit}) \ (\text{NewExit})] \ \text{by}(\text{fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF CFG]})

\text{from} (\text{inner-node} (\text{targetnode } ax)) \ \text{have} \ \text{valid-node} (\text{targetnode } ax) \\
\text{by}(\text{rule inner-is-valid})

\text{then obtain } ax \ \text{where} \ \text{targetnode } ax - asx -\ast \text{ Exit} \ \text{by}(\text{fastforce dest:Exit-path})

\text{from this \ valid-edge } ax \ \text{have} \ \exists es. \ \text{CFG.path src try} \\
(\text{lift-valid-edge} \text{ valid-edge sourcenode targetnode kind } \text{Entry Exit}) \\
(Node (\text{targetnode } ax)) \ es \ (Node \text{ Exit})

\text{proof}(\text{induct ax arbitrary:ax})

\text{case Nil}

\text{from} (\text{targetnode } ax - [] -\ast \text{ Exit}) \ \text{have} \ \text{targetnode } ax = \text{Exit} \ \text{by} \ \text{fastforce}

\text{hence} \ \text{CFG.path src try} \\
(\text{lift-valid-edge} \text{ valid-edge sourcenode targetnode kind } \text{Entry Exit}) \\
(Node (\text{targetnode } ax)) \ [] (Node \text{ Exit}) \\
\text{apply simp apply}(\text{rule CFG.empty-path[OF CFG]}) \\
\text{by}(\text{auto intro:lve-Exit-edge simp:CFG.valid-node-def[OF CFG]})

\text{thus} \ ?\text{case by blast}

\text{next}

\text{case} \ (\text{Cons } x xs)

\text{note} \ \text{IH} = \lambda ax. \ [\text{targetnode } ax - xs -\ast \text{ Exit}; \text{valid-edge } ax] \Longrightarrow \exists es. \ \text{CFG.path src try} \\
(\text{lift-valid-edge} \text{ valid-edge sourcenode targetnode kind } \text{Entry Exit}) \\
(Node (\text{targetnode } ax)) \ es \ (Node \text{ Exit})

\text{from} (\text{targetnode } ax - x \# xs -\ast \text{ Exit}) \\
\text{have} \ \text{targetnode } x - xs -\ast \text{ Exit \ and} \ \text{valid-edge } x \\
\ \text{and} \ \text{sourcenode } x = \text{targetnode } ax \ \text{by}(\text{auto elim:path-split-Cons}) \\
\{ \ \text{assume} \ \text{sourcenode } x = \text{Entry} \\
\ \text{with} \ (\text{valid-edge } ax) \ (\text{sourcenode } x = \text{targetnode } ax) \ \text{have} \ \text{False by} -(\text{rule Entry-target,simp+}) \} \\
\text{hence} \ \text{sourcenode } x \neq \text{ Entry} \ \text{by} \ \text{clarsimp}

\text{with} \ (\text{valid-edge } x) \ (\text{sourcenode } x = \text{targetnode } ax) [\text{THEN sym}] \\
\text{have} \ \text{edge:} \text{lift-valid-edge} \text{ valid-edge sourcenode targetnode kind } \text{Entry Exit} \\
(Node (\text{targetnode } ax), \text{kind } x, Node (\text{targetnode } x)) \\
\text{by}(\text{fastforce intro:} \text{lift-valid-edge.lve-edge})

\text{from IH}[OF (\text{targetnode } x - xs -\ast \text{ Exit}) (\text{valid-edge } x)] \ \text{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 ?thesis by blast

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, (\$s. True), NewExit)] 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, (\$s. 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 Exit

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(Node Exit, (\$s. 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, (\$s. 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 \rightarrow* Exit
by (fastforce dest: Exit-path)

with inner have \exists 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 \rightarrow* 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 = \(\exists m. \{inner-node m; m - xs\rightarrow Exit\}\)
  \(\Rightarrow \exists es. \text{CFG.path src try}\)
  (lift-valid-edge valid-edge source node target node kind Entry Exit)
  (Node m) es (Node Exit)
  from (m - x#xs\rightarrow Exit) have target node x - xs\rightarrow Exit
  and valid-edge x and m = source node x by(auto elim:path-split-Cons)
  with (inner-node m)
  have edge:lift-valid-edge valid-edge source node target node kind Entry Exit
  (Node m,kind x,Node (target node x))
  by (fastforce intro:lve-edge simp:inner-node-def)
  from (valid-edge x) have valid-node (target node x) by simp
  thus ?case
  proof (cases target node 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 try
    (lift-valid-edge valid-edge source node target node 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 \(\langle\text{target node x - xs\rightarrow Exit}\rangle\)] obtain es
    where CFG.path src try
    (lift-valid-edge valid-edge source node target node kind Entry Exit)
    (Node (target node x)) es (Node Exit) by blast
  with edge have CFG.path src try
  (lift-valid-edge valid-edge source node target node kind Entry Exit)
  (Node m) [(Node m,kind x,Node (target node 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 try
(lift-valid-edge valid-edge source node target node kind Entry Exit)
(Node m) es (Node Exit) by blast
have lift-valid-edge valid-edge source node target node kind Entry Exit
(Node Exit,(\(\lambda s. \text{True}\),\(\_\)),New Exit) by (fastforce intro:lve-Exit-edge)
hence CFG.path src try
(lift-valid-edge valid-edge source node target node kind Entry Exit)
(Node Exit) [(Node Exit, (\lambda. 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 try
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
n (es@[(Node Exit, (\lambda. True), NewExit)]) NewExit
by (fastforce intro: CFG.path-Append[OF CFG])

thus ?thesis by blast
qed
qed
qed

lemma lift-PDG-scd:
assumes PDG:PDG source-node target-node kind valid-edge Entry Def Use state-val Exit
(Postdomination.standard-control-dependence source-node target-node valid-edge Exit)
and pd:Postdomination source-node target-node kind valid-edge Entry Exit
and inner:CFGExit.inner-node source-node target-node valid-edge Entry Exit nz
shows PDG src try knd
(lift-valid-edge valid-edge source-node target-node 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 try
(lift-valid-edge valid-edge source-node target-node kind Entry Exit) NewExit)

proof
interpret PDG source-node target-node kind valid-edge Entry Def Use state-val Exit
Postdomination.standard-control-dependence source-node target-node
valid-edge Exit

by (rule PDG)

have wf:CFGExit-wf source-node target-node kind valid-edge Entry Def Use state-val Exit by (unfold-locales)

from wf pd inner have pd':Postdomination src try knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit) NewEntry
NewExit
by (rule lift-Postdomination)
from wf have CFG:CFG src try
(lift-valid-edge valid-edge source-node target-node kind Entry Exit) NewEntry
by (rule lift-CFG)
from wf have CFG-wf:CFG-wf src try knd
(lift-valid-edge valid-edge source-node target-node 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 try knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit) NewEntry
NewExit
by (rule lift-CFGExit)
from wf have CFGExit-wf:CFGExit-wf src try knd

30
(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
given a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and try a = NewEntry
with CFG show False by(rule CFG.Entry-target)
next
given a a' assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and try a = try a'
with CFG show a = a' by(rule CFG.edge-det)
next
from CFG-wf
show lift-Def Def Entry Exit H L NewEntry = {} \land 
lift-Use Use Entry Exit H L NewEntry = {}
by(rule CFG-wf.Entry-empty)
next
given a V s assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V \notin lift-Def Def Entry Exit H L (src a) and 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
given a s s' assume asms: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
given 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
given a a'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and try a \neq try a'
with CFG-wf show \exists Q Q'. knd a = (Q) \lor knd a' = (Q') \lor \land
(\forall s. (Q s \rightarrow \neg Q' s) \land (Q' s \rightarrow \neg Q s))
by(rule CFG-wf.deterministic)
next


```plaintext
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 ∧ src a = NewEntry ∧ trg a = NewExit ∧ knd a = (\( \lambda s. \) 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 \( \exists \) 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

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:le ≡ 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)
```

32
and pd:Postdomination sourcenode targetnode kind valid-edge Entry Exit
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit
and H ∩ L = {} and H ∪ L = UNIV
shows NonInterferenceIntraGraph src try knd lve NewEntry lDef lUse state-val
(PDG.PDG-BS src try lve lDef lUse
   (Postdomination.standard-control-dependence src try 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-locale)
interpret wf’:CFGExit-wf src try 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 try knd
   lve NewEntry lDef lUse state-val NewExit
   (Postdomination.standard-control-dependence src try lve NewExit)
   by (fastforce intro: lift-PDG-scd)
from wf pd inner have pd’:Postdomination src try knd
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   NewEntry NewExit
   by (rule lift-Postdomination)
from wf lve have CFG:CFG src try lve NewEntry
   by (fastforce intro: lift-CFG)
from wf lve lDef lUse have CFG-wf:CFG-wf src try knd lve NewEntry lDef lUse state-val
   by (fastforce intro: lift-CFG-wf)
from wf lve have CFGExit:CFGExit src try knd lve NewEntry NewExit
   by (fastforce intro: lift-CFGExit)
from wf lve lDef lUse have CFGExit-wf:CFGExit-wf src try 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 try lve lDef lUse
   (Postdomination.standard-control-dependence src try lve NewExit) S
with PDG’ show CFG.valid-node src try lve n
   by (rule PDG.PDG-BS-valid-node)
next
fix n S assume CFG.valid-node src try lve n and n ∈ S
thus n ∈ PDG.PDG-BS src try lve lDef lUse
   (Postdomination.standard-control-dependence src try lve NewExit) S
   by (fastforce intro: PDG.PDG-path-nil [OF PDG’] simp: PDG.PDG-BS-def [OF PDG’])
fix $n'\ S\ n\ V$

assume $n' \in PDG\cdot PDG-BS\ src\ trg\ lve\ lDef\ lUse$

(Postdomination.standard-control-dependence\ src\ trg\ lve\ NewExit)\ S$

and $\ CFG\cdot wf\cdot data-dependence\ src\ trg\ lve\ lDef\ lUse\ n\ V\ n'$

thus $n \in PDG\cdot PDG-BS\ src\ trg\ lve\ lDef\ lUse$

(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:split-if-asm)

next

fix $n\ S$

interpret $PDGx\cdot PDG\ src\ trg\ knd\ lve\ NewEntry\ lDef\ lUse\ state-val\ NewExit$

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-locale)

from StandardControlDependencePDG.obs-singleton[OF scd]

have $(\exists m.\ CFG\cdot obs\ src\ trg\ lve\ n)$

(PDG.PDG-BS src trg lve lDef lUse

(Postdomination.standard-control-dependence\ src\ trg\ lve\ NewExit)\ S) = \{m\}$

∨

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-locale)

from StandardControlDependencePDG.obs-singleton[OF scd]

have $(\exists m.\ CFG\cdot obs\ src\ trg\ lve\ n)$

(PDG.PDG-BS src trg lve lDef lUse

(Postdomination.standard-control-dependence\ src\ trg\ lve\ NewExit)\ S) = \{m\}$

∨

CFG.obs src trg lve n
(PDG_PDGBS src trg lve lDef lUse
  (Postdomination.standard-control-dependence src trg lve NewExit) S) = {} by (fastforce simp: StandardControlDependencePDG_PDGBS-s-def [OF src trg lve lDef lUse])

thus card (CFG.obs src trg lve n (PDG_PDGBS 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 try a = NewExit \lor try a = Node Entry
    by (fastforce elim: lift-valid-edge.cases)

next
  from lve-Entry-edge lve show \exists a. lve a \land src a = NewEntry \land try a = Node Entry \land knd a = (\lambda s. True) \lor
    by (fastforce simp: knd-def)

next
  fix a assume lve a and try a = Node Entry
  with lve show src a = NewEntry by (fastforce elim: lift-valid-edge.cases)

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

next
  from lve-Exit-edge lve show \exists a. lve a \land src a = NewExit \land try a = NewExit \land knd a = (\lambda s. True) \lor
    by (fastforce simp: knd-def)

next
  fix a assume lve a and src a = Node Exit
  with lve show try 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-locals)
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 \( \{ m', \exists a'. \text{valid-edge } a' \land \text{sourcenode } a' = m \land \text{targetnode } a' = m' \} \)
by (rule successor-set-finite)

have \( \{ m', \exists a'. \text{valid-edge } a' \land \text{sourcenode } a' = m \land \text{targetnode } a' = m' \} \subseteq \{ m'. \exists a'. \text{valid-edge } a' \land \text{sourcenode } a' = m' \land \text{targetnode } a' = m' \} \)
by (fastforce elim: lift-valid-edge.cases)

with finite \( \{ m', \exists a'. \text{valid-edge } a' \land \text{sourcenode } a' = m \land \text{targetnode } a' = m' \} \)

have finite \( \{ m', \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind } Entry Exit a' \land \text{src } a' = \text{Node } m \land \text{trg } a' = \text{Node } m' \} \subseteq \{ m'. \exists a'. \text{valid-edge } a' \land \text{sourcenode } a' = m \land \text{targetnode } a' = m' \} \)
by (rule finite-subset)

hence finite \( \{ m', \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind } Entry Exit a' \land \text{src } a' = \text{Node } m \land \text{trg } a' = \text{Node } m' \} \)

by fastforce

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 try kind
(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 try
(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 try knnd

(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 knnd

(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 knnd

(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 knnd

(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 \( \text{thesis} \)

proof

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

and try \( a = \) 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 try \( a = \) try \( a' \)

with \( CFG \) show \( a = a' \) by (rule CFG.edge-det)

next

from \( CFG-wf \)

show lift-Def Def Entry Exit H L NewEntry = \{ \} \land

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 \( V \notin lift-Def Def Entry Exit H L \) (src \( a \)) and pred (knnd \( a \)) \( s \)

with \( CFG-wf \) show state-val (transfer (knnd \( a \)) \( s \)) \( V = \) 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 \( a \)
\( \forall V \in \text{lift-Use} \hspace{1em} \text{Use Entry Exit H L (src a)}. \) state-val \( s \) \( V = \text{state-val } s' \) \( V \)

\( \text{pred (knd a) } s \) \( \text{pred (knd a) } s' \)

with \( \text{CFG-wf} \) show \( \forall V \in \text{lift-Def} \hspace{1em} \text{Def Entry Exit H L (src a)}. \)

state-val (transfer (knd a) ) \( s \) \( V = \text{state-val (transfer (knd a) ) } s' \) \( V \)

by (rule \( \text{CFG-wf} \). \( \text{CFG-edge-transfer-uses-only-Use} \))

next

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

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

and \( \text{pred (knd a) } s \)

and \( \forall V \in \text{lift-Use} \hspace{1em} \text{Use Entry Exit H L (src a)}. \) state-val \( s \) \( V = \text{state-val } s' \) \( V \)

with \( \text{CFG-wf} \) show \( \text{pred (knd a) } s' \) by (rule \( \text{CFG-wf} \). \( \text{CFG-edge-Uses-pred-equal} \))

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 \( \text{src a = src a' and trg a \neq trg a'} \)

with \( \text{CFG-wf} \) show \( \exists Q Q'. \text{knd a = (Q)} \lor \land \text{knd a' = (Q')} \lor \land \)

\( (\forall s. (Q s \rightarrow \neg Q' s) \land (Q' s \rightarrow \neg Q s)) \)

by (rule \( \text{CFG-wf} \). deterministic)

next

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

and \( \text{src a = NewExit} \)

with \( \text{CFGExit} \) show \( \text{False} \)

by (rule \( \text{CFGExit} \). Exit-source)

next

from \( \text{CFGExit} \)

show \( \exists a. \) lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a \land

\( \text{src a = NewEntry} \land \text{trg a = NewExit} \land \text{knd a = (\lambda s. False)} \lor \land \)

by (rule \( \text{CFGExit} \). Entry-Exit-edge)

next

from \( \text{CFGExit-wf} \)

show lift-Def Def Entry Exit H L NewExit = \{\} \land

\( \text{lift-Use Use Entry Exit H L NewExit = \{} \land \

by (rule \( \text{CFGExit-wf} \). Exit-empty)

next

fix \( n \) \( n' \)

assume \( \text{wcd:StrongPostdomination.weak-control-dependence src trg} \)

\((\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \) NewExit \( n \)

\( n' \)

show \( n' \neq \text{NewExit} \)

proof (rule ccontr)

assume \( \neg n' \neq \text{NewExit} \)

hence \( n' = \text{NewExit} \) by simp

with \( \text{wcd spd'} \) show \( \text{False} \)

by (fastforce intro: StrongPostdomination.Exit-not-weak-control-dependent)

qed

next

fix \( n \) \( n' \)

assume \( \text{StrongPostdomination.weak-control-dependence src trg} \)

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

thus \exists \text{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 \text{lift-valid-edge}\ valid-edge\ sourcenode\ targetnode\ kind\ Entry\ Exit

and lDef: lDef \equiv \text{lift-Def}\ Def\ Entry\ Exit\ H\ L

and lUse: lUse \equiv \text{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 nz

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

40
have \( CFGwf : CFGwf \ src \ trg \ knd \ lve \ NewEntry \ lDef \ lUse \ state-val \) by (fastforce intro: lift-CFGwf) from \( wf \ lve \ have \ CFGExit : CFGExit \ src \ trg \ knd \ lve \ NewEntry \ NewExit \) by (fastforce intro: lift-CFGExit) from \( wf \ lve \ lDef \ lUse \) have \( CFGExitwf : CFGExit-wf \ src \ trg \ knd \ lve \ NewEntry \ lDef \ lUse \ state-val \ NewExit \) by (fastforce intro: lift-CFGExit-wf)

show \( ?thesis \)
proof
  fix \( n \) \( S \)
  assume \( n \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse \) (StrongPostdomination.weak-control-dependence src trg lve NewExit) \( S \)
  with \( PDG' \) show \( CFG.\ valid-nodes \ src \ trg \ lve \ n \)
    by (rule PDG.PDG-BS-valid-node)
next
  fix \( n \) \( S \)
  assume \( CFG.\ valid-nodes \ src \ trg \ lve \ n \) and \( n \in S \)
  thus \( n \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse \) (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 \( n' \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse \) (StrongPostdomination.weak-control-dependence src trg lve NewExit) \( S \)
  and \( CFGwf.\ data-dependence \ src \ trg \ lve \ lDef \ lUse \ n \ V \ n' \)
  thus \( n \in PDG.PDG-BS \ src \ trg \ lve \ lDef \ lUse \) (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: split-if-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 \( spd : StrongPostdomination \ src \ trg \ knd \ lve \ NewEntry \ NewExit \)
    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 \( \exists m. \ CFG.\ obs \ src \ trg \ lve \ n \)
    (PDG.PDG-BS src trg lve lDef lUse)
    (StrongPostdomination.weak-control-dependence src trg lve NewExit) \( S \) = \{m\})
    \( \vee \)
    CFG.\ obs \ src \ trg \ lve \ n
    (PDG.PDG-BS src trg lve lDef lUse)
\[(\text{StrongPostdomination.weak-control-dependence src trg lve NewExit}) \implies \{\}\]
\[\text{by (fastforce simp: WeakControlDependencePDG.PDG-BS-w-def[OF wcd])}\]
\[\text{thus finite (CFG.obs src trg lve n)}\]
\[\text{(PDG.PDG-BS src trg lve lDef lUse)}\]
\[\text{(StrongPostdomination.weak-control-dependence src trg lve NewExit}) \implies \{\}\]
\[\text{by fastforce}\]
\[\text{next}\]
\[\text{fix n S}\]
\[\text{interpret PDGx:PDG src trg knd lve NewEntry lDef lUse state-val NewExit}\]
\[\text{StrongPostdomination.weak-control-dependence src trg lve NewExit}\]
\[\text{by (rule PDG')}\]
\[\text{interpret spd:StrongPostdomination src trg knd lve NewEntry NewExit}\]
\[\text{by (fastforce intro: spd' simp: lve)}\]
\[\text{have wcd:WeakControlDependencePDG src trg knd lve NewEntry lDef lUse state-val NewExit by (unfold-locales)}\]
\[\text{from WeakControlDependencePDG.obs-singleton[OF wcd]}\]
\[\text{have (}\exists m. \text{CFG.obs src trg lve n)}\]
\[\text{(PDG.PDG-BS src trg lve lDef lUse)}\]
\[\text{(StrongPostdomination.weak-control-dependence src trg lve NewExit}) \implies \{m\}\}\]
\[\text{by (fastforce simp: WeakControlDependencePDG.PDG-BS-w-def[OF wcd])}\]
\[\text{thus card (CFG.obs src trg lve n)}\]
\[\text{(PDG.PDG-BS src trg lve lDef lUse)}\]
\[\text{(StrongPostdomination.weak-control-dependence src trg lve NewExit}) \leq 1}\]
\[\text{by fastforce}\]
\[\text{next}\]
\[\text{fix a assume lve a and src a = NewEntry}\]
\[\text{with lve show trg a = NewExit} \lor \text{trg a = Node Entry}\]
\[\text{by (fastforce elim:lift-valid-edge.cases)}\]
\[\text{next}\]
\[\text{from lve-Entry-edge lve}\]
\[\text{show (}\exists a. \text{lve a} \land \text{src a = NewEntry} \land \text{trg a = Node Entry} \land \text{knd a = (}\lambda s. True\text{))}\]
\[\text{by (fastforce simp: knd-def)}\]
\[\text{next}\]
\[\text{fix a assume lve a and trg a = Node Entry}\]
\[\text{with lve show src a = NewEntry by (fastforce elim:lift-valid-edge.cases)}\]
\[\text{next}\]
\[\text{fix a assume lve a and trg a = NewExit}\]
\[\text{with lve show src a = NewEntry} \lor \text{src a = Node Exit}\]
\[\text{by (fastforce elim:lift-valid-edge.cases)}\]
\[\text{next}\]
\[\text{from lve-Exit-edge lve}\]
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 U L = UNIV show H U L = UNIV .
qed
qed

end

4 Information Flow for While

theory NonInterferenceWhile imports
  ../Slicing/While/SemanticsWellFormed
  ../Slicing/While/StaticControlDependences
  LiftingIntra
begin

locale SecurityTypes =
  fixes H :: vname set
  fixes L :: vname set
  assumes HighLowDistinct: H ∩ L = {}
  and HighLowUNIV: H U 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 try
(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,(λs. True) _,Node Entry)
  by (fastforce intro:lve-Entry-edge)
  with While-CFGExit-uf-aux[of prog] ⟨n = sourcenode a; True show ?thesis
  by (fastforce simp:While-CFGExit-wf-aux[of prog]
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-uf-aux[of prog] ⟨n = sourcenode a⟩ show ?thesis
  by (fastforce simp:While-CFGExit-wf-aux[of prog]
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,(λs. True) _,NewExit)
  by (fastforce intro:lve-Exit-edge)
  with While-CFGExit-uf-aux[of prog] ⟨n = targetnode a; True show ?thesis
  by (fastforce simp:While-CFGExit-wf-aux[of prog]
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-uf-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 \( \langle c, s \rangle \rightarrow^* \langle c', s' \rangle \)
shows \( \exists n' \text{ as } CFG.\text{path } src \text{ trg} \)
(lift-valid-edge (valid-edge prog) source node target node kind (-Entry-) (-Exit-))
n as n' \land transfers (CFG.\text{kind as} s = s' \land
preds (CFG.\text{kind as} s \land labels-LDCFG-nodes prog n' c')

proof
from \( \langle \\text{labels-LDCFG-nodes } \text{prog } n \text{ c} \rangle \) obtain nx where \( n = \text{Node } nx \)
and labels-nodes prog nx c by (cases n) auto
from \( \langle \\text{labels-nodes } \text{prog } n \text{ c} \rangle \) ⟨\( \langle c, s \rangle \rightarrow^* \langle c', s' \rangle \)⟩
obtain n' as where prog ⊢ nx \rightarrow^* n' \land transfers \( (CFG.\text{kind as} s = s' \land
preds (CFG.\text{kind as} s) \land labels-nodes prog n' c' \land
\langle prog ⊢ nx \rightarrow^* n' \rangle \langle \\text{labels-nodes } \text{prog } n' \text{ c'} \rangle \)

proof (induct arbitrary: n s c rule: WCFG-path-induct)
case (empty-path n nx)
from \( \langle CFG.\text{valid-node } \text{source node } \text{target node } \text{valid-edge prog} \text{ } n \rangle \)

have valid-node: \( CFG.\text{valid-node } \text{src } \text{ trg} \)
lift-valid-edge (valid-edge prog) source node target node kind (-Entry-) (-Exit-)
(\( \text{Node } nx \)) es (\( \text{Node } n' \) \land transfers \( (CFG.\text{kind as} s = s' \land
preds (CFG.\text{kind as} s) \land labels-nodes prog n' c' \land
\langle prog ⊢ nx \rightarrow^* n' \rangle \langle \\text{labels-nodes } \text{prog } n' \text{ c'} \rangle \)

have \( \exists es. \text{CFG.\text{path } src } \text{ trg} \)
lift-valid-edge (valid-edge prog) source node target node kind (-Entry-) (-Exit-)
(\( \text{Node } nx \)) es (\( \text{Node } n' \) \land transfers \( (CFG.\text{kind as} s = s' \land
preds (CFG.\text{kind as} s) \land labels-nodes prog n' c' \land
\langle prog ⊢ nx \rightarrow^* n' \rangle \langle \\text{labels-nodes } \text{prog } n' \text{ c'} \rangle \)

proof (induct arbitrary: n s c rule: WCFG-path-induct)
case (empty-path n nx)
from \( \langle CFG.\text{valid-node } \text{source node } \text{target node } \text{valid-edge prog} \text{ } n \rangle \)

have valid-node: \( CFG.\text{valid-node } \text{src } \text{ trg} \)
lift-valid-edge (valid-edge prog) source node target node kind (-Entry-) (-Exit-)
(\( \text{Node } n \))
by (rule lift-valid-node)

have CFG.\text{kind as} [] :: \( (w-node \text{ LDCFG-node } \times \text{ state } \text{edge-kind } \times \text{w-node } \text{ LDCFG-node} \) list) =

by (simp add: CFG.\text{kind-def}[OF lift-CFG[OF While-CFGExit-wf-aux]])
with \( \langle \\text{transfers } \text{CFG.\text{kind kind [] } s = s'} \rangle \langle \\text{preds } \text{CFG.\text{kind kind [] } s} \rangle \)
valid-node

show \( ? \) case
by (fastforce intro: CFG.empty-path[OF lift-CFG[OF While-CFGExit-wf-aux]]
simp: While-CFG.\text{kind-def})

next

case (Cons-path n'' as n' a nx)
\textbf{note IH} = \{\forall n \ s \ c. \ \#\text{transfers (CFG.kinds kind as) \ s} = s'; \\
\text{pdds (CFG.kinds kind as) \ s; \ n = LDCFG-node.Node \ n''; \\
\text{labels-nodes prog \ n'' \ c; \ labels-nodes prog \ n' \ c}\}
\implies \exists \ es. \ CFG.path \ src \ trg

\text{(lift-valid-edge (valid-edge prog) source-node target-node kind \ (-Entry-) \ (-Exit-))}
\text{\langle LDCFG-node.Node \ n'' \rangle \ es \ \langle LDCFG-node.Node \ n' \rangle \ \land

\text{transfers (CFG.kinds kind as) \ s} = s' \ \land
\text{pdds (CFG.kinds kind as) \ s;}

\text{have transfers (CFG.kinds kind as) \ (transfer (kind a) \ s)} = s'

\text{by(simp add: While-CFG.kinds-def)}
\text{from \ pdds (CFG.kinds kind (a \ # \ as)) \ s;}
\text{have pdds (CFG.kinds kind as) \ (transfer (kind a) \ s)}
\text{and \ pred (kind a) \ s \ by(simp-all add: While-CFG.kinds-def)}
\text{show \ ?case}
\text{proof(cases source-node a = (-Entry-))}
\text{case True}
\text{with \ (source-node a = nx): \ (labels-nodes prog nx c) \ have False by simp}
\text{thus \ ?thesis by simp}
\text{next}
\text{case False}
\text{with \ (valid-edge prog a) \ (valid-edge prog) \ source-node target-node kind}
\text{\ Entry \ Exit \ (Node \ (source-node a),kind a,Node \ (target-node a))}
\text{by(fastforce intro:ve-edge)

\text{from \ (prog \ n'' \as\rightarrow\ n')}
\text{have \ CFG.valid-node source-node target-node \ (valid-edge prog) \ n''}
\text{by(rule While-CFG.path-valid-node)

\text{then obtain c'' where \ (labels-nodes prog n'' \ c'')
\text{proof(cases rule: While-CFGExit.valid-node-cases)
\text{case Entry}
\text{with \ (target-node a = n''): \ (valid-edge prog a) \ have False by fastforce}
\text{thus \ ?thesis by simp
\text{next}
\text{case Exit}
\text{with \ (prog \ n'' \as\rightarrow\ n') \ have \ n' = (-Exit-) \ by fastforce}
\text{with \ (labels-nodes prog n' \ c') \ have False by fastforce}
\text{thus \ ?thesis by simp
\text{next}
\text{case inner}
\text{then obtain l'' where \ simp;n'' = (- l'' -) \ by(cases n'') \ auto

\text{with \ (valid-edge prog a) \ (target-node a = n'') \ have \ l'' < #:prog

\text{by(fastforce intra:WCFG-targetlabel-less-num-nodes simp:valid-edge-def)

\text{then obtain c'' where \ labels prog l'' \ c''

\text{by(fastforce dest:less-num-inner-nodes-label)

\text{with \ that \ show \ ?thesis by fastforce

\text{qed

\text{from \ IH[OF \ (transfers (CFG.kinds kind as) \ (transfer (kind a) \ s)) = s'\}
\text{\ (pdds (CFG.kinds kind as) \ (transfer (kind a) \ s)) \ - \ this
\text{\ (labels-nodes prog n' \ c'')}}

46
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 kind es) (transfer (kind a) s) = s′
and preds (CFG.kinds kind 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
by (fastforce intro:CFG.Cons-path[OF lift-CFG[OF While-CFGExit-wf-aux]])
with edge have knl (Node (sourcenode a),kind a,Node (targetnode a)) =
kind a
by (simp add:kind-def)
with ⟨transfers (CFG.kinds kind es) (transfer (kind a) s) = s′⟩
⟨preds (CFG.kinds kind es) (transfer (kind a) s)⟩ ⟨pred (kind a) s⟩
have transfers
(CFG.kinds kind ((Node (sourcenode a),kind a,Node (targetnode a))) # es) s = s′
and preds
(CFG.kinds kind ((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 \textbf{?case by}(fastforce intro:\texttt{WCFG-LAssSkip})

next

\textbf{case} (\texttt{Seq} \(b\) \(c_1\) \(c_2\))

\textbf{note} \(IH1 = (\forall n.\, \text{labels-nodes} \; c_1 \; n \; \text{Skip} \Rightarrow c_1 \; \vdash \; n \; \downarrow \text{id} \rightarrow \text{-\text{l-}})\)

\textbf{from} \(\text{\texttt{labels-nodes} (\; c_1 {}{}; c_2 {}{} \; n \; \text{Skip}) \; \text{obtain} \; l \; \text{where} \; n = (\; - \; l \; - \; )\)

\quad \textbf{and} \(l \geq \# : c_1\) \textbf{and} \(\text{labels-nodes} \; c_2 \; (l \; - \; \# : c_1 \; - \; \text{Skip})\)

\textbf{by} (\texttt{cases} \(n\)) (\texttt{auto elim:labels.cases})

\textbf{from} \(IH2[\texttt{OF} \; (\text{\texttt{labels-nodes} c_2 \; (\; - \; l \; - \; \# c_1 \; - \; ) \; \text{Skip})}]\)

\textbf{have} \(c_2 \; \vdash \; (l \; - \; \# : c_1 \; - \; \text{id} \rightarrow \text{-\text{l-}})\).

\textbf{with} \((l \geq \# : c_1)\) \textbf{have} \((c_1 {}{}; c_2 {}{} \; \vdash \; (l \; - \; \# : c_1 \; - \; \text{id} \rightarrow \text{-\text{l-}}) \; \oplus \; \# : c_1)\)

\textbf{by} (fastforce intro:WCFG-SeqSecond)

\textbf{with} \((n = (\; - \; l \; - \; ) \; \cdot \; l \geq \# : c_1)\) \textbf{show} \textbf{?case by}(simp add:id-def)

next

\textbf{case} (\texttt{Cond} \(b\) \(c_1\) \(c_2\))

\textbf{note} \(IH1 = (\forall n.\, \text{labels-nodes} \; c_1 \; n \; \text{Skip} \Rightarrow c_1 \; \vdash \; n \; \downarrow \text{id} \rightarrow \text{-\text{l-}})\)

\textbf{note} \(IH2 = (\forall n.\, \text{labels-nodes} \; c_2 \; n \; \text{Skip} \Rightarrow c_2 \; \vdash \; n \; \downarrow \text{id} \rightarrow \text{-\text{l-}})\)

\textbf{from} \(\text{\texttt{labels-nodes} (\; (b \; c_1 \; \text{else} \; c_2) \; n \; \text{Skip})} \)

\textbf{obtain} \(l \; \text{where} \; n = (\; - \; l \; - \; )\) \textbf{and} \(\text{disj:}(l \; \geq \; 1 \; \land \; \text{labels-nodes} \; c_1 \; (\; - \; l \; - \; \text{Skip}) \; \lor \; (\; l \; \geq \; \# : c_1 \; + \; 1 \; \land \; \text{labels-nodes} \; c_2 \; (\; - \; l \; - \; \# : c_1 \; - \; 1 \; - \; \text{Skip}) \; \lor \; (\; \text{cases} \; n\) (\texttt{fastforce elim:labels.cases})+\)

\textbf{from} \(\text{\texttt{disj show} \; ?case}\)

\textbf{proof}

\textbf{assume} \((1 \leq l \; \land \; \text{labels-nodes} \; c_1 \; (\; - \; l \; - \; \text{Skip}) \; \text{Skip})\)

\textbf{hence} \((1 \leq l) \; \text{and} \; \text{labels-nodes} \; c_1 \; (\; - \; l \; - \; \text{Skip}) \; \text{by simp-all}\)

\textbf{from} \(IH1[\texttt{OF} \; (\text{\texttt{labels-nodes} c_1 \; (\; - \; l \; - \; \text{Skip})}]\)

\textbf{have} \((c_1 \; \vdash \; (\; - \; l \; - \; \text{Skip}) \; \downarrow \text{id} \rightarrow \text{-\text{l-}})\).

\textbf{with} \((1 \leq l)\) \textbf{have} \((b \; c_1 \; \text{else} \; c_2 \; \vdash \; (\; - \; l \; - \; \text{Skip}) \; \lor \; (\; - \; l \; - \; \text{Skip}) \; \downarrow \text{id} \rightarrow \text{-\text{l-}} \; \oplus \; \text{id-def})\)

\textbf{by} (fastforce intro:WCFG-CondThen)

\textbf{with} \((n = (\; - \; l \; - \; ) \; \cdot \; l \leq b)\) \textbf{show} \textbf{?case by}(simp add:id-def)

next

\textbf{assume} \((\# : c_1 \; + \; 1 \leq l \; \land \; \text{labels-nodes} \; c_2 \; (\; - \; l \; - \; \# : c_1 \; - \; \text{Skip}) \; \text{Skip})\)

\textbf{hence} \((\# : c_1 \; + \; 1 \leq l) \; \text{and} \; \text{labels-nodes} \; c_2 \; (\; - \; l \; - \; \# : c_1 \; - \; \text{Skip}) \; \text{by simp-all}\)

\textbf{from} \(IH2[\texttt{OF} \; (\text{\texttt{labels-nodes} c_2 \; (\; - \; l \; - \; \# c_1 \; - \; \text{Skip})}]\)

\textbf{have} \((c_2 \; \vdash \; (\; - \; l \; - \; \# c_1 \; - \; \text{Skip}) \; \downarrow \text{id} \rightarrow \text{-\text{l-}})\).

\textbf{with} \((\# : c_1 \; + \; 1 \leq b)\) \textbf{have} \((b \; c_1 \; \text{else} \; c_2 \; \vdash \; (\; - \; l \; - \; \# : c_1 \; - \; \text{Skip}) \; \lor \; (\; \# : c_1 \; + \; 1) \; \downarrow \text{id} \rightarrow \text{-\text{l-}} \; \oplus \; (\# : c_1 \; + \; 1))\)

\textbf{by} (fastforce intro:WCFG-CondElse)

\textbf{with} \((n = (\; - \; l \; - \; ) \; \cdot \; \# : c_1 \; + \; 1 \leq b)\) \textbf{show} \textbf{?case by}(simp add:id-def)

qed

next

\textbf{case} (\texttt{While} \(b\) \(c\))

\textbf{from} \(\text{\texttt{labels-nodes} (\; (\texttt{while} \; (b) \; c) \; n \; \text{Skip})} \; \text{have} \; n = (\; - \; 1 \; - \; )\)

\textbf{by} (\texttt{cases} \(n\)) (\texttt{auto elim:labels.cases})

\textbf{thus} \textbf{?case by}(fastforce intro:WCFG-WhileFalseSkip)

qed

\textbf{4.2} \textbf{Semantic Non-Interference for Weak Order Dependence}

\textbf{lemmas} \texttt{WODNonInterferenceGraph} =

\textbf{48}
\textit{lift-wod-backward-slice}[OF While-CFGExit-wf-aux HighLowDistinct HighLowU-NIV]

\textbf{lemma} \textit{WODNonInterference:}

\textit{NonInterferenceIntra src trg knd}
\hspace{1cm}(\textit{lift-valid-edge (valid-edge prog) sourcenode targetnode kind})
\hspace{1cm}(-Entry-) (-Exit-))
\hspace{1cm}\textit{NewEntry (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)}
\hspace{1cm}(\textit{lift-Use (Uses prog) (-Entry-) (-Exit-) H L id})
\hspace{1cm}(\textit{CFG-wf,wod-backward-slice src trg})
\hspace{1cm}(\textit{lift-valid-edge (valid-edge prog) sourcenode targetnode kind})
\hspace{1cm}(-Entry-) (-Exit-))
\hspace{1cm}(\textit{lift-Def (Defs prog) (-Entry-) (-Exit-) H L})
\hspace{1cm}(\textit{lift-Use (Uses prog) (-Entry-) (-Exit-) H L}))
\hspace{1cm}\textit{reds (labels-LDCFG-nodes prog)}
\hspace{1cm}\textit{NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final}

\textbf{proof --}

\textbf{interpret} \textit{NonInterferenceIntraGraph src trg knd}
\hspace{1cm}(\textit{lift-valid-edge (valid-edge prog) sourcenode targetnode kind})
\hspace{1cm}(-Entry-) (-Exit-))
\hspace{1cm}\textit{NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L}
\hspace{1cm}(\textit{lift-Use (Uses prog) (-Entry-) (-Exit-) H L id})
\hspace{1cm}(\textit{CFG-wf,wod-backward-slice src trg})
\hspace{1cm}(\textit{lift-valid-edge (valid-edge prog) sourcenode targetnode kind})
\hspace{1cm}(-Entry-) (-Exit-))
\hspace{1cm}(\textit{lift-Def (Defs prog) (-Entry-) (-Exit-) H L})
\hspace{1cm}(\textit{lift-Use (Uses prog) (-Entry-) (-Exit-) H L})
\hspace{1cm}\textit{NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node.Node (-Exit-)}
\hspace{1cm}\textbf{by(rule WODNonInterferenceGraph)}

\textbf{interpret} \textit{BackwardSlice-wf src trg knd}
\hspace{1cm}(\textit{lift-valid-edge (valid-edge prog) sourcenode targetnode kind})
\hspace{1cm}(-Entry-) (-Exit-))
\hspace{1cm}\textit{NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L}
\hspace{1cm}(\textit{lift-Use (Uses prog) (-Entry-) (-Exit-) H L id})
\hspace{1cm}(\textit{CFG-wf,wod-backward-slice src trg})
\hspace{1cm}(\textit{lift-valid-edge (valid-edge prog) sourcenode targetnode kind})
\hspace{1cm}(-Entry-) (-Exit-))
\hspace{1cm}(\textit{lift-Def (Defs prog) (-Entry-) (-Exit-) H L})
\hspace{1cm}(\textit{lift-Use (Uses prog) (-Entry-) (-Exit-) H L})
\hspace{1cm}\textit{reds labels-LDCFG-nodes prog}

\textbf{proof(unfold-locale)}
\hspace{1cm}\textbf{fix} n c s c' s'
\hspace{1cm}\textbf{assume} labels-LDCFG-nodes prog n c and ⟨c,s⟩ ↔ ⟨c',s'⟩
\hspace{1cm}\textbf{thus} \exists n' as. CFG.path src trg
\hspace{1cm}(\textit{lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))}
\hspace{1cm}n as n' ∧ transfers (CFG.kinds knd as) s = s' ∧
\hspace{1cm}\textit{preds (CFG.kinds knd as) s ∧ labels-LDCFG-nodes prog n' c'}
\hspace{1cm}\textbf{by(rule lifted-CFG-fund-prop)}

\textbf{qed}
show \( ?\text{thesis} \)

proof (unfold-locales)
  fix \( c \ n \)
  assume final \( c \) and labels-LDCFG-nodes prog \( n \ c \)
  from final \( c \) have \([\text{simp}]: c = \text{Skip}\) by (cases \( c \)) auto
  from labels-LDCFG-nodes prog \( n \ c \) obtain \( n x \) where \([\text{simp}]: n = \text{Node} n x\)
  and labels-nodes prog \( n x \) Skip by (cases \( n \)) auto
  from \( \langle \text{labels-nodes prog} \ n x \ \text{Skip} \rangle \) have \( n \vdash n x \sim \uparrow \text{id} \rightarrow \sim (\text{-Exit-}) \)
  by (rule final-edge)
  then obtain \( a \) where valid-edge prog \( a \) and sourcenode \( a = n x \)
  and kind \( a = \uparrow \text{id} \) and targetnode \( a = (\text{-Exit-}) \)
  by (auto simp: valid-edge-def)
  with \( \langle \text{labels-nodes prog} \ n x \ \text{Skip} \rangle \) show \( \\
  \exists a. \ \text{lift-valid-edge} (\text{valid-edge prog}) \ \text{sourcenode} \ \text{targetnode} \)
  kind \( (\text{-Entry-}) \) \( (\text{-Exit-}) \) \( a \) \wedge
  src \( a = n \) \wedge trg \( a = \text{LDCFG-node. Node} (\text{-Exit-}) \) \wedge \( \text{kind} \ a = \uparrow \text{id} \)
  by (rule-tac \( x = (\text{Node} n x, \uparrow \text{id}, \text{Node} (\text{-Exit-})) \) in \( \text{exI} \))
  (auto intro \! : \text{lve-edge} simp: knd-def valid-edge-def)
qed

4.3 Semantic Non-Interference for Standard Control Dependence

lemma inner-node-exists: \( \exists n. \ \text{CFGExit. inner-node} \ \text{sourcenode} \ \text{targetnode} \)
  \( (\text{valid-edge prog}) \ (\text{-Entry-}) \ (\text{-Exit-}) \) \( n \)
proof –
  have prog \( \vdash (\text{-Entry-}) \sim (\lambda s. \text{True}) \sim (\text{-0-}) \) by (rule WCFG-Entry)
  hence CFG. valid-node sourcenode targetnode \( \text{targetnode} (\text{valid-edge prog}) \ (\text{-0-}) \)
  by (auto simp: valid-edge-def)
  thus \( ?\text{thesis} \) 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 \( \text{valid-edge prog} \) \( \text{sourcenode} \ \text{targetnode} \) kind
  \( (\text{-Entry-}) \) \( (\text{-Exit-}) \))
  NewEntry \( (\text{lift-Def} \ (\text{Defs prog}) \ (\text{-Entry-}) \ (\text{-Exit-}) \ \text{H L}) \)
  \( \text{(lift-Use} \ (\text{Uses prog}) \ (\text{-Entry-}) \ (\text{-Exit-}) \ \text{H L}) \) \( \text{id} \)
  \( \text{PDG. PDG-BS src trg} \)
  (lift-valid-edge \( \text{valid-edge prog} \) \( \text{sourcenode} \ \text{targetnode} \) kind
  \( (\text{-Entry-}) \) \( (\text{-Exit-}) \))
  (lift-Def \ (\text{Defs prog}) \ (\text{-Entry-}) \ (\text{-Exit-}) \ \text{H L})

50
(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 targetnode

(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** \( \text{final } c \) and \( \text{labels-LDCFG-nodes prog } n \) \( c \)

**from** \( \text{final } c \) **have** \( \text{simp}: c = \text{Skip} \) **by** (cases \( c \)) auto

**from** \( \text{labels-LDCFG-nodes prog } n \) \( c \) **obtain** \( n x \) **where** \( \text{simp}: n = \text{Node } n x \)

**and** \( \text{labels-nodes prog } n x \) \( \text{Skip} \) **by** (cases \( n \)) auto

**from** \( \text{labels-nodes prog } n x \) \( \text{Skip} \) **have** \( \text{prog} \) \( \vdash \) \( n x \) \( \leftarrow \\
\text{id} \rightarrow \text{(-Exit-)} \)

by (rule final-edge)

**then obtain** \( a \) **where** \( \text{valid-edge prog } a \) **and** \( \text{sourcenode } a = n x \)

**and** \( \text{kind } a = \uparrow \text{id} \) **and** \( \text{targetnode } a = \text{(-Exit-)} \)

by (auto simp: valid-edge-def)

**with** \( \text{labels-nodes prog } n x \) \( \text{Skip} \)

**show** \( \exists a. \text{ lift-valid-edge (valid-edge prog) sourcenode targetnode} \)

kind (\( \text{-Entry-} \)) (\( \text{-Exit-} \)) \( a \) \( \land \)

src \( a = n \) \( \land \) trg \( a = \text{LDCFG-node.Node (\( \text{-Exit-} \)) \( \land \) \( \text{kind } a = \uparrow \text{id} \) \)

by (rule-tac \( x=\text{(Node } n x,\uparrow \text{id},\text{Node (\( \text{-Exit-} \)))} \in \text{exI} \)

(auto intro (!: lve-edge simp: knd-def valid-edge-def))

qed

qed

### 4.4 Semantic Non-Interference for Weak Control Dependence

**lemmas** \( \text{WCDNonInterferenceGraph} = \)

\( \text{lift-PDG-weak-backward-slice} \) (OF \( \text{WWeakControlDependence.PDG-wed} \)

\text{WhileStrongPostdomination-aux} - \text{HighLowDistinct} \text{HighLowUNIV}]

**lemma** \( \text{WCDNonInterference:} \)

\( \text{NonInterferenceIntra src trg knd} \)

\( \text{(lift-valid-edge (valid-edge prog) sourcenode targetnode kind} \)

\( \text{(-Entry-)} \) (\( \text{-Exit-} \))\)

\( \text{NewEntry} (\text{lift-Def (Defs prog)} (\text{-Entry-}) (\text{-Exit-}) H L) \)

\( \text{(lift-Use (Uses prog)} (\text{-Entry-}) (\text{-Exit-}) H L ) \text{id} \)

\( \text{PDG.PDG-BS src trg} \)

\( \text{(lift-valid-edge (valid-edge prog) sourcenode targetnode kind} \)

\( \text{(-Entry-)} \) (\( \text{-Exit-} \))\)

\( \text{(lift-Def (Defs prog)} (\text{-Entry-}) (\text{-Exit-}) H L) \)

\( \text{(lift-Use (Uses prog)} (\text{-Entry-}) (\text{-Exit-}) H L ) \)

\( \text{(StrongPostdomination.weak-control-dependence src trg} \)

\( \text{(lift-valid-edge (valid-edge prog) sourcenode targetnode kind} \)

\( \text{(-Entry-)} \) (\( \text{-Exit-} \)) \text{NewExit}) \)

\( \text{reds (labels-LDCFG-nodes prog} \)

\( \text{NewExit H L (LDCFG-node.Node (\( \text{-Exit-} \))) (LDCFG-node.Node (\( \text{-Exit-} \)) \) final} \)

**proof** –

**from** \( \text{inner-node-exists obtain } n \) **where** \( \text{CFGExit.inner-node sourcenode targetnode} \)

\( \text{(valid-edge prog} \) (\( \text{-Entry-} \) \( \text{-Exit-} \)) \( n \) **by** blast

**then interpret** \( \text{NonInterferenceIntraGraph src trg knd} \)

\( \text{lift-valid-edge (valid-edge prog) sourcenode targetnode kind} \)

\( \text{(-Entry-)} \) (\( \text{-Exit-} \))
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 \langle c, s \rangle \rightarrow^* \langle c', s' \rangle
  thus \exists n' as. CFG.path src trg
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
  n as n' \land transfers (CFG.kinds knd as) s = s' \land
  preds (CFG.kinds knd as) s \land 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 \vdash nx \twoheadrightarrow (-Exit-)
  by (rule final-edge)
then obtain a where valid-edge prog a and sourcenode a = nx
  and kind a = \twoheadrightarrow id \land targetnode a = (-Exit-)
  by (auto simp:valid-edge-def)
with \{labels-nodes prog nx Skip\}
show \exists a. lift-valid-edge (valid-edge prog) sourcenode targetnode
  kind (-Entry-) (-Exit-) a \land

53
\( \text{src a} = n \land \text{trg a} = \text{LDCF\-node.Node (-Enter-)} \land \text{knd a} = \uparrow \text{id} \)

\( \text{by (rule-tac x:=(Node nx, \uparrow \text{id}, \text{Node (-Enter-)})) in exI} \)

(auto intro!ine-edge simp:knd-def valid-edge-def)

\( \text{qed} \)

\( \text{qed} \)

\( \text{end} \)

\( \text{end} \)

**References**


Formal proof development.