Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

April 19, 2020

Abstract

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

1 Introduction

Information Flow Control (IFC) encompasses algorithms which 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

description theory NonInterferenceIntra imports Slicing.Slice Slicing.CFGExit-uf begin
2.1 Assumptions of this Approach

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

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

locale NonInterferenceIntraGraph =
BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val
backward-slice +
CFGExit-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 ('Entry') and Def :: 'node ⇒ 'var set
and Use :: 'node ⇒ 'var set and state-val :: 'state ⇒ 'var ⇒ 'val
and backward-slice :: 'node set ⇒ 'node set
and Exit :: 'node ('Exit') +
fixes $H$ :: 'var set
fixes $L$ :: 'var set
fixes High :: 'node ('High')
fixes Low :: 'node ('Low')
assumes Entry-edge-Exit-or-High:
[valid-edge a; sourcenode a = (-Entry-)]
⇒ targetnode a = (-Exit-) ∨ targetnode a = (-High-)
and High-target-Entry-edge:
∃ a. valid-edge a ∧ sourcenode a = (-Entry-) ∧ targetnode a = (-High-)
and kind a = (λs, True)
and Entry-predecessor-of-High:
[valid-edge a; targetnode a = (-High-)] ⇒ sourcenode a = (-Entry-)
and Exit-edge-Entry-or-Low: [valid-edge a; targetnode a = (-Exit-)]
⇒ sourcenode a = (-Exit-) ∨ sourcenode a = (-Low-)
and Low-source-Exit-edge:
∃ a. valid-edge a ∧ sourcenode a = (-Low-) ∧ targetnode a = (-Exit-)
and kind a = (λs, True)
and Exit-successor-of-Low:
[valid-edge a; sourcenode a = (-Low-)] ⇒ targetnode a = (-Exit-)
and DefHigh: Def (-High-) = $H$
and UseHigh: Use (-High-) = $H$
and \textit{UseLow}: Use (-Low-) = L
and \textit{HighLowDistinct}: H \cap L = \{\}
and \textit{HighLowUNIV}: H \cup L = \text{UNIV}

\textbf{lemma \textit{Low-neq-Exit}:} assumes \(L \neq \{\}\) shows (-Low-) \neq (-Exit-)
\textbf{proof}
\hspace{1em}assume (-Low-) = (-Exit-)
\hspace{1em}have Use (-Exit-) = \{\} by fastforce
\hspace{1em}with UseLow \langle L \neq \{\} \rangle \langle (-Low-) = (-Exit-) \rangle show False by simp
\textbf{qed}

\textbf{lemma \textit{Entry-path-High-path}:}
\hspace{1em}assumes (-Entry-) \rightarrow∗ n and inner-node n
\hspace{1em}obtains \(a\) \(a'\) where as = \(a'\#as'\) and (-High-) \rightarrow∗ n and kind a' = (\lambda s. True)√
\textbf{proof(\textit{atomize-elim})}
\hspace{1em}from (-Entry-) \rightarrow∗ n (inner-node n)
\hspace{1em}show \exists a' as'. as = a'\#as' ∧ (-High-) \rightarrow∗ n ∧ kind a' = (\lambda s. True)√
\textbf{proof(\textit{induct} n \equiv (-Entry-) as n rule:\textit{path.induct})}
\hspace{1em}case (\textit{Cons-path} n'\equiv (-Entry-) as n rule:rev-induct)
\hspace{1em}from \langle n' \rightarrow∗ n'\rangle \langle inner-node n'\rangle have n' \neq (-Exit-)
\hspace{1em}by (fastforce simp:inner-node-def)
\hspace{1em}with \langle valid-edge a \rangle \langle targetnode a = n'\rangle \langle sourcenode a = (-Entry-) \rangle
\hspace{1em}have n' = (-High-) by (drule Entry-edge-Exit-or-High, auto)
\hspace{1em}from High-target-Entry-edge
\hspace{1em}obtain \(a'\) where valid-edge a' and sourcenode a' = (-Entry-)
\hspace{1em}and targetnode a' = (-High-) and kind a' = (\lambda s. True)√
\hspace{1em}by blast
\hspace{1em}with \langle valid-edge a \rangle \langle targetnode a = (-Entry-) \rangle \langle sourcenode a = n'' \rangle
\hspace{1em}⟨ n'' = (-High-) ⟩
\hspace{1em}have a = a' by (auto dest:edge-det)
\hspace{1em}with \langle n'' \rightarrow∗ n'\rangle \langle n'' = (-High-) \rangle \langle kind a' = (\lambda s. True)√ \rangle show ?case by blast
\textbf{qed fastforce}
\textbf{qed}

\textbf{lemma \textit{Exit-path-Low-path}:}
\hspace{1em}assumes \(n \rightarrow∗ (-Exit-)\) and inner-node n
\hspace{1em}obtains \(a\) \(a'\) where as = \(as'\#[a']\) and \(n \rightarrow∗ (-Low-)\)
\hspace{1em}and kind a' = (\lambda s. True)√
\textbf{proof(\textit{atomize-elim})}
\hspace{1em}from \langle n \rightarrow∗ (-Exit-) \rangle
\hspace{1em}show \exists as' a'. as = as'\#[a'] ∧ n \rightarrow∗ (-Low-) ∧ kind a' = (\lambda s. True)√
\textbf{proof(\textit{induct} as rule:rev-induct)
\hspace{1em}case Nil
\textbf{qed}
with \(\text{inner-node } n\) show \(?\)case by fastforce

next

case \((\text{snoc } a' \text{ as}')\)
from \(n \rightarrow^* \text{as}'[a] \rightarrow^* (-\text{Exit})\):
have \(n \rightarrow^* \text{as}' \text{ and valid-edge } a' \text{ and targetnode } a' = (-\text{Exit})\)
  by (auto elim: path-split-snoc)
{ assume source node \(a' = (-\text{Entry})\)
  with \(n \rightarrow^* \text{source node } a'\) have \(n = (-\text{Entry})\)
  by (blast intro!: path-Entry-target)
  with \(\text{inner-node } n\) have False by (simp add: inner-def)
} with \(\text{valid-edge } a'\) (targetnode \(a' = (-\text{Exit})\)) have source node \(a' = (-\text{Low})\)
  by (blast dest!: Exit-edge-Entry-or-Low)
from Low-source-Exit-edge
obtain \(ax\) where \(\text{valid-edge } ax \text{ and source node } ax = (-\text{Low})\)
  and targetnode \(ax = (-\text{Exit})\) and \(\text{kind } ax = (\lambda s. \text{True})\)
  by blast
  with \(\text{valid-edge } a'\) (targetnode \(a' = (-\text{Exit})\)) \(\langle\text{source node } a' = (-\text{Low})\rangle\)
  have \(a' = ax\) by (fastforce intro: edge-det)
  with \(n \rightarrow^* \text{source node } a'\) \(\langle\text{source node } a' = (-\text{Low})\rangle\) \(\langle\text{kind } ax = (\lambda s. \text{True})\rangle\)
  show \(?\)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 :: 'state \Rightarrow 'state \Rightarrow bool (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 \(V \in rv S \text{ (-Entry-)}\) \(\text{and (-High-)} \notin \text{backward-slice } S\)
  shows \(V \in L\)
proof –
from \( \langle V \in \text{rv} S \rangle \) obtain as \( n' \) where \( \text{-Entry-} \rightarrow n' \)
and \( n' \in \text{backward-slice} S \) and \( V \in \text{Use} n' \)
and \( \forall nx \in \text{set(source nodes as)}. \ V \notin \text{Def} nx \) by (erule \( \text{rvE} \))
from \( \langle \text{-Entry-} \rightarrow n' \rangle \) have valid-node \( n' \) by (rule \( \text{path-valid-node} \))
thus \( \text{thesis} \)
proof (cases \( n' \) rule: valid-node-cases)
  case Entry
  with \( \langle V \in \text{Use} n' \rangle \) have False by (simp add: Entry-empty)
  thus \( \text{thesis} \) by simp
next
  case Exit
  with \( \langle V \in \text{Use} n' \rangle \) have False by (simp add: Exit-empty)
  thus \( \text{thesis} \) by simp
next
  case inner
  with \( \langle \text{-Entry-} \rightarrow n' \rangle \) obtain \( a' \) as \( \text{as} = a'\#\text{as'} \)
  and \( \text{-High-} \rightarrow n' \) by (erule \( \text{Entry-path-High-path} \))
  from \( \langle \text{-Entry-} \rightarrow n' \rangle \) have sourcenode \( a' \) = \( \text{-Entry-} \) by (fastforce elim: path.cases)
  show \( \text{thesis} \)
  proof (cases \( \text{as'} = [] \))
    case True
    with \( \langle \text{-High-} \rightarrow n' \rangle \) have \( n' = \text{-High-} \) by fastforce
    with \( n' \in \text{backward-slice} S \) \( \langle \text{-High-} \rangle \notin \text{backward-slice} S \)
    have False by simp
    thus \( \text{thesis} \) by simp
  next
    case False
    with \( \langle \text{-High-} \rightarrow n' \rangle \) have \( \text{hd (sourcenodes as') = (-High-)} \)
    by (rule path-sourcenode)
    from False have \( \text{hd (sourcenodes as') \in \text{set (sourcenodes as')}} \)
    by (fastforce intro:hd-in-set simp:sourcenodes-def)
    with \( \text{as = a'\#as'} \) have \( \text{hd (sourcenodes as') \in \text{set (sourcenodes as')}} \)
    by (simp add:sourcenodes-def)
    with \( \langle \text{hd (sourcenodes as') = (-High-):} \forall nx \in \text{set (sourcenodes as)}. \ V \notin \text{Def} nx \rangle \)
    have \( \langle V \notin \text{Def (\text{-High-})} \rangle \) by fastforce
    hence \( \langle V \notin H \rangle \) by (simp add:DefHigh)
    thus \( \text{thesis} \) by (rule not-High-Low)
  qed
qed

lemma \( \text{lowEquivalence-relevant-nodes-Entry:} \)
assumes \( s \approx_L s' \) and \( \text{(-High-) \notin \text{backward-slice} S} \)
shows \( \forall V \in \text{rv} S \langle \text{-Entry-}, \text{state-val} s V = \text{state-val} s' V \rangle \)
proof

5
fix \( V \) assume \( V \in \text{rv} \ S \) (-Entry-)
with \((-\text{High-}) \notin \text{backward-slice} \ S \) have \( V \in L \) by \(-(\text{rule relevant-vars-Entry})\)
with \( s \approx_L \ s' \) show state-val \( s \) \( V \) = state-val \( s' \) \( V \) by \((\text{simp add:lowEquivalence-def})\)
qed

**Lemma:** \( \text{rv-Low-Use-Low:} \)

assumes \((-\text{Low-}) \in S \)
shows \([n \ -as\rightarrow* \ (-\text{Low-}); \ n \ -as'\rightarrow* \ (-\text{Low-}); \ \forall V \in rv \ S \ n. \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V; \ \text{preds} \ (\text{slice-kinds} \ S \ as) \ s; \ \text{preds} \ (\text{slice-kinds} \ S \ as') \ s' \] \implies \( \forall V \in Use \ (-\text{Low-}). \ \text{state-val} \ (\text{transfers} \ (\text{slice-kinds} \ S \ as) \ s) \ V = \text{state-val} \ (\text{transfers} \ (\text{slice-kinds} \ S \ as') \ s') \ V \)

**Proof:**

(induct \( n \) as \( n\equiv(-\text{Low-}) \) arbitrary:as' \( s \ s' \) rule:path.induct)

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

moreover
from \( (\text{valid-node} \ (-\text{Low-}) \ \text{have} \ (-\text{Low-}) \ -[]\rightarrow* \ (-\text{Low-}) \)
by \((\text{fastforce intro:path.empty-path})\)

moreover
from \( (\text{valid-node} \ (-\text{Low-}) \ ((-\text{Low-}) \in S \ \text{have} \ (-\text{Low-}) \in \text{backward-slice} \ S \)
by \((\text{fastforce intro:refl})\)

ultimately have \( V \in rv \ S \ (-\text{Low-})\)
by \((\text{fastforce intro:refl simp:sourcenodes-def})\) \}

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

show \?thesis

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

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

next

case False
from \((-\text{Low-}) \ -as'\rightarrow* \ (-\text{Low-}) \ \text{have} \ as' = [] \)

proof(induct \( n\equiv(-\text{Low-}) \ as' n'\equiv(-\text{Low-}) \) rule:path.induct)

case \( (\text{Cons-path} \ n'' \ as \ a) \)
from \( (\text{valid-edge} \ a) \ (\text{sourcenode} \ a = (-\text{Low-}) \)
have \( \text{targetnode} \ a = (-\text{Exit-}) \) by \(-(\text{rule Exit-successor-of-Low,simp+})\)

with \( \text{targetnode} \ a = n'' \) \( \langle n'' = a \ -as''\rightarrow* (-\text{Low-}) \rangle \)

have \( (-\text{Low-}) = (-\text{Exit-}) \) by \(-(\text{rule path-Exit-source,fastforce})\)

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

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

qed \( \text{simp} \)

with \( \forall V \in Use \ (-\text{Low-}). \ V \in rv \ S \ (-\text{Low-}) \)

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

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

qed

next

case \( (\text{Cons-path} \ n'' \ as \ a \ n) \)

note \( IH = \langle \forall \ s \ s'. \ \exists n''. \ n'' \ -as'\rightarrow* (-\text{Low-}); \ \forall V \in rv \ S \ n''. \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V; \)
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 ?case
proof(cases L = { })
  case True with UseLow show ?thesis by simp
next
  case False
  show ?thesis
proof(cases as')
  case Nil
  with \((n - \text{as'})\) have n = (\text{-Low-}) by fastforce
  with \(\text{valid-edge a} : \langle \text{source-node a} = n \rangle \) have targetnode a = (\text{-Exit-})
  by \(-\text{(rule Exit-successor-of-Low,simp+)}\)
  from Low-source-Exit-edge obtain ax where valid-edge ax
  and source-node ax = (\text{-Low-}) and target-node ax = (\text{-Exit-})
  and kind ax = (\text{ax} True) by blast
  from (valid-edge ax) \(\langle \text{source-node a} = n \rangle \) \(\langle \text{target-node a} = (\text{-Exit-}) \rangle \)
  (valid-edge ax) \(\langle \text{source-node ax} = (\text{-Low-}) \rangle \) \(\langle \text{target-node ax} = (\text{-Exit-}) \rangle \)
  have ax = ax \by\text{fastforce dest:edge-det}  
  with (kind ax = (\text{ax} True) False ax = (\text{ax} True) by simp
  with \(\langle \text{target-node a} = (\text{-Exit-}) \rangle \) \(\langle \text{target-node a} = n'' \rangle \) \(\langle n'' - \text{as}'\rangle (\text{-Low-})\)
  have (\text{-Low-}) by \(-\text{(rule path-Exit-source,auto)}\)
  with False have False by \(-\text{(drule Low-neq-Exit,simp)}\)
  thus ?thesis by simp
next
  case (\text{Cons ax asx})
  with \((n - \text{as'})\) have n = source-node ax and valid-edge ax
  and target-node ax = asx \by\text{(auto elim:path-split-Cons)}
  show ?thesis
proof(cases target-node ax = n''\)
  case True
  with \(\langle \text{target-node ax} = \text{asx} \rangle (\text{-Low-})\) have n'' - asx \by\text{simp}
  from (valid-edge ax) \(\langle \text{valid-edge a} \rangle \) \(\langle n = \text{source-node ax} \rangle \)
  \(\langle \text{source-node a} = n \rangle \)
  True \(\langle \text{target-node a} = n'' \rangle\) have ax = a \by\text{fastforce intro:edge-det}
  from \(\langle \text{preds (slice-kinds S (\text{a'}) as}) s \rangle\)
  \(\langle \text{preds (slice-kinds S as)} \rangle \) \(\langle \text{transfer (slice-kind S a)} s \rangle\)
  \by\text{simp add:slice-kinds-def}
  from \(\langle \text{preds (slice-kinds S as') s'} \rangle \) \(\langle \text{Cons (ax a)} \rangle\)
  \(\langle \text{have pred2:preds (slice-kinds S asx)} \rangle\)
  \(\langle \text{(transfer (slice-kind S a)} s' \rangle\)
  \by\text{simp add:slice-kinds-def}
  from \(\langle \text{valid-edge a} \rangle \) \(\langle \text{source-node a} = n \rangle \) \(\langle \text{target-node a} = n'' \rangle\)
  \(\langle \text{preds (slice-kinds S (\text{a'}) as}) s \rangle \) \(\langle \text{preds (slice-kinds S as') s'} \rangle\)
  \(\langle \text{ax a} \rangle \) \(\langle \text{Cons (ax a)} \rangle \) \(\forall V \in \text{rv S n}\)
  \(\langle \text{state-val s V = state-val s'} V \rangle\)
  have \(\forall V \in \text{rv S n''}, \text{state-val} \ (\text{transfer (slice-kind S a)} s) V = \text{state-val} \ (\text{transfer (slice-kind S a)} s') V\)
  by \(-\text{(rule rv-edge-slice-kinds,auto)}\)
  from IH[\(\langle n'' - \text{as}'\rangle (\text{-Low-})\) \(\langle \text{this pred1 pred2}\rangle\)
\[ Cons \; \langle ax = a \rangle \; \text{show \; ?thesis \; by (simp \; add: \; slice-kinds-def) \; next} \]
\[
\text{case \; False} \n\]
\[
\text{with \; (valid-edge \; a) \; \langle \text{valid-edge} \; ax \rangle \; \langle \text{source} \; a = n \rangle \; \langle \text{source} \; ax \rangle \; \langle \text{target} \; a = n' \rangle \; \langle \text{preds} \; (\text{slice-kinds} \; S \; (\text{a}#\text{as})) \; s \rangle \; \langle \text{preds} \; (\text{slice-kinds} \; S \; (\text{as}') \; s') \; \text{Cons} \rangle \; \langle \forall \; V \in \text{rv} \; S \; n. \; \text{state-val} \; s \; V = \text{state-val} \; s' \; V \rangle \; \text{have \; False \; by \; -(rule rv-branching-edges-slice-kinds-False, auto) \; thus \; ?thesis \; by \; simp} \]
\text{qed} \quad \text{qed} \quad \text{qed} \quad \text{2.3 \; The \; Correctness \; Proofs} \]

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

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

\text{lemma \; nonInterference-path-to-Low:} \n\text{assumes \; s \; \approx_L \; s' \; \text{and \; (-High-) \notin \text{backward-slice} \; S \; \text{and \; (-Low-) \in S} \; \text{and \; (-Entry-) \rightarrow \text{as} \rightarrow \ast \; (-Low-) \; \text{and \; preds \; (kinds as) \; s} \; \text{and \; (-Entry-) \rightarrow \text{as}' \rightarrow \ast \; (-Low-) \; \text{and \; preds \; (kinds as') \; s'}}\;
\text{shows \; transfers \; (kinds as) \; s \; \approx_L \; \text{transfers \; (kinds as') \; s'}}\;
\text{proof \; -} \n\text{from \; (-Entry-) \rightarrow \text{as} \rightarrow \ast \; (-Low-) \; \langle \text{preds} \; (\text{kinds as}) \; s \rangle \; \langle (-\text{Low}) \in S \rangle \; \text{obtain \; asz \; where \; preds \; (slice-kinds \; S \; \text{asz}) \; s} \; \text{and \; \forall \; V \in \text{Use} \; (-\text{Low}.) \; \text{state-val} (\text{transfers} \; (\text{slice-kinds} \; S \; \text{asz}) \; s) \; V = \text{state-val} (\text{transfers} \; (\text{kinds as}) \; s) \; V} \; \text{and \; slice-edges \; S \; as = \; slice-edges \; S \; \text{asz}} \; \text{and \; (-Entry-) \rightarrow \text{as} \rightarrow \ast \; (-\text{Low}) \; \text{by (erule fundamental-property-of-static-slicing)}} \; \text{from \; (-Entry-) \rightarrow \text{as}' \rightarrow \ast \; (-\text{Low}) \; \langle \text{preds} \; (\text{kinds as'}) \; s' \rangle \; \langle (-\text{Low}) \in S \rangle \; \text{obtain \; asz' \; where \; preds \; (slice-kinds \; S \; \text{asz'}) \; s'} \; \text{and \; \forall \; V \in \text{Use} \; (-\text{Low}.) \; \text{state-val} (\text{transfers} \; (\text{slice-kinds} \; S \; \text{asz'}) \; s') \; V = \text{state-val} (\text{transfers} \; (\text{kinds as'}) \; s') \; V} \; \text{and \; slice-edges \; S \; as' = \; slice-edges \; S \; \text{asz'}} \; \text{and \; (-Entry-) \rightarrow \text{as} \rightarrow \ast \; (-\text{Low}) \; \text{by (erule fundamental-property-of-static-slicing)}} \; \text{from \; (s \; \approx_L \; s') \; \langle (-\text{High}) \notin \text{backward-slice} \; S \rangle \; \text{have \; \forall \; V \in \text{rv} \; S \; (-\text{Entry}.) \; \text{state-val} \; s \; V = \text{state-val} \; s' \; V} \; \text{by (rule lowEquivalence-relevant-nodes-Entry)}} \; \text{with \; (-Entry-) \rightarrow \text{as} \rightarrow \ast \; (-\text{Low}) \; \langle (-\text{Entry}) \rightarrow \text{as}' \rightarrow \ast \; (-\text{Low}) \rangle \; \langle (-\text{Low}) \in S \rangle \; \langle \text{preds} \; (\text{slice-kinds} \; S \; \text{asz}) \; s \rangle \; \langle \text{preds} \; (\text{slice-kinds} \; S \; \text{asz'}) \; s' \rangle \; \text{have \; \forall \; V \in \text{Use} \; (-\text{Low}) \; \text{state-val} (\text{transfers} \; (\text{slice-kinds} \; S \; \text{asz}) \; s) \; V = \text{state-val} (\text{transfers} \; (\text{slice-kinds} \; S \; \text{asz'}) \; s') \; V} \; \text{by -(rule rv-Low-Use-Low, auto) \; qed} \]
with \( \forall V \in \text{Use (}\text{Low}-\text{). state-val(transfers (slice-kinds \text{S} \text{ax} z) s)} \) \( V = \text{state-val(transfers (kinds as) s)} \) \( V \)
\( \forall V \in \text{Use (}\text{Low}-\text{). state-val (transfers (slice-kinds \text{S} \text{ax} z') s') V = state-val (transfers (kinds as') s') V \)
show ?thesis by(auto simp:lowEquivalence-def UseLow)
qed

theorem nonInterference-path:
assumes \( s \approx_{\text{L}} s' \) and \( (\text{High}-) \notin \text{backward-slice S and (Low-)} \in S \)
and \( (\text{Entry}-) -\text{as} \rightarrow\text{ (Exit-)} \) and \( \text{preds (kinds as) s} \)
and \( (\text{Entry}-) -\text{as'} \rightarrow\text{ (Exit-)} \) and \( \text{preds (kinds as') s'} \)
shows transfers (kinds as) s \( \approx_{\text{L}} \) transfers (kinds as') s'
proof -
from \( (\text{Entry}-) -\text{as} \rightarrow\text{ (Exit-)} \) obtain \( x \text{xs} \) where \( \text{as} = x\#\text{xs} \)
and \( (\text{Entry}-) = \text{sourcenode x and valid-edge x} \)
and \( \text{targetnode} x = \text{xs}\rightarrow\text{ (Exit-)} \)
apply(cases \( \text{as} = [] \))
apply(simp,erule empty-path-nodes,erule Entry-noteq-Exit,simp)
by(erule path-split-Cons)
from \( \forall \text{valid-edge x} \) have \( \text{valid-node (targetnode x) by simp} \)
hence \( \text{inner-node (targetnode x)} \)
proof(cases rule:valid-node-cases)
  case \( \text{Entry} \)
  with \( \forall \text{valid-edge x} \) have \( \text{False by(rule Entry-target)} \)
  thus ?thesis by simp
next
  case \( \text{Exit} \)
  with \( \forall \text{targetnode x = xs}\rightarrow\text{ (Exit-)} \) have \( \text{xs} = [] \)
  by -(rule path-Exit-source,simp)
from \( \text{Entry}\text{-Exit}\text{-edge obtain} \ z \) where \( \text{valid-edge z} \)
and \( \text{sourcenode} z = (\text{Entry-}) \) and \( \text{targetnode} z = (\text{Exit-}) \)
and \( \lambda z. \text{False} \) by blast
from \( \forall \text{valid-edge z} \) \( (\text{Entry-}) = \text{sourcenode x} \)
\( \text{(sourcenode} z = (\text{Entry-}) \) \( \text{Exit} \) \( \text{targetnode} z = (\text{Exit-}) \) \)
have \( \text{x = z by(fastforce intro:edge-det)} \)
with \( \text{preds (kinds as) s} \) \( \langle as = x\#\text{xs} \rangle \) \( \langle \text{xs} = [] \rangle \) \( \langle \text{kind} z = (\lambda z. \text{False}) \rangle \)
have \( \text{False by(simp add:kinds-def)} \)
thus ?thesis by simp
qed simp

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

```plaintext
locale NonInterferenceIntra = 
NonInterferenceIntraGraph sourcenode targetnode kind valid-edge Entry 
  Def Use state-val backward-slice Exit H L High Low + 
BackwardSlice-ref sourcenode targetnode kind valid-edge Entry Def Use state-val 
  backward-slice sem identifies
```

```plaintext
for sourcenode :: 'edge ⇒ 'node and targetnode :: 'edge ⇒ 'node 
  and kind :: 'edge ⇒ 'state edge-kind and valid-edge :: 'edge ⇒ bool 
  and Entry :: 'node ('(Entry')::) and Def :: 'node ⇒ 'var set 
  and Use :: 'node ⇒ 'var set and state-val :: 'state ⇒ 'var ⇒ 'val 
  and backward-slice :: 'node set ⇒ 'node set 
  and sem :: 'com ⇒ 'state ⇒ 'com ⇒ 'state ⇒ bool 
  ((((1(\_,\_/)) ⇒/ (1(\_,\_/))) [0,0,0,0] 81) 
  and identifies :: 'node ⇒ 'com ⇒ bool (\_ ≈ [51, 0] 80) 
  and Exit :: 'node ('(Exit')::) 
  and H :: 'var set and L :: 'var set 
  and High :: 'node ('(High')::) and Low :: 'node ('(Low')::) + 
  fixes final :: 'com ⇒ bool 
  assumes final-edge-Low: [final c; n ≈ c] 
  ⇒ ∃ a. valid-edge a ∧ sourcenode a = n ∧ targetnode a = (Low-) ∧ kind a = 
  ↑id 
begin 

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

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

```plaintext
theorem nonInterference: 
  assumes s₁ ≈₄, s₂ and (-High-) ∉ backward-slice S and (-Low-) ∈ S 
  and valid-edge a and sourcenode a = (-High-) and targetnode a = n 
  and kind a = (λs. True), and n ≈ c and final e' 
  and ⟨c,s₁⟩ ⇒ ⟨c',s₁'⟩ and ⟨c,s₂⟩ ⇒ ⟨c',s₂'⟩ 
  shows s₁ ≈₄, s₂
proof – 
from High-target-Entry-edge obtain ax where valid-edge ax 
  and sourcenode ax = (Entry-) and targetnode ax = (-High-) 
  and kind ax = (λs. True), by blast 
from ⟨n ≈ c⟩ ⟨c,s₁⟩ ⇒ ⟨c',s₁'⟩ 
obtain n₁ as₁ where n →₄ as₁ and transfers (kinds as₁) s₁ = s₁' 
  and preds (kinds as₁) s₁ and n₁ ≈ c' 
  by (fastforce dest: fundamental-property) 
from ⟨n →₄ as₁ ⟩ ⟨valid-edge a⟩ ⟨sourcecenode a = (-High-)⟩ ⟨targetnode a = n⟩ 
have ⟨(-High-) →₄ a as₁ →₄ n₁⟩ by (rule Cons-path) 
from ⟨final e'⟩ ⟨n₁ ≈ c'⟩ 
obtain a₁ where valid-edge a₁ and sourcenode a₁ = n₁
```
```
\[\text{and \ targetnode} \ a_1 = (-\text{Low}) \ \text{and \ kind} \ a_1 = \uparrow\text{id} \ \text{by (fastforce dest: final-edge-Low)}\]
\[\text{hence} \ n_1 \ [a_1] \rightarrow \ast \ (-\text{Low}) \ \text{by (fastforce intro: path-edge)}\]
\[\text{with} \ (-\text{High}) - a\#(a\#a_1) \rightarrow (a\#a_1) \rightarrow (\text{Low}) \ \text{by (rule Cons-path)}\]
\[\text{by (rule path-Append)}\]
\[\text{with} \ \text{valid-edge ax) (sourcenode ax = (-Entry)) \ (targetnode ax = (-High))}\]
\[\text{have} \ (-\text{Entry}) - ax\#((a\#a_1)@[a_1]) \rightarrow (\text{Low}) \ \text{by (rule Cons-path)}\]
\[\text{from} \ (\text{kind} \ ax = (\lambda s. \ True)) \ \text{by (fastforce intro: path-edge)}\]
\[\text{have} \ (-\text{High}) - a\#a_2 \rightarrow (\text{Low}) \ \text{by (rule Cons-path)}\]
\[\text{from} \ (\text{final} \ c') \ \langle a_2 \triangleq c' \rangle \ \text{by (rule path-Append)}\]
\[\text{with} \ \text{valid-edge ax) (sourcenode ax = (-Entry)) \ (targetnode ax = (-High))}\]
\[\text{have} \ (-\text{Entry}) - ax\#((a\#a_2)@[a_2]) \rightarrow (\text{Low}) \ \text{by (rule Cons-path)}\]
\[\text{from} \ (\text{kind} \ ax = (\lambda s. \ True)) \ \text{by (fastforce intro: path-edge)}\]
\[\text{have} \ (-\text{High}) - a\#a_2 \rightarrow (\text{Low}) \ \text{by (rule Cons-path)}\]
\[\text{by (simp add: kinds-def} \ \text{preds-split)}\]
\[\text{from} \ (a_n \triangleq c) \ \langle (c, a_2) \Rightarrow (c', a_2') \rangle \ \text{by (rule nonInterference-path-to-Low)}\]
\[\langle (-\text{Entry}) - ax\#((a\#a_1)@[a_1]) \rightarrow (\text{Low}) \ \text{by (rule nonInterference-path-to-Low)}\]
\[\langle (-\text{Entry}) - ax\#((a\#a_2)@[a_2]) \rightarrow (\text{Low}) \ \text{by (rule nonInterference-path-to-Low)}\]
\[\langle \text{transfers} \ (\text{kinds} \ ax\#((a\#a_1)@[a_1])) \ s_1 \ \text{by (rule nonInterference-path-to-Low)}\]
\[\text{show} \ \$\text{thesis} \ \text{by (simp add: kinds-def} \ \text{transfers-split)}\]
\[\text{qed}\]

end

end

3 Framework Graph Lifting for Noninterference

theory LiftingIntra

imports NonInterferenceIntra Slicing.CDepInstantiations
In this section, we show how a valid CFG from the slicing framework in [8] can be lifted to fulfill all properties of the NonInterferenceIntraGraph locale. Basically, we redefine the hitherto existing Entry and Exit nodes as new High and Low nodes, and introduce two new nodes NewEntry and NewExit. Then, we have to lift all functions to operate on this new graph.

3.1 Liftings

3.1.1 The datatypes

datatype 'node LDCFG-node = Node 'node 
  | NewEntry 
  | NewExit

type-synonym ('edge,'node,'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 ⇒ ('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

\[ \text{inductive-set lift-Def-set :: } ('\text{node} \Rightarrow '\text{var set}) \Rightarrow '\text{node} \Rightarrow '\text{node} \Rightarrow '\text{var set} \Rightarrow '\text{var set} \Rightarrow ('\text{node LDCFG-node} \times '\text{var}) \text{ set} \]

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

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

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

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

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

\[ \text{inductive-set lift-Use-set :: } ('\text{node} \Rightarrow '\text{var set}) \Rightarrow '\text{node} \Rightarrow '\text{node} \Rightarrow '\var set \Rightarrow '\var set \Rightarrow '\text{node LDCFG-node} \Rightarrow '\text{var set} \]

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

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

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

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

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

where lift-Use Use E X H L n \equiv \{ 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

abbreviation src :: ('\text{edge},'\text{node},'\text{state}) LDCFG-edge \Rightarrow 'node LDCFG-node

where src a \equiv \text{fst } a

abbreviation trg :: ('\text{edge},'\text{node},'\text{state}) LDCFG-edge \Rightarrow 'node LDCFG-node

where trg a \equiv \text{snd}(\text{snd } a)

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

where knd a \equiv \text{fst}(\text{snd } a)
lemma lift-CFG:
assumes \( wf : \text{CFGExit-wf} \text{ sourcenode targetnode kind valid-edge Entry Def Use} \)
state-val Exit
shows \( \text{CFG} \text{ src trg} \)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
proof –
interpret \( \text{CFGExit-wf} \text{ sourcenode targetnode kind valid-edge Entry Def Use} \)
state-val Exit
by (rule \( wf \))
show \( \text{?thesis} \)
proof
fix \( a \) assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit \( a \)
and \( \text{trg} \ a = \text{NewEntry} \)
thus \( \text{False} \) 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 \( \text{src} \ a = \text{src} \ a' \text{ and trg} \ a = \text{trg} \ a' \)
thus \( a = a' \)
proof (induct rule:lift-valid-edge.induct)

case lve-edge thus \( \text{?case} \) by (erule lift-valid-edge.cases, auto dest:edge-det)
qed (auto elim:lift-valid-edge.cases)
qed

lemma lift-CFG-wf:
assumes \( wf : \text{CFGExit-wf} \text{ sourcenode targetnode kind valid-edge Entry Def Use} \)
state-val Exit
shows \( \text{CFG-wf src trg knd} \)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val
proof –
interpret \( \text{CFGExit-wf} \text{ sourcenode targetnode kind valid-edge Entry Def Use} \)
state-val Exit
by (rule \( wf \))
interpret \( \text{CFG:CFG} \text{ src trg knd} \)
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
by (fastforce intro:lift-CFG \( wf \))
show \( \text{?thesis} \)
proof
show lift-Def Def Entry Exit H L NewEntry = \{ \} \land
lift-Use Use Entry Exit H L NewEntry = \{ \}
by (fastforce elim:lift-Use-set.cases lift-Def-set.cases)
next
fix \( a \ V s \)
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V /∈ lift-Def Def Entry Exit H L (src a) and pred (kind a) s
thus state-val (transfer (kind a) s) V = state-val s V
proof (induct rule: lift-valid-edge.induct)
case lee-edge
thus ‟case by (fastforce intro: CFG-edge-no-Def-equal dest: lift-Def-node[af -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
∀ V ∈ 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 ∀ V ∈ 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 ∈ 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 (lee-edge a e)
show ‟thesis by simp
proof (cases Node (sourcenode a) = Node Entry)
case True
hence sourcenode a = Entry by simp
from Entry-Exit-edge obtain a’ where valid-edge a’
and sourcenode a’ = Entry and targetnode a’ = Exit
and kind a’ = (λs. False) by blast
have ∃ Q. kind a = (Q) by blast
proof (cases targetnode a = Exit)
case True
with ⟨valid-edge a⟩ ⟨valid-edge a’⟩ ⟨sourcenode a = Entry⟩
⟨sourcenode a’ = Entry⟩ ⟨targetnode a’ = Exit⟩
have a = a’ by (fastforce dest: edge-det)
with ⟨kind a’ = (As. False)⟩ show ‟thesis by simp
next
case False
with ⟨valid-edge a⟩ ⟨valid-edge a’⟩ ⟨sourcenode a = Entry⟩
⟨sourcenode a’ = Entry⟩ ⟨targetnode a’ = Exit⟩
show ‟thesis by (auto dest: deterministic)
qed
from True (V ∈ lift-Def Entry Exit H L (src e)) Entry-empty
e = (Node (sourcenode a), kind a, Node (targetnode a))
have V ∈ H by (fastforce elim: lift-Def-set.cases)
from True (e = (Node (sourcenode a), kind a, Node (targetnode a)))
⟨sourcenode a ≠ Entry ∨ targetnode a ≠ Exit⟩
have ∀ V ∈ H. V ∈ lift-Use Entry Exit H L (src e)
by (fastforce intro: lift-Use-High)

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

\[ \text{state-val} \ s \ V = \text{state-val} \ s' \ V \langle V \in H \rangle \]

have \( \text{state-val} \ s \ V = \text{state-val} \ s' \ V \) by simp

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

\( \exists Q. \text{kind} \ a = (Q) \sqrt {\langle V \in H \rangle} \)

show ?thesis by (fastforce simp: knd-def)

next

case False

\{ fix \ V' \ assume \ V' \in \text{Use} \ (\text{sourcenode} \ a) \)

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

have \( V' \in \text{lift-Use} \ \text{Use} \ \text{Entry} \ \text{Exit} \ H \ L \ (\text{src} \ e) \) by (fastforce intro: lift-Use-node)

\}

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

\[ \text{state-val} \ s \ V = \text{state-val} \ s' \ V \]

by fastforce

from (valid-edge a) this \( \langle \text{pred} \ (\text{knd} \ e) \ s \rangle \langle \text{pred} \ (\text{knd} \ e) \ s' \rangle \)

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

have \( \forall V \in \text{Def} \ (\text{sourcenode} \ a), \text{state-val} \ (\text{transfer} \ (\text{knd} \ a) \ s) \ V = \text{state-val} \ (\text{transfer} \ (\text{knd} \ a) \ s') \ V \)

by \( \neg (\text{erule CFG-edge-transfer-uses-only-Use}, \text{auto simp:knd-def}) \)

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

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

have \( V \in \text{Def} \ (\text{sourcenode} \ a) \) by (fastforce elim: lift-Def-set.cases)

with \( \forall V \in \text{Def} \ (\text{sourcenode} \ a), \text{state-val} \ (\text{transfer} \ (\text{knd} \ a) \ s) \ V = \text{state-val} \ (\text{transfer} \ (\text{knd} \ a) \ s') \ V \)

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

show ?thesis by (simp add: knd-def)

qed

next

case (lve-Entry-edge e)

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

\( e = (\text{NewEntry}, (\lambda s. \text{True}) \sqrt {\langle \text{Node} \ \text{Entry} \rangle}) \)

have False by (fastforce elim: lift-Def-set.cases)

thus ?case by simp

next

case (lve-Exit-edge e)

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

\( e = (\text{Node} \ \text{Exit}, (\lambda s. \text{True}) \sqrt {\langle \text{NewExit} \rangle}) \)

have False by (fastforce elim: lift-Def-set.cases intro!: Entry-noteq-Exit simp: Exit-empty)

thus ?case by simp

qed (simp add: knd-def)

next

fix \ a \ s \ s' \)

assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and pred (knd a) s
and ∀ V∈lift-Use Use Entry Exit H L (src a). state-val s V = state-val s’ V
thus pred (knd a) s’
by (induct rule: lift-valid-edge.induct,
   auto elim!: CFG-edge-Uses-pred-equal dest: lift-Use-node simp: knd-def)

next
fix a a’
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a’
and src a = src a’ and trg a ≠ trg a’
thus ∃ Q Q’. knd a = (Q) ∨ ∧ knd a’ = (Q’) ∨ ∧
   (∀ s. (Q s → ¬ Q’ s) ∧ (Q’ s → ¬ Q s))
proof (induct rule: lift-valid-edge.induct)
   case (lve-edge a e)
   from ⟨lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a’
    valid-edge a; e = (Node (sourcenode a), kind a, Node (targetnode a))⟩;
   ⟨src e = src a’; trg e ≠ trg a’⟩
   show ?case
   proof (induct rule: lift-valid-edge.induct)
   case lve-edge thus ?case by (auto dest: deterministic simp: knd-def)
next
   case (lve-Exit-edge e’)
   from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩;
   ⟨e’ = (Node Exit, λ s. True) ∨, NewExit)⟩; ⟨src e = src e’⟩
   have sourcenode a = Exit by simp
   with valid-edge a; have False by (rule Exit-source)
   thus ?case by simp
   qed auto
   qed (fastforce elim: lift-valid-edge.cases simp: knd-def)+
   qed
   qed

lemma lift-CFGExit:
assumes wf: CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
state-val Exit
shows CFGExit src trg knd
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   NewEntry NewExit
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
state-val Exit
by (rule wf)
interpret CFG: CFG src trg knd
   lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
   by (fastforce intro: lift-CFG wf)
show ?thesis
proof
   fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a

18
and src a = NewExit
thus False by (fastforce elim: lift-valid-edge.cases)
next
from lve-Entry-Exit-edge
show \exists a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a \land
   src a = NewEntry \land trg a = NewExit \land knd a = (\lambda s. False) √
by (fastforce simp: knd-def)
qed
qed

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

3.2.2 Lifting wod-backward-slice

lemma lift-wod-backward-slice:
fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
and Def and Use and H and L
defines lve: lve \equiv lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and \ld: ldef \equiv lift-Def Def Entry Exit H L
and \luse: luse \equiv lift-Use Use Entry Exit H L
assumes wf: CFGExit_wf sourcenode targetnode kind valid-edge Entry Def Use
state-val Exit
and H \cap L = {} and H \cup L = UNIV
shows NonInterferenceIntraGraph src try knd lve NewEntry ldef luse state-val
\[
(CFG-wf \cdot wod-backward-slice \ src \ trg \ lve \ lDef \ lUse) \\
NewExit H L (Node \ Entry) \ (Node \ Exit)
\]

**proof**

**interpret** \(CFGExit-wf\) \(\text{source}\) \(\text{target}\) \(kind\) \(valid-edge\) \(Entry\) \(Def\) \(Use\) \\
state-val \(Exit\)

by (rule \(wf\))

**interpret** \(CFGExit-wf\): 
\(CFGExit-wf\) \(\text{src}\) \(\text{trg}\) \(\text{knd}\) \(lve\) \(\text{NewEntry}\) \\
by (fastforce intro: \(\text{lift-CFGExit-wf}\) \(wf\) simp: \(lve\) \(\text{lDef}\) \(\text{lUse}\))

**from** \(wf\) \(lve\) \(\text{have} \ CFG:CFG \ src \ trg \ lve \ \text{NewEntry}\) \\
by (fastforce intro: \(\text{lift-CFG}\))

**from** \(wf\) \(lve\) \(\text{lDef}\) \(lUse\) \(\text{have} \ CFG-wf:CFG-wf \ src \ trg \ \text{trg} \ \text{knd} \ lve \ \text{NewEntry}\) \\
\(\text{lDef}\) \(\text{lUse}\) \(\text{state-val}\) \\
by (fastforce intro: \(\text{lift-CFG-wf}\))

**show** \(\text{?thesis}\)

**proof**

fix \(n\) \(S\)

assume \(n \in CFG-wf \cdot wod-backward-slice \ src \ trg \ lve \ lDef \ lUse\) \(S\) \\
with \(CFG-wf\) \(\text{show}\) \(CFG\).\(\text{valid-node}\) \(\text{src}\) \(\text{trg}\) \(\text{lve}\) \(n\) \\
by -(rule \(CFG-wf\).\(\text{wod-backward-slice-valid-node}\))

**next**

fix \(n\) \(S\) \(\text{assume}\) \(CFG\).\(\text{valid-node}\) \(\text{src}\) \(\text{trg}\) \(\text{lve}\) \(n\) and \(n \in S\) \\
with \(CFG-wf\) \(\text{show}\) \(n \in CFG-wf \cdot wod-backward-slice \ src \ trg \ lve \ lDef \ lUse\) \(S\) \\
by -(rule \(CFG-wf\).\(\text{refl}\))

**next**

fix \(n'\) \(S\) \(n\) \(V\)

assume \(n' \in CFG-wf \cdot wod-backward-slice \ src \ trg \ lve \ lDef \ lUse\) \(S\) \\
and \(CFG-wf\).\(\text{data-dependence}\) \(\text{src}\) \(\text{trg}\) \(\text{lve}\) \(lDef\) \(lUse\) \(n\) \(V\) \(n'\) \\
with \(CFG-wf\) \(\text{show}\) \(n \in CFG-wf \cdot wod-backward-slice \ src \ trg \ lve \ lDef \ lUse\) \(S\) \\
by -(rule \(CFG-wf\).\(\text{dd-closed}\))

**next**

fix \(n\) \(S\)

from \(CFG-wf\)

have \((\exists m. \ (CFG\).\(\text{obs}\) \(\text{src}\) \(\text{trg}\) \(lve\) \(n\))\) \\
\((CFG-wf \cdot wod-backward-slice \ src \ trg \ lve \ lDef \ lUse\) \(S\)) = \(\{m\}\) \(\vee\) \\
\((CFG\).\(\text{obs}\) \(\text{src}\) \(\text{trg}\) \(lve\) \(n\) \(\text{CFG-wf} \cdot \text{wod-backward-slice} \ src \ trg \ lve \ lDef \ lUse\) \(S\)) = \(
\{\}
\)

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

thus \(\text{finite}\) \\
\((CFG\).\(\text{obs}\) \(\text{src}\) \(\text{trg}\) \(lve\) \(n\) \(\text{CFG-wf} \cdot \text{wod-backward-slice} \ src \ trg \ lve \ lDef \ lUse\) \(S\)) \\
by fastforce

**next**

fix \(n\) \(S\)

from \(CFG-wf\)

have \((\exists m. \ (CFG\).\(\text{obs}\) \(\text{src}\) \(\text{trg}\) \(lve\) \(n\))\) \\
\((CFG-wf \cdot wod-backward-slice \ src \ trg \ lve \ lDef \ lUse\) \(S\)) = \(\{m\}\) \(\vee\) \\
\((CFG\).\(\text{obs}\) \(\text{src}\) \(\text{trg}\) \(lve\) \(n\) \(\text{CFG-wf} \cdot \text{wod-backward-slice} \ src \ trg \ lve \ lDef \ lUse\) \(S\)) = \(
\{\}
\)

by (rule \(CFG-wf\).\(\text{obs-singleton}\))
thus \( \text{card} \ (CFG \_\text{obs} \ src \ trg \ lve \ n) \)
\[
= (CFG \_\text{wf} \_\text{wod-backward-slice} \ src \ trg \ lve \ lDef \ lUse \ S) \leq 1
\]
by fastforce

next
fix \( a \) assume \( lve \ a \) and \( src \ a = \text{NewEntry} \)
with \( lve \) show \( trg \ a = \text{NewExit} \lor trg \ a = \text{Node Entry} \)
by (fastforce elim:lift-valid-edge.cases)

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

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

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

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

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

next
from \( lDef \) show \( lDef \ (\text{Node Entry}) = H \)
by (fastforce elim:lift-def-set.cases intro:lift-Def-High)

next
from \( \text{Entry-noteq-Exit} \) lUse show \( lUse \ (\text{Node Entry}) = H \)
by (fastforce elim:lift-use-set.cases intro:lift-Use-High)

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

\label{sec:lifting}

3.2.3 Lifting PDG-BS with standard-control-dependence

lemma lift-Postdomination:
assumes af:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
and pd:Postdomination sourcenode targetnode kind valid-edge Entry Exit
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
shows Postdomination src trg knp
(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 knp
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry Lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val NewExit
by (fastforce intro:lift-CFGExit-wf wf)
from wf have CFG:CFG src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
by (rule lift-CFG)
show ?thesis
proof
fix n assume CFG:valid-node src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
show ∃ as. CFG: path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry as n
proof (cases n)
case NewEntry
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry, (λ s. False), NewExit) by (fastforce intro: lve-Entry-Exit-edge)
with NewEntry have CFG: path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
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, (λ s. False), NewExit) by (fastforce intro: lve-Entry-Exit-edge)
with NewExit have CFG: path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewExit [(NewEntry, (λ s. False), NewExit)] n
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,(\x. 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,(\x. 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 ⟨ inner obtain ax where valid-edge ax and inner-node (sourcenode ax) ⟩
    and targetnode ax = Exit by (erule inner-node-Exit-edge)
  hence lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (Node (sourcenode ax), kind ax, Node Exit)
    by (auto intro: lve-Entry-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,(\x. True),Node Entry) by (fastforce intro: lve-Entry-edge)
  from ⟨ inner-node (sourcenode ax) ⟩ have valid-node (sourcenode ax)
    by (rule inner-is-valid)
  then obtain asx where Entry − asx →∗ sourcenode ax
    by (fastforce dest: Entry-path)
  from this ⟨ valid-edge ax ⟩ have ∃ es. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) es (Node (sourcenode ax))
  proof (induct asx arbitrary: ax rule: rev-induct)
    case Nil
      from ⟨ Entry − [] →∗ sourcenode ax ⟩ have sourcenode ax = Entry by fastforce
    hence CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Entry) [] (Node (sourcenode ax))
      apply simp apply (rule CFG.empty-path[OF CFG])
      by (auto intro: lve-Entry-edge simp: CFG.valid-node-def[OF CFG])
    thus ?case by blast
  next
case (snoc x xs)
  note IH = ⟨ Entry − xs →∗ sourcenode ax; valid-edge ax ⟩ ⇒
    ∃ es. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node Entry) es (Node (sourcenode ax))
  from ⟨ Entry − xs@[x] →∗ sourcenode ax ⟩ have Entry − xs →∗ sourcenode x and valid-edge x

23
\[
\text{and } \text{targetnode } x = \text{sourcenode } ax \text{ by } (\text{auto elim:path-split-snoc})
\]

\[
\begin{align*}
\{ \text{ assume } \text{targetnode } x = \text{Exit} \\
\text{ with } (\text{valid-edge } ax) \cdot (\text{targetnode } x = \text{sourcenode } ax) \\
\text{ have } \text{False by } (\text{rule Exit-source}, \text{simp+}) \}
\end{align*}
\]

\[
\text{hence } \text{targetnode } x \neq \text{Exit by } \text{clarsimp}
\]

\[
\begin{align*}
\text{with } (\text{valid-edge } x) \cdot (\text{targetnode } x = \text{sourcenode } ax) & \text{[THEN sgm]} \\
\text{have } \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \\
& (\text{Node } (\text{sourcenode } x), \text{kind } x, \text{Node } (\text{sourcenode } ax)) \\
\text{by } (\text{fastforce intro:lift-valid-edge,lve-edge}) \\
\text{hence } \text{path:CFG:path src trg} \\
& (\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \\
& (\text{Node Entry}) \cdot (\text{es @ \[(\text{Node } (\text{sourcenode } x), \text{kind } x, \text{Node } (\text{sourcenode } ax))\]}) \\
& (\text{Node } (\text{sourcenode } ax)) \\
\text{by } (\text{fastforce intro:CFG:Cons-path}[\text{OF CFG}])
\end{align*}
\]

\[
\begin{align*}
\text{thus } ?\text{case by blast} \\
\text{then obtain } \text{es where } \text{CFG:path src trg} \\
& (\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \\
& (\text{Node Entry}) \cdot (\text{es @ \[(\text{Node } (\text{sourcenode } x), \text{kind } x, \text{Node } (\text{sourcenode } ax))\]}) \\
& (\text{Node } (\text{sourcenode } ax)) \\
\text{by } (\text{rule CFG:path-Append}[\text{OF CFG}]) \\
\text{with } \text{Node Exit show } ?\text{thesis by fastforce}
\end{align*}
\]

\[
\text{next}
\]

\[
\begin{align*}
\text{case inner} \\
\text{from } (\text{valid-node } m) & \text{ obtain as } \text{where } \text{Entry } \rightarrow as \rightarrow* m \\
\text{by } (\text{fastforce dest:Entry-path}) \\
\text{with inner } \exists \text{es. CFG:path src trg} \\
& (\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \\
& (\text{Node Entry}) \cdot (\text{es @ \[(\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node Exit})\]}) \\
\text{proof(induct arbitrary:m rule:rev-induct)}
\end{align*}
\]
case Nilrom (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)
ote IH = (∃m. [inner-node m; Entry −xs →∗ m])
   ⇒ ∃es. CFG.path src trg
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   (Node Entry) (Node m)
from (Entry −xs@[x] →∗ m) have Entry −xs →∗ sourcenode x
   and valid-edge x and m = targetnode x by (auto elim: path-split-snoc)
with (inner-node m)
have edge: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
   (Node (sourcenode x), kind x, Node m)
   by (fastforce intro; lve-edge simp: inner-node-def)
   hence path: CFG.path src trg
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node (sourcenode x)) [] (Node (sourcenode x), kind x, Node m) [] (Node m)
by (fastforce intro; CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG] simp: CFG.valid-node-def [OF CFG])
from (valid-edge x) have valid-node (sourcenode x) by simp
thus ?case
proof (cases sourcenode x rule: valid-node-cases)
case Entry
with edge have CFG.path src trg
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   (Node Entry) [] (Node Entry, kind x, Node m)
   apply − apply (rule CFG.Cons-path[OF CFG])
   apply (rule CFG.empty-path[OF CFG])
   by (auto simp: CFG.valid-node-def [OF CFG])
thus ?thesis by blast
next
case Exit
with (valid-edge x) have False by (rule Exit-source)
thus ?thesis by simp
next
case inner
from IH[OF this] (Entry −xs →∗ sourcenode x) obtain es
where CFG.path src trg
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
   (Node Entry) es (Node (sourcenode x)) by blast
with path have CFG.path src trg
   (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node Entry) (es@[((Node (sourcenode x),kind x,Node m))] (Node m)
by (rule CFG.path-Append[OF CFG])
thus ?thesis by blast
qed
qed
then obtain es where path:CFG.path src trg
(Node Entry) es (Node m) by blast
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,(λs. True),Node Entry) by (fastforce intro: lve-Entry-edge)
from this path Node have CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry ((NewEntry,(λs. True),Node Entry)≠es) n
by (fastforce intro: CFG.Cons-path[OF CFG])
thus ?thesis by blast
qed
qed
next
fix n assume CFG.valid-node src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
show ∃as. CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n as NewExit
proof (cases n)
case NewEntry
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,(λs. False),NewExit) by (fastforce intro: lve-Entry-Exit-edge)
with NewEntry have CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
((NewEntry,(λs. False),NewExit)) NewExit
by (fastforce intro: CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
simp: CFG.valid-node-def[OF CFG])
thus ?thesis by blast
next
case NewExit
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,(λs. False),NewExit) by (fastforce intro: lve-Entry-Exit-edge)
with NewExit have CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
n [] NewExit
by (fastforce intro: CFG.empty-path[OF CFG] simp: CFG.valid-node-def[OF CFG])
thus ?thesis by blast
next
case (Node m)
with Entry-Exit-edge (CFG.valid-node src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n
have valid-node m
by (auto elim: lve-valid-edge.cases)
\[
\text{simp: CFG.valid-node-def [OF CFG] valid-node-def}
\]

thus \(\text{thesis}\)

proof(cases m rule: valid-node-cases)

\begin{itemize}
  \item case Entry
  \begin{itemize}
    \item from inner obtain \(ax\) where valid-edge \(ax\) and inner-node (targetnode \(ax\)) and
      sourcenode \(x\) = Entry by (rule inner-node-Entry-edge)
    \item hence edge: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
      (Node Entry, kind \(ax\), Node (targetnode \(ax\))
      by (auto intro: lift-valid-edge.lve-edge simp: inner-node-def)
      have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
        (Node Exit, (\(\lambda s. \text{True}\)), NewExit) by (fastforce intro: lve-Exit-edge)
    \item hence path: CFG.path src try
      (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
      (Node Exit) [(Node Exit, (\(\lambda s. \text{True}\)), NewExit)] (NewExit)
      by (fastforce intro: CFG.Cons-path [OF CFG] CFG.empty-path [OF CFG]
        simp: CFG.valid-node-def [OF CFG])
      from (inner-node (targetnode \(ax\))) have valid-node (targetnode \(ax\)) by (rule inner-is-valid)
  \end{itemize}
\end{itemize}

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

from this valid-edge \(ax\) have \(\exists es.\) CFG.path src try

\begin{itemize}
  \item (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node (targetnode \(ax\))) es (Node Exit)
\end{itemize}

proof (induct \(ax\) arbitrary: \(ax\))

\begin{itemize}
  \item case Nil
    \begin{itemize}
      \item from (targetnode \(ax\) \(-\) \(\rightarrow\) Exit) have targetnode \(ax\) = Exit by fastforce
      \item hence CFG.path src try
        (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
        (Node (targetnode \(ax\))) \(\rightarrow\) (Node Exit)
        apply simp apply (rule CFG.empty-path [OF CFG])
        by (auto intro: lve-Exit-edge simp: CFG.valid-node-def [OF CFG])
    \end{itemize}
\end{itemize}

thus ?case by blast

next

\begin{itemize}
  \item case (Cons \(x\) \(xs\))
    \begin{itemize}
      \item note \(\exists es.\) CFG.path src try
        (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
        (Node (targetnode \(ax\))) es (Node Exit)
      \item from (targetnode \(ax\) \(-\) \(x\#\) \(\rightarrow\) Exit)
        have targetnode \(x\) \(-\text{as} \rightarrow\) Exit and valid-edge \(x\)
        and sourcenode \(x\) = targetnode \(ax\) by (auto elim: path-split-Cons)
        \begin{itemize}
          \item assume sourcenode \(x\) = Entry
            \begin{itemize}
              \item with (valid-edge \(ax\)) (sourcenode \(x\) = targetnode \(ax\))
              \item have False by (rule Entry-target, simp+)
            \end{itemize}
            hence sourcenode \(x\) \(\neq\) Entry by clarsimp
            \begin{itemize}
              \item with (valid-edge \(x\)) (sourcenode \(x\) = targetnode \(ax\)) [THEN sym]
              \item have edge: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
                (Node (targetnode \(ax\)), kind \(x\), Node (targetnode \(x\))
                by (fastforce intro: lift-valid-edge.lve-edge)
              \end{itemize}
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

from \(IH\) [OF (targetnode \(x\) \(-\text{as} \rightarrow\) Exit); (valid-edge \(x\))] obtain \(es\)
where $CFG.\text{path\ src\ trg}$

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

(Node (targetnode $x$)) es (Node Exit) by blast

with edge have $CFG.\text{path\ src\ trg}$

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

(Node (targetnode $ax$))

by (fastforce intro:$CFG.\text{Cons-path}[OF CFG])

thus $?'case$ by blast

qed

then obtain es where $CFG.\text{path\ src\ trg}$

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

(Node (targetnode $ax$)) es (Node Exit) by blast

with edge have $CFG.\text{path\ src\ trg}$

(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:$CFG.\text{Cons-path}[OF CFG]$)

with path have $CFG.\text{path\ src\ trg}$

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

(Node Entry) (((Node Entry, kind $ax$, Node (targetnode $ax$))#es)@

[(Node Exit, (λs. True)\/, NewExit)]) NewExit

by $-(\text{rule$ CFG.\text{path-Append}[OF CFG]$})$

with Node Entry show $?'thesis$ by fastforce

next

case Exit

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

(Node Exit, (λs. True)\/, NewExit) by (fastforce intro:lve-Exit-edge)

with Exit Node have $CFG.\text{path\ src\ trg}$

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

n [(Node Exit, (λs. True)\/, NewExit)] NewExit

by (fastforce intro:$CFG.\text{Cons-path}[OF CFG]$ $CFG.\text{empty-path}[OF CFG]$ $simp:CFG.\text{valid-node-def}[OF CFG]$)

thus $?'thesis$ by blast

next

case inner

from $\langle\text{valid-node } m\rangle$ obtain as where $m \rightarrow^* \text{ Exit}$

by (fastforce dest:Exit-path)

with inner have $\exists\ es.\ CFG.\text{path\ src\ trg}$

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

(Node $m$) es (Node Exit)

proof (induct as arbitrary:$m$)

case Nil

from $m \rightarrow^* \text{ Exit}$

have $m = \text{ Exit}$ by fastforce

with lve-Exit-edge have $CFG.\text{path\ src\ trg}$

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

(Node $m$) [] (Node Exit)

by (fastforce intro:$CFG.\text{empty-path}[OF CFG]$ $simp:CFG.\text{valid-node-def}[OF CFG]$)
thus \?case by blast

next

case (Cons x xs)

note IH = \(\forall m. [inner-node m; m \rightarrow xs \rightarrow Exit]\)

\(\Rightarrow \exists es. CFG.path src try\) \(\langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit\rangle\)

\((Node m) es (Node Exit)\)

from \(m \rightarrow xs \rightarrow Exit\) have \(targetnode x \rightarrow xs \rightarrow Exit\)
and valid-edge \(x\) and \(m = sourcenode x\) by (auto elim:path-split-Cons)

with \(\langle inner-node m\rangle\)

have edge:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
\((Node m, kind x, Node (targetnode x))\)
by (fastforce intro:lve-edge simp:inner-node-def)

from \(valid-edge x\) have valid-node \((targetnode x)\) by simp
thus \?case

proof(cases targetnode x rule:valid-node-cases)

case Entry

with \(\langle valid-edge x\rangle\) have False by (rule Entry-target)
thus \?thesis by simp

next

case Exit

with \(\langle valid-edge x\rangle\) have CFG.path src try
\((\langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit\rangle\)
\((Node m) ((Node m, kind x, Node (targetnode x)) \# es) (Node Exit)\)

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 \(\langle targetnode x \rightarrow xs \rightarrow Exit\rangle\)] obtain es
where CFG.path src try
\((\langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit\rangle\)
\((Node (targetnode x)) es (Node Exit)\) by blast

with \(\langle valid-edge x\rangle\) have CFG.path src try
\((\langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit\rangle\)
\((Node m) ((Node m, kind x, Node (targetnode x)) \# es) (Node Exit)\)
by (fastforce intro:CFG.Cons-path[OF CFG])
thus \?thesis by blast

qed

then obtain es where path:CFG.path src try
\((\langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit\rangle\)
\((Node m) es (Node Exit)\) by blast

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

hence CFG.path src try
\((\langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit\rangle)\)
(Node Exit) [(Node Exit, \lambda. True) \rightarrow 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 sourcenode targetnode kind Entry Exit)
  n (es@[\{(Node Exit, \lambda. True) \rightarrow NewExit\}]) NewExit
by (fastforce intro: CFG.path-Append[OF CFG])

thus ?thesis by blast

qed

lemma lift-PDG-scd:
assumes PDG::PDG sourcenode targetnode kind valid-edge Entry Def Use state-val
Exit
(Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)
and pd::Postdomination sourcenode targetnode kind valid-edge Entry Exit
and inner::CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit
shows PDG src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
(Postdomination.standard-control-dependence src 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
  Postdomination.standard-control-dependence sourcenode targetnode
  valid-edge Exit
  by (rule PDG)

  have wf: CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
  state-val Exit by (unfold-locales)
  from wf pd inner have pd': Postdomination src try knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry NewExit
  by (rule lift-Postdomination)
  from wf have CFG: CFG src try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  by (rule lift-CFG)
  from wf have CFG-wf:: CFG-wf src try knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val
  by (rule lift-CFG-wf)
  from wf have CFGExit:: CFGExit src try knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry NewExit
  by (rule lift-CFGE exit)
  from wf have CFGExit-wf:: CFGExit-wf src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
by(rule lift-CFGExit-wf)

show ⊢thesis

proof

fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and 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 = {} ∧
lift-Use Use Entry Exit H L NewEntry = {}
by(rule CFG-wf.Entry-empty)

next

fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V ∉ lift-Def Def Entry Exit H L (src a) and pred (knd a) s
with CFG-wf show state-val (transfer (knd a) s) V = state-val s V
by(rule CFG-wfCFG-edge-no-Def-equal)

next

fix a s s'
assume assms: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
∀ V ∈ lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
pred (knd a) s pred (knd a) s'
with CFG-wf show ∀ V ∈ lift-Def Def Entry Exit H L (src a).
state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
by(rule CFG-wfCFG-edge-transfer-uses-only-Use)

next

fix a s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and pred (knd a) s
and ∀ V ∈ lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
with CFG-wf show pred (knd a) s' by(rule CFG-wfCFG-edge-Uses-pred-equal)

next

fix a a'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and try a ≠ try a'
with CFG-wf show ∃ Q Q'. knd a = (Q) ∨ ∧ knd a' = (Q') ∨ ∧
(∀ s. (Q s → ¬ Q' s) ∧ (Q' s → ¬ Q s))
by(rule CFG-wf.deterministic)

next
fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and src a = NewExit
with CFGExit show False by (rule CFGExit.Exit-source)
next
  from CFGExit
  show ∃ a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
    src a = NewEntry ∧ try a = NewExit ∧ knd a = (λ 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 ∃ as. CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    n as n′ ∧ as ≠ []
  by (fastforce simp: Postdomination.standard-control-dependence-def[OF pd′])
  qed
qed

lemma lift-PDG-standard-backward-slice:
fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
and Def and Use and H and L
defines lve: lve ≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and lDef:lDef ≡ lift-Def Def Entry Exit H L
and Use:Use ≡ lift-Use Use Entry Exit H L
assumes PDG: PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
(Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)
and \textit{pd}: Postdomination soucenode targetnode kind valid-edge Entry Exit
and \textit{inner}: CFGExit.inner-node soucenode targetnode valid-edge Entry Exit nx
and H \cap L = \{\} \text{ and } H \cup L = \text{UNIV}
shows NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val
(\textit{PDG}.PDG-BS src trg lve lDef lUse
 (Postdomination.standard-control-dependence src trg lve NewExit))
NewExit H L (Node Entry) (Node Exit)

proof
- interpret \textit{PDG} soucenode targetnode kind valid-edge Entry Def Use state-val Exit
  Postdomination.standard-control-dependence soucenode targetnode
  valid-edge Exit
  by (rule \textit{PDG})
  have \textit{wf}: CFGExit-wf soucenode targetnode kind valid-edge Entry Def Use state-val Exit
  by (unfold-locales)
  interpret \textit{wf’}: CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit
  by (fastforce intro: lift-CFGExit-wf \textit{wf} simp: lve lDef lUse)
  from \textit{pd} inner lve lDef lUse have PDG’: PDG src trg knd
    lve NewEntry lDef lUse state-val NewExit
    (Postdomination.standard-control-dependence src trg lve NewExit)
  by (fastforce intro: lift-PDG-scd)
  from \textit{wf} pd inner have pd’: Postdomination src trg knd
  (lift-valid-edge valid-edge soucenode targetnode kind Entry Exit)
  NewEntry NewExit
  by (rule lift-Postdomination)
  from \textit{wf} lve have CFG: CFG src trg lve NewEntry
  by (fastforce intro: lift-CFG)
  from \textit{wf} lve lDef lUse
  have CFG-wf: CFG-wf src trg knd lve NewEntry lDef lUse state-val
  by (fastforce intro: lift-CFG-wf)
  from \textit{wf} lve have CFGExit: CFGExit src trg knd lve NewEntry NewExit
  by (fastforce intro: lift-CFGExit)
  from \textit{wf} lve lDef lUse
  have CFGExit-wf: CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit
  by (fastforce intro: lift-CFGExit-wf)
show \textit{thesis}

proof
- fix n S
  assume n \in PDG.PDG-BS src trg lve lDef lUse
  (Postdomination.standard-control-dependence src trg lve NewExit) S
  with PDG’ show CFG.valid-node src trg lve n
  by (rule PDG.PDG-BS-valid-node)
next
- fix n S assume CFG.valid-node src trg lve n and n \in S
  thus n \in PDG.PDG-BS src trg lve lDef lUse
  (Postdomination.standard-control-dependence src trg lve NewExit) S
  by (fastforce intro: PDG.PDG-path-Nil[OF PDG’] simp: PDG.PDG-BS-def[OF PDG’])
next
fix n S n V
assume n' ∈ PDG.PDG-BS src trg lve lDef lUse
(Postdomination.standard-control-dependence src trg lve NewExit) S
and CFG-wf.data-dependence src trg lve lDef lUse n V n'
thus n ∈ PDG.PDG-BS src trg lve lDef lUse
(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')

next
fix n S
interpret PDGx:PDG src trg knd lve NewEntry lDef lUse state-val NewExit
Postdomination.standard-control-dependence src trg lve NewExit
by (rule PDG')
interpret pdx:Postdomination src trg knd lve NewEntry NewExit
by (fastforce intro:pd' simp:lve)
have scd:StandardControlDependencePDG src trg knd lve NewEntry
lDef lUse state-val NewExit by (unfold-locales)
from StandardControlDependencePDG.obs-singleton[OF scd]
have (∃ m. CFG.obs src trg lve n
(PDG.PDG-BS src trg lve lDef lUse
(Postdomination.standard-control-dependence src trg lve NewExit) S) = {m}) ∨
CFG.obs src trg lve n
(PDG.PDG-BS src trg lve lDef lUse
(Postdomination.standard-control-dependence src trg lve NewExit) S) = {}
by (fastforce simp:StandardControlDependencePDG.PDG-BS-s-def[OF scd])
thus finite (CFG.obs src trg lve n
(PDG.PDG-BS src trg lve lDef lUse
(Postdomination.standard-control-dependence src trg lve NewExit) S))
by fastforce

next
fix n S
interpret PDGx:PDG src trg knd lve NewEntry lDef lUse state-val NewExit
Postdomination.standard-control-dependence src trg lve NewExit
by (rule PDG')
interpret pdx:Postdomination src trg knd lve NewEntry NewExit
by (fastforce intro:pd' simp:lve)
have scd:StandardControlDependencePDG src trg knd lve NewEntry
lDef lUse state-val NewExit by (unfold-locales)
from StandardControlDependencePDG.obs-singleton[OF scd]
have (∃ m. CFG.obs src trg lve n
(PDG.PDG-BS src trg lve lDef lUse
(Postdomination.standard-control-dependence src trg lve NewExit) S) = {m}) ∨
CFG.obs src trg lve n

34
(PDG.PDG-BS src trg lve lDef lUse
  (Postdomination.standard-control-dependence src trg lve NewExit) S) = {}
by (fastforce simp: StandardControlDependence_PDG.PDG-BS-s-def [OF scd])
thus card (CFG.obs src trg lve n
  (PDG.PDG-BS src trg lve lDef lUse
  (Postdomination.standard-control-dependence src trg lve NewExit) S)) ≤ 1
by fastforce
next
  fix a assume lve a and src a = NewEntry
  with lve show trg a = NewExit ∨ trg a = Node Entry
    by (fastforce elim: lift-valid-edge_cases)
next
  from lve-Entry-edge lve
  show ∃ a. lve a ∧ src a = NewEntry ∧ trg a = Node Entry ∧ knd a = (λs. True) √
    by (fastforce simp: knd-def)
next
  fix a assume lve a and trg a = Node Entry
  with lve show src a = NewEntry by (fastforce elim: lift-valid-edge_cases)
next
  fix a assume lve a and trg a = NewExit
  with lve show src a = NewEntry ∨ src a = Node Exit
    by (fastforce elim: lift-valid-edge_cases)
next
  from lve-Exit-edge lve
  show ∃ a. lve a ∧ src a = NewExit ∧ trg a = Node Exit ∧ knd a = (λs. True) √
    by (fastforce simp: knd-def)
next
  fix a assume lve a and src a = Node Exit
  with lve show trg a = NewExit by (fastforce elim: lift-valid-edge_cases)
next
  from lDef show lDef (Node Entry) = H
    by (fastforce elim: lift-Def-set_cases intro: lift-Def-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Entry) = H
    by (fastforce elim: lift-Use-set_cases intro: lift-Use-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Exit) = L
    by (fastforce elim: lift-Use-set_cases intro: lift-Use-Low)
next
  from H ∩ L = {} show H ∩ L = {}
next
  from H ∪ L = UNIV show H ∪ L = UNIV
qed
qed

3.2.4 Lifting PDG-BS with weak-control-dependence

lemma lift-StrongPostdomination:
assumes \( \text{wf}: \text{CFGExit-wf} \) sourcename targetname kind valid-edge Entry Def Use state-val Exit

and \( \text{spd}: \text{StrongPostdomination} \) sourcename targetname kind valid-edge Entry Exit

and \( \text{inner}: \text{CFGExitinner-node} \) sourcename targetname valid-edge Entry Exit nz

shows \( \text{StrongPostdomination} \) sourcename targetname kind valid-edge Exit

\( \text{state-val} \) Exit

and \( \text{spd}: \text{StrongPostdomination} \) sourcename targetname kind valid-edge Entry Exit

and \( \text{inner}: \text{CFGExitinner-node} \) sourcename targetname valid-edge Entry Exit nz

proof —

interpret \( \text{StrongPostdomination} \) sourcename targetname kind valid-edge Entry Exit

by (rule \( \text{spd} \))

have \( \text{pd}: \text{Postdomination} \) sourcename targetname kind valid-edge Entry Exit

by (unfold-locales)

interpret \( \text{pd}: \text{Postdomination} \) sourcename targetname kind valid-edge Entry Exit

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

interpret \( \text{CFGExit-wf}: \text{CFGExit-wf} \) sourcename targetname kind valid-edge Entry Exit

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

from \( \text{wf} \) have \( \text{CFG}: \text{CFG} \) sourcename targetname kind valid-edge Entry Exit

by (rule lift-CFG)

show \( ?\text{thesis} \)

proof

fix \( n \) assume \( \text{CFG}.\text{valid-node} \) sourcename targetname kind valid-edge Entry Exit

\( n \)

show finite

\( \{ n': \exists a'. \text{lift-valid-edge valid-edge source-sourcename target-sourcename kind Entry Exit a'} \land \) src \( a' = n' \land \) trg \( a' = n' \}\)

proof (cases \( n \))

case \( \text{NewEntry} \)

hence \( \{ n': \exists a'. \text{lift-valid-edge valid-edge source-sourcename target-sourcename kind Entry Exit a'} \land \) src \( a' = n' \land \) trg \( a' = n' \} = \{ \text{NewExit}, \text{Node} \} \)

by (auto elim: lift-valid-edge \( \text{cases} \) intro: lift-valid-edge \( \text{intros} \))

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

next
case \( \text{NewExit} \)

hence \( \{ n': \exists a'. \text{lift-valid-edge valid-edge source-sourcename target-sourcename kind Entry Exit a'} \land \) src \( a' = n' \land \) trg \( a' = n' \} = \{ \} \)

by fastforce

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

next
case \( \text{Node} m \)

with \( \text{Entry-Exit-edge} \) \( \langle \text{CFG}.\text{valid-node} \) sourcename targetname kind Entry Exit \( n \rangle \)

have valid-node \( m \)

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

36
hence finite \{ m', \exists a'. valid-edge a' \land sourcenode a' = m \land targetnode a' = m' \}

by (rule successor-set-finite)

have \{ m', \exists a'. lift-valid-edge valid-edge sourcenode targetnode kind
Entry Exit a' \land src a' = Node m \land trg a' = Node m' \} \subseteq
\{ m', \exists a'. valid-edge a' \land sourcenode a' = m \land targetnode a' = m' \}

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

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

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

by -(rule finite-subset)

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

by fastforce

hence finite \{ Node ' \{ m', \exists a'. lift-valid-edge valid-edge sourcenode
targetnode kind Entry Exit a' \land src a' = Node m \land trg a' = Node m' \}\} \cup
\{ NewEntry, NewExit \} by fastforce

with Node have \{ n', \exists a'. lift-valid-edge valid-edge sourcenode targetnode kind
Entry Exit a' \land src a' = n \land trg a' = n' \} \subseteq
\{ Node ' \{ m', \exists a'. lift-valid-edge valid-edge sourcenode
targetnode kind Entry Exit a' \land src a' = Node m \land trg a' = Node m' \}\} \cup
\{ NewEntry, NewExit \} by auto (case-tac x, auto)

with fin show \{ thesis \} by -(rule finite-subset)

qed

qed

lemma lift-PDG-wcd:
assumes PDG: PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
(StrongPostdomination.weak-control-dependence sourcenode targetnode
equal-edge Exit)
and spd: StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit
and inner: CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
shows PDG src trg 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 trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit)

proof -
interpret PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
StrongPostdomination.weak-control-dependence sourcenode targetnode

37
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 knad
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry NewExit
  by (rule lift-StrongPostdomination)

from wf have CFG:CFG src try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  by (rule lift-CFG)

from wf have CFG-wf:CFG-wf src try knad
  (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 try knad
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry NewExit
  by (rule lift-CFGExit)

from wf have CFGExit-wf:CFGExit-wf src try knad
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
  by (rule lift-CFGExit-wf)

show ?thesis

proof

fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and 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 = \{\} ∧
  lift-Use Use Entry Exit H L NewEntry = \{\}

  by (rule CFG-wf.Entry-empty)

next

fix a V s

assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and V ∉ lift-Def Def Entry Exit H L (src a) and pred (knd a) s

  with CFG-wf show state-val (transfer (knd a) s) V = state-val s V

  by (rule CFG-wf.CFG-edge-no-Def-equal)

next

fix a s s’

assume assms: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
∀ V∈lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
pred (knd a) s pred (knd a) s'
with CFG-wf show ∀ V∈lift-Def Def Entry Exit H L (src a), state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
by (rule CFG-wf_CFG-edge-transfer-uses-only-Use)
next
fix a s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and pred (knd a) s
and ∀ V∈lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
with CFG-wf show pred (knd a) s' by (rule CFG-wf_CFG-edge-Uses-pred-equal)
next
fix a a'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and trg a ≠ trg a'
with CFG-wf show ∃ Q Q'. knd a = (Q)√ ∧ knd a' = (Q')√ ∧
(∀ s. (Q s → ¬ Q' s) ∧ (Q' s → ¬ Q s))
by (rule CFG-wf.cfg-deterministic)
next
fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and src a = NewExit
with CFGExit show False by (rule CFGExit.Exit-source)
next
from CFGExit
show ∃ a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
src a = NewEntry ∧ trg a = NewExit ∧ knd a = (λs. False)√
by (rule CFGExit.Entry-Exit-edge)
next
from CFGExit-wf
show lift-Def Def Entry Exit H L NewExit = {} ∧
lift-Use Use Entry Exit H L NewExit = {}
by (rule CFGExit-wf.Exit-empty)
next
fix n n'
assume wcd:StrongPostdomination.weak-control-dependence src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n n'
show n' ≠ NewExit
proof (rule ccontr)
assume ¬ n' ≠ NewExit
hence n' = NewExit by simp
with wcd spd' show False
by (fastforce intro: StrongPostdomination.Exit-not-weak-control-dependent)
qed
next
fix n n'
assume StrongPostdomination.weak-control-dependence src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n
Thus \(\exists as. \ CFG\_path \ src \ trg\)

\((lift\_valid\_edge \ valid\_edge \ sourcenode \ targetnode \ kind \ Entry \ Exit)\)

\(n \ as \ n' \land as \neq []\)

by (fastforce simp: StrongPostdomination.weak-control-dependence-def (OF spd'))

qed

qed

\textbf{Lemma} \(lift\_PDG\_weak\_backward\_slice:\)

fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit and Def and Use and H and L

defines lve: lve \equiv lift\_valid\_edge \ valid\_edge \ sourcenode \ targetnode \ kind \ Entry \ Exit

and lDef: lDef \equiv lift\_Def \ Def \ Entry \ Exit \ H \ L

and lUse: lUse \equiv lift\_Use \ Use \ Entry \ Exit \ H \ L

assumes PDG: PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit

wardomination.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 \(H \cap L = \{\} \land H \cup L = \text{UNIV}\)

shows NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val

(PDG.PDG-BS src trg lve lDef lUse

(StrongPostdomination.weak-control-dependence src trg lve NewExit))

NewExit H L (Node Entry) (Node Exit)

proof –

interpret PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit

StrongPostdomination.weak-control-dependence sourcenode targetnode valid-edge Exit

by (rule PDG)

have wf: CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit by (unfold-locales)

interpret wf': CFGExit-wf src trg knd lve NewEntry lDef lUse state-val NewExit

by (fastforce intro: lift-CFGExit-wf wf simp: lve lDef lUse)

from PDG spd inner lve lDef lUse have PDG': PDG src trg knd lve NewEntry lDef lUse state-val NewExit

(StrongPostdomination.weak-control-dependence src trg lve NewExit)

by (fastforce intro: lift-PDG-wcd)

from wf spd inner have spd': StrongPostdomination src trg knd

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

NewEntry NewExit

by (rule lift-StrongPostdomination)

from wf lve have CFG: CFG src trg lve NewEntry

by (fastforce intro: lift-CFG)

from wf lve lDef lUse
have \( \text{CFG-wf}: \text{CFG-wf} \) src trg knd lve NewEntry lDef lUse state-val
by (fastforce intro:lift-CFG-wf)

from \( \text{wflve} \) have \( \text{CFGExit}: \text{CFGExit} \) src trg knd lve NewEntry NewExit
by (fastforce intro:lift-CFGExit)

from \( \text{wflve} \) lDef lUse
have \( \text{CFGExit-wf}: \text{CFGExit-wf} \) src trg knd lve NewEntry lDef lUse state-val
NewExit
by (fastforce intro:lift-CFGExit-wf)

show \?thesis
proof
fix \( n \) S
assume \( n \in \text{PDG-PDG-BS} \) src trg lve lDef lUse
(StrongPostdomination.weak-control-dependence src trg lve NewExit) S
with \( \text{PDG'} \)
show \( \text{CFG.valid-node} \) src trg lve n
by (rule PDG.PDG-BS-valid-node)

next
fix \( n \) S assume \( \text{CFG.valid-node} \) src trg lve n and \( n \in S \)
thus \( n \in \text{PDG-PDG-BS} \) 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 \( n' \in \text{PDG-PDG-BS} \) src trg lve lDef lUse
(StrongPostdomination.weak-control-dependence src trg lve NewExit) S
and \( \text{CFG-wf.data-dependence} \) src trg lve lDef lUse n V n'
thus \( n \in \text{PDG-PDG-BS} \) 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')

split:if-split-asm)

next
fix \( n \) S
interpret \( \text{PDGx}: \text{PDG} \) src trg knd lve NewEntry lDef lUse state-val NewExit
StrongPostdomination.weak-control-dependence src trg lve NewExit
by (rule PDG')
interpret \( \text{spdx}: \text{StrongPostdomination} \) src trg knd lve NewEntry NewExit
by (fastforce intro:spdx simp:lve)

have \( \text{wcd}: \text{WeakControlDependencePDG} \) src trg knd lve NewEntry
lDef lUse state-val NewExit by (unfold-locales)

from \text{WeakControlDependencePDG.obs-singleton[OF wcd]}
have \( \exists m. \text{CFG.obs src trg lve n} \)
(\( \text{PDG-PDG-BS} \) src trg lve lDef lUse
(StrongPostdomination.weak-control-dependence src trg lve NewExit) S) = \{m\} \) v
\( \text{CFG.obs src trg lve n} \)
(\( \text{PDG-PDG-BS} \) src trg lve lDef lUse

41
(StrongPostdomination.weak-control-dependence src trg lve NewExit) S) = 

\{\} 

by (fastforce simp: WeakControlDependence PDG.PDG-BS-w-def [OF wcd])

thus finite (CFG.obs src trg lve n)

(PDG.PDG-BS src trg lve lDef lUse

(StrongPostdomination.weak-control-dependence src trg lve NewExit) S))

by fastforce

next

fix n S

interpret PDGx:PDG src trg knd lve NewEntry lDef lUse state-val NewExit
  StrongPostdomination.weak-control-dependence src trg lve NewExit

by (rule PDG')

interpret spdx:StrongPostdomination src trg knd lve NewEntry NewExit

by (fastforce intro: spd' simp: lve)

have wcd: WeakControlDependence PDG src trg knd lve NewEntry
  lDef lUse state-val NewExit by (unfold-locales)

from WeakControlDependence PDG.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\} \lor 

CFG.obs src trg lve n

(PDG.PDG-BS src trg lve lDef lUse

(StrongPostdomination.weak-control-dependence src trg lve NewExit) S) = 

\{\}

by (fastforce simp: WeakControlDependence PDG.PDG-BS-w-def [OF wcd])

thus card (CFG.obs src trg lve n)

(PDG.PDG-BS src trg lve lDef lUse

(StrongPostdomination.weak-control-dependence src trg lve NewExit) S)) \leq 

I

by fastforce

next

fix a assume lve a and src a = NewEntry

with lve show trg a = NewExit \lor trg 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 trg a = Node Entry \land knd a = (\lambda s. True)

by (fastforce simp: knd-def)

next

fix a assume lve a and trg a = Node Entry

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

next

fix a assume lve a and trg a = NewExit

with lve show src a = NewEntry \lor src a = Node Exit

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

next

from lve-Exit-edge lve

42
\[
\exists a. \text{lve} a \land \text{src} a = \text{Node Exit} \land \text{trg} a = \text{NewExit} \land \text{knd} a = (\lambda s. \text{True})
\]

by (fastforce simp:knd-def)

next

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

next

from \ lDef \ show \ lDef (\text{Node Entry}) = \text{H}
by (fastforce elim:lift-Def-set.cases intro:lift-Def-High)

next

from \ Entry\-noteq\-Exit \ lUse \ show \ lUse (\text{Node Entry}) = \text{H}
by (fastforce elim:lift-Use-set.cases intro:lift-Use-High)

next

from \ Entry\-noteq\-Exit \ lUse \ show \ lUse (\text{Node Exit}) = \text{L}
by (fastforce elim:lift-Use-set.cases intro:lift-Use-Low)

next

from : \text{H} \cap \text{L} = \{\}; \ show \ \text{H} \cap \text{L} = \{\} .
next

from : \text{H} \cup \text{L} = \text{UNIV}; \ show \ \text{H} \cup \text{L} = \text{UNIV} .

qed

qed

end

4 Information Flow for While

theory NonInterferenceWhile imports Slicing.SemanticsWellFormed Slicing.StaticControlDependences LiftingIntra
begin

locale SecurityTypes =
fixes \text{H} :: \text{vname set}
fixes \text{L} :: \text{vname set}
assumes HighLowDistinct: \text{H} \cap \text{L} = \{\}
and HighLowUNIV: \text{H} \cup \text{L} = \text{UNIV}
begin

4.1 Lifting labels-nodes and Defining final

fun labels-LDCFG-nodes :: \text{cmd} \Rightarrow \text{w-node LDCFG-node} \Rightarrow \text{cmd} \Rightarrow \text{bool}
where labels-LDCFG-nodes prog (\text{Node} n) c = labels-nodes prog n c
| labels-LDCFG-nodes prog n c = \text{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
(Node (sourcenode a),kind a,Node (targetnode a))
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)
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 n,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)
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 (targetnode a),kind a,Node (sourcenode a))
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)
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 n,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

lemma lift-CFG-fund-prop:
assumes \[ labels-LDCFG-nodes \ prog \ n \ c \ and \ \langle c, s \rangle \rightarrow^* \langle c', s' \rangle \]
shows \[ \exists n' as. CFG.path src trg \]
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-)) \[ n \ as \ n' \ and \ transfers (CFG.kinds knd as) \ s = s' \and \]
preds (CFG.kinds knd as) \ s \and\ labels-LDCFG-nodes \ prog \ n' \ c'
proof –
from \( \langle \text{labels-LDCFG-nodes} \ \text{prog} \ n \ c \rangle \) obtain \( nx \) where \( n = \text{Node} \ nx \)
and \( \text{labels-nodes} \ \text{prog} \ nx \ c \) by (cases \( n \)) auto
from \( \langle \text{labels-nodes} \ \text{prog} \ nx \ c \rangle \) obtain \( n' \) as where \( \text{prog} \vdash nx \rightarrow^* n' \) and \( \text{transfers} \ (CFG.kinds \text{knd as}) \ s = s' \)
by (auto dest: While-semantics-CFG-wf fundamental-property)
from \( \langle \text{labels-nodes} \ \text{prog} \ n' \ c' \rangle \) have \( \text{labels-LDCFG-nodes} \ \text{prog} \ (\text{Node} \ n') \ c' \)
by simp
from \( \langle \text{prog} \vdash nx \rightarrow^* n' \rangle \) obtain \( s \) where \( \text{transfers} \ (CFG.kinds \text{knd as}) \ s = s' \)
\( \langle \text{preds} \ (CFG.kinds \text{knd as}) \ s \rangle \) \( n = \text{Node} \ nx \)
\( \langle \text{labels-nodes} \ \text{prog} \ nx \ c \rangle \) \( \langle \text{labels-nodes} \ \text{prog} \ n' \ c' \rangle \)
have \( \exists \ es. CFG.path src trg \)
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-)) \( \langle \text{Node} \ nx \rangle \)
\( \text{es} \ (\text{Node} \ n') \ and \ \text{transfers} \ (CFG.kinds \text{knd es}) \ s = s' \and \)
preds (CFG.kinds knd es) \ s
proof(induct arbitrary: \( n \ s \ c \) rule: WCFG-path-induct)
case (empty-path \( n \ nx \))
from \( \text{CFG.valid-node} \ \text{source} \ \text{node} \ \text{target} \ \text{node} \ (\text{valid-edge} \ \text{prog}) \ n \)
have \( \text{valid-node} : CFG.valid-node \ \text{src} \ \text{trg} \)
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-)) \( \langle \text{Node} \ n \rangle \)
by (rule lift-valid-node)
have \( CFG.kinds knd \)
(\[ \text{[\text{[w-node} LDCFG-node \times \text{state} \ \text{edge} \ \text{kind} \times \text{w-node} LDCFG-node] list} = \]
\)
by (simp add: CFG.kinds-def [OF lift-CFG] [OF While-CFGExit-wf-aux])
with \( \text{transfers} \ (CFG.kinds \text{knd [\]}) \ s = s' \) \( \langle \text{preds} \ (CFG.kinds \text{knd [\]}) \ s \rangle \)
valid-node
show \( ?\)case
by (fastforce intro: CFG.empty-path [OF lift-CFG] [OF While-CFGExit-wf-aux])
simp: While-CFG.kinds-def
next
case (Cons-path \( n'' \) as \( n' \) a \( nx \))
note IH = \{ \forall n \ s c. \ [\text{transfers (CFG.kinds kind as)} s = s'] \land \text{preds (CFG.kinds kind as)} s; \ n = \text{LDCFG-node.Node n''}; \ text{labels-nodes prog n'' c}; \ text{labels-nodes prog n' c} \}
\implies \exists s. \text{CFG.path src trg}
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind (\text{-Entry-}) (\text{-Exit-}))
\langle \text{LDCFG-node.Node n''} \rangle \ es \text{ (LDCFG-node.Node n'')} \land
\text{transfers (CFG.kinds kind es)} s = s' \land \text{preds (CFG.kinds kind es)} s;
from \text{transfers (CFG.kinds kind (a \# as)) s = s'}
\text{have transfers (CFG.kinds kind as)} (\text{transfer (kind a) s}) = s'
by (\text{simp add:While-CFG.kinds-def})
from \text{preds (CFG.kinds kind (a \# as)) s} \text{have preds (CFG.kinds kind as)} (\text{transfer (kind a) s})
and \text{pred (kind a) s} by (\text{simp-all add:While-CFG.kinds-def})
\begin{proof}
\text{cases source node a = (\text{-Entry-})}
\begin{case}
\text{True}
\begin{with}
\text{(source node a = nx) (labels-nodes prog nx c) have False by simp}
\end{with}
\end{case}
thus \text{thesis by simp}
\end{proof}
next
\begin{case}
\text{False}
\begin{with}
\text{(valid-edge prog a)}
\text{have edge:lift-valid-edge (valid-edge prog) sourcenode targetnode kind}
\begin{Entry Exit (Node (source node a),kind a,Node (targetnode a))
\end{Entry}
by (fastforce intro:live-edge)
\end{with}
from \text{(prog \# as->* n')}
\text{have CFG.valid-node source node targetnode (valid-edge prog) n''}
by (rule While-CFG.path-valid-node)
\begin{then}
\text{obtain c'' where labels-nodes prog n'' c''}
\end{then}
\begin{proof}
\text{cases rule:While-CFGExit.valid-node-cases}
\begin{case}
\text{Entry}
\begin{with}
\text{(targetnode a = n'\#) (valid-edge prog a) have False by fastforce}
\end{with}
\end{case}
thus \text{thesis by simp}
\end{proof}
next
\begin{case}
\text{Exit}
\begin{with}
\text{(prog \# as->* n') have n' = (\text{-Exit-}) by fastforce}
\end{with}
\text{with (labels-nodes prog n' c') have False by fastforce}
\end{case}
thus \text{thesis by simp}
next
\begin{case}
\text{inner}
\begin{then}
\text{where \text{[simp]:n'' = (- l'' \#) by (cases n'')}} auto
\end{then}
\begin{with}
\text{(valid-edge prog a)} \text{(targetnode a = n'')} \text{have l'' < \#:prog}
\end{with}
by (fastforce intro:WCFG-targetlabel-less-num-nodes simp:valid-edge-def)
\begin{then}
\text{obtain c'' where labels prog l'' c''}
\end{then}
by (fastforce dest:less-num-inner-nodes-label)
\text{with that show \text{thesis by fastforce}}
\end{case}
\text{qed}
\end{proof}
from \text{IH[OF \langle \text{transfers (CFG.kinds kind as)} (\text{transfer (kind a) s}) = s'\rangle \land \text{preds (CFG.kinds kind as)} (\text{transfer (kind a) s}) \rangle - this}
\text{\langle labels-nodes prog n' c' \rangle]}
\end{proof}
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 n''') ((Node (sourcenode a),kind a,Node (targetnode a))#es)
and transfers (CFG.kinds kind es) (transfer (kind a) s) = s'
and 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 \(\text{case by fastforce intro: WCFG-LAssSkip}\)

next

case (Seq \(c_1 \ c_2\))

note IH2 = (\(\forall n. \text{labels-nodes} \ c_2 \ n \ \text{Skip} \implies c_2 \vdash n - \uparrow \text{id} \to (\text{-Exit-})\))

from \(\langle \text{labels-nodes} \ (c_1; c_2) \ n \ \text{Skip} \rangle\) obtain \(l\) where \(n = (\cdot \ l \cdot)\)

and \(l \geq \#c_1\) and \(\text{labels-nodes} \ c_2 \ (\cdot \ l - \#c_1 \cdot) \ \text{Skip}\)

by (cases \(n\))(auto elim:labels.cases)

from IH2[OF (\(\text{labels-nodes} \ c_2 \ (\cdot \ l - \#c_1 \cdot) \ \text{Skip}\))] have \(c_2 \vdash (\cdot \ l - \#c_1 \cdot) - \uparrow \text{id} \to (\text{-Exit-})\).

with \((l \geq \#c_1)\) have \(c_1; c_2 \vdash (\cdot \ l - \#c_1 \cdot) \oplus \#c_1 - \uparrow \text{id} \to (\text{-Exit-}) \oplus \#c_1\)

by (fastforce intro: WCFG-SkipSecond)

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

next

case (Cond \(b \ c_1 \ c_2\))

note IH1 = (\(\forall n. \text{labels-nodes} \ c_1 \ n \ \text{Skip} \implies c_1 \vdash n - \uparrow \text{id} \to (\text{-Exit-})\))

note IH2 = (\(\forall n. \text{labels-nodes} \ c_1 \ n \ \text{Skip} \implies c_1 \vdash n - \uparrow \text{id} \to (\text{-Exit-})\))

from \(\langle \text{labels-nodes} \ (b \ c_1 \ c_2) \ n \ \text{Skip} \rangle\) obtain \(l\) where \(n = (\cdot \ l \cdot \cdot)\) and \(\text{disj:}(l \geq 1 \land \text{labels-nodes} \ c_1 \ (\cdot \ l - 1 \cdot) \ \text{Skip}) \lor\)

\((l \geq \#c_1 + 1 \land \text{labels-nodes} \ c_2 \ (\cdot \ l - \#c_1 - 1 \cdot) \ \text{Skip})\)

by (cases \(n\))(fastforce elim:labels.cases+)

from disj show \(\text{case by simp add:id-def}\)

proof

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

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

from IH1[OF (\(\text{labels-nodes} \ c_1 \ (\cdot \ l - 1 \cdot) \ \text{Skip}\))] have \(c_1 \vdash (\cdot \ l - 1 \cdot) - \uparrow \text{id} \to (\text{-Exit-})\).

with \((1 \leq l)\) have if \((b) \ c_1 \ \text{else} \ c_2 \vdash (\cdot \ l - 1 \cdot) \oplus 1 - \uparrow \text{id} \to (\text{-Exit-}) \oplus 1\)

by (fastforce intro: WCFG-CondThen)

with \(n = (\cdot \ l \cdot)\) \((1 \leq l)\) show \(\text{case by simp add:id-def}\)

next

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

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

from IH2[OF (\(\text{labels-nodes} \ c_2 \ (\cdot \ l - \#c_1 - 1 \cdot) \ \text{Skip}\))] have \(c_2 \vdash (\cdot \ l - \#c_1 - 1 \cdot) - \uparrow \text{id} \to (\text{-Exit-})\).

with \(\#c_1 + 1 \leq l\) have if \((b) \ c_1 \ \text{else} \ c_2 \vdash (\cdot \ l - \#c_1 - 1 \cdot) \oplus (\#c_1 + 1)\)

\(- \uparrow \text{id} \to (\text{-Exit-}) \oplus (\#c_1 + 1)\)

by (fastforce intro: WCFG-CondElse)

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

qed

next

case (While \(b \ c\))

from \(\langle \text{labels-nodes} \ (\text{while} \ (b) \ c) \ n \ \text{Skip} \rangle\) have \(n = (\cdot \ l \cdot)\)

by (cases \(n\))(auto elim:labels.cases)

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

qed

4.2 Semantic Non-Interference for Weak Order Dependence

lemmas WODNonInterferenceGraph =
\texttt{lift-wod-backward-slice}[OF While-CFGExit-wf-aux HighLowDistinct HighLowU-NIV]

**Lemma** \texttt{WODNonInterference}:

NonInterferenceIntra src trg knd

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind

(-Entry-) (-Exit-))

NewEntry (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L) id

(CFG-wf, wod-backward-slice src trg

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind

(-Entry-) (-Exit-))

(lift-Def (Defs prog) (-Entry-) (-Exit-) H L)

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L))

reds labels-LDCFG-nodes prog

NewExit H L (LDCFG-node. Node (-Entry-)) (LDCFG-node. Node (-Exit-))

**Proof**

\texttt{interpret} NonInterferenceIntraGraph src trg knd

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind

(-Entry-) (-Exit-))

NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L) id

(CFG-wf, wod-backward-slice src trg

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind

(-Entry-) (-Exit-))

(lift-Def (Defs prog) (-Entry-) (-Exit-) H L)

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L)

NewExit H L LDCFG-node. Node (-Entry-) LDCFG-node. Node (-Exit-)

by (rule WODNonInterferenceGraph)

\texttt{interpret} BackwardSlice-wf src trg knd

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind

(-Entry-) (-Exit-))

NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L) id

(CFG-wf, wod-backward-slice src trg

(lift-valid-edge (valid-edge prog) sourcenode targetnode kind

(-Entry-) (-Exit-))

(lift-Def (Defs prog) (-Entry-) (-Exit-) H L)

(lift-Use (Uses prog) (-Entry-) (-Exit-) H L) reds labels-LDCFG-nodes prog

**Proof** (unfold-localases)

\texttt{fix} n c s c' s'

\texttt{assume} labels-LDCFG-nodes prog n c and \langle c, s \rangle \rightarrow^* \langle c', s' \rangle

\texttt{thus} \exists 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)

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

4.3 Semantic Non-Interference for Standard Control Dependence

lemma inner-node-exists: ∃ n. CFGExit. inner-node sourcenode targetnode
  (valid-edge prog) (∓ Entry-) (∓ Exit-) n
proof –
  have prog ⊢ (∓ Entry-) − (∙ ls. True) ∨→ (∓ 0-) by (rule WCFG-Entry)
  hence CFG. valid-node sourcenode targetnode (valid-edge prog) (∓ 0-)
    by (auto simp: While-CFG. valid-node-def valid-edge-def)
  thus thesis by (auto simp: While-CFGExit. inner-node-def)
qed

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

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

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))
NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node.Node (-Exit-)
proof (unfold-locales)
fix n c s c′ s′
assume labels-LDCFG-nodes prog n c and ⟨c,s⟩ →* ⟨c′,s′⟩
thus 3 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.fand-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 \(nx\) where \([\text{simp}]: n = \text{Node } nx\)

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

from \(\text{labels-nodes prog nx Skip}\) have \(\text{prog } \vdash nx \downarrow id \rightarrow (\text{-Exit-})\)

by (rule final-edge)

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

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

by (auto simp: \(\text{valid-edge-def}\))

with \(\text{labels-nodes prog nx Skip}\)

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

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

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

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

(auto intro !: \(\text{lve-edge simp: \(\text{knd-def valid-edge-def}\)}\))

qed

qed

4.4 Semantic Non-Interference for Weak Control Dependence

lemmas \(\text{WCDNonInterferenceGraph} = \)

\(\text{lift-PDG-weak-backward-slice}\ OF \ W\text{WeakControlDependence}P\text{DG-wcd}\)

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

lemma \(\text{WCDNonInterference}:\)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

proof

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

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

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

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

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

\(\text{52}\)
NewEntry lif-Def (Defs prog) (-Entry-) (-Exit-) H L
l-if-Use (Uses prog) (-Entry-) (-Exit-) H L id
PDG.PDG-BS src trg
  (l-if-valid-edge (valid-edge prog) sourcenode targetnode kind
  (-Entry-) (-Exit-))
  (l-if-Def (Defs prog) (-Entry-) (-Exit-) H L)
  (l-if-Use (Uses prog) (-Entry-) (-Exit-) H L)
(StrongPostdomination.weak-control-dependence src trg
  (l-if-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 kind
l-if-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-)
NewEntry lif-Def (Defs prog) (-Entry-) (-Exit-) H L
l-if-Use (Uses prog) (-Entry-) (-Exit-) H L id
PDG.PDG-BS src trg
  (l-if-valid-edge (valid-edge prog) sourcenode targetnode kind
  (-Entry-) (-Exit-))
  (l-if-Def (Defs prog) (-Entry-) (-Exit-) H L)
  (l-if-Use (Uses prog) (-Entry-) (-Exit-) H L)
(StrongPostdomination.weak-control-dependence src trg
  (l-if-valid-edge (valid-edge prog) sourcenode targetnode kind
  (-Entry-) (-Exit-) NewExit)
NewExit H L LDCFG-node.Node (-Entry-) LDCFG-node. Node (-Exit-)
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
  (l-if-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
  n as n' \land transfers (CFG.kinds kind as) s = s' \land
  preds (CFG.kinds kind 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]: \langle c, \rangle = \text{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 \rightarrow^* id \rightarrow (-Exit-)
  by (rule final-edge)
then obtain a where valid-edge prog a and sourcenode a = nx
  and kind a = \langle id \rangle and targetnode a = (-Exit-)
  by (auto simp: valid-edge-def)
with labels-nodes prog nx Skip
show \exists a. l-if-valid-edge (valid-edge prog) sourcenode targetnode
  kind (-Entry-) (-Exit-) a \land

53
\text{src} \ a = n \land \text{trg} \ a = \text{LDCFG-node} \ (-\text{Exit}) \land \text{knd} \ a = \uparrow \text{id} \\
\text{by} (\text{rule-tac} \ z = (\text{Node} \ nx. \uparrow \text{id}, \text{Node} \ (-\text{Exit}))) \ \text{in} \ \text{exI} \\
\text{(auto intro!; le-edge simp:knd-def valid-edge-def)} \\
\text{qed} \\
\text{qed} \\
\text{end} \\
\text{end} \\

\textbf{References} \\


