Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

August 16, 2018

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

2 Slicing guarantees IFC Noninterference

theory NonInterferenceIntra imports
Slicing.Slice
Slicing.CFGExit-af
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 cannot be altered during a program run. Thus, we have to extend the prerequisites of the slicing framework in [8] accordingly in a new locale:

```
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-) ∧  
      kind a = (λs. True) ∨  
  and Entry-predecessor-of-High:  
    [valid-edge a; targetnode a = (-High-)] ⇒ sourcenode a = (-Entry-)  
  and Exit-edge-Entry-or-Low:  
    [valid-edge a; targetnode a = (-Exit-)]  
    ⇒ sourcenode a = (-Entry-) ∨ sourcenode a = (-Low-)  
  and Low-source-Exit-edge:  
    ∃ a. valid-edge a ∧ sourcenode a = (-Low-) ∧ targetnode a = (-Exit-) ∧  
      kind a = (λs. True) ∨  
  and Exit-successor-of-Low:  
    [valid-edge a; sourcenode a = (-Low-)] ⇒ targetnode a = (-Exit-)  
  and DefHigh: Def (-High-) = H  
  and UseHigh: Use (-High-) = H
```
and UseLow: Use (-Low-) = L
and HighLowDistinct: H \cap L = \{\}
and HighLowUNIV: H \cup L = UNIV

begin

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

lemma Entry-path-High-path:
assumes (-Entry-) as \rightarrow^* n and inner-node n
obtains a' as' where as = a'#as' and (-High-) as' \rightarrow^* n
and kind a' = (\lambda s. True)√
proof(atomize-elim)
  from (-Entry-) as \rightarrow^* n (inner-node n)
  show \exists a' as', as = a'#as' ∧ (-High-) as' \rightarrow^* n ∧ kind a' = (\lambda s. True)√
  proof(induct n \equiv (-Entry-) as n rule: path.induct)
    case (Cons-path n'≡ (-Entry-) as n)
    from \equiv (-Entry-) as \rightarrow^* n' (inner-node n') have n'' ≠ (-Exit-)
      by (fastforce simp: inner-node-def)
    with (valid-edge a) (targetnode a = n'') (sourcenode a = (-Entry-))
    have n'' = (-High-) by (drule Entry-edge-Exit-or-High, auto)
    from High-target-Entry-edge
    obtain a' where valid-edge a' and sourcenode a' = (-Entry-)
    and targetnode a' = (-High-) and kind a' = (\lambda s. True)√
    by blast
    with (valid-edge a) (sourcenode a = (-Entry-)) (targetnode a = n'')
    (n'' = (-High-))
    have a = a' by (auto dest: edge-det)
    with (n'' = as \rightarrow^* n') (n'' = (-High-)) (kind a' = (\lambda s. True)√) show ?case by blast
  qed fastforce
qed

lemma Exit-path-Low-path:
assumes n as \rightarrow^* (-Exit-) and inner-node n
obtains a' as' where as = as''@[a'] and n as' \rightarrow^* (-Low-)
and kind a' = (\lambda s. True)√
proof(atomize-elim)
  from \equiv (-Exit-) as \rightarrow^* n
  show \exists as' a', as = as''@[a'] ∧ n as' \rightarrow^* (-Low-) ∧ kind a' = (\lambda s. True)√
  proof(induct as rule: rev-induct)
    case Nil

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

next
case \((\text{snoc } a' \text{ as}')\)
from \(n \xrightarrow{\text{as}'} [a'] \rightarrow\ast \text{ (Exit-)}\):
have \(n \xrightarrow{\text{as}'} \rightarrow\ast \text{ sourcenode } a'\) and \(\text{valid-edge } a'\) and \(\text{targetnode } a' = \text{ (Exit-)}\)
by (auto elim:path-split-snoc)
\{
assume \(\text{sourcenode } a' = \text{ (Entry-)}\)
with \(n \xrightarrow{\text{as}'} \rightarrow\ast \text{ sourcenode } a'\)
by (blast intro!:path-Entry-target)
with \(\text{inner-node } n\) have \(\text{False}\)
by (simp add:inner-node-def)
\}
with \(\text{valid-edge } a'\) \(\text{ (targetnode } a' = \text{ (Exit-)}\) have \(\text{sourcenode } a' = \text{ (Low-)}\)
by (blast dest!:Exit-edge-Entry-or-Low)
from Low-source-Exit-edge
obtain \(ax\) where \(\text{valid-edge } ax\) and \(\text{sourcenode } ax = \text{ (Low-)}\)
and \(\text{targetnode } ax = \text{ (Exit-)}\) and \(\text{kind } ax = (\lambda s. \text{True})\)
by blast
with \(\text{valid-edge } a'\) \(\text{ (targetnode } a' = \text{ (Exit-)}\) \(\text{ sourcenode } a' = \text{ (Low-)}\)
\(\text{ have } a' = ax\)
by (fastforce intro:edge-det)
with \(n \xrightarrow{\text{as}'} \rightarrow\ast \text{ sourcenode } a'\) \(\text{ sourcenode } a' = \text{ (Low-)}\) \(\text{ kind } ax = (\lambda s. \text{True})\)

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

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

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

2.2 Low Equivalence

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

definition lowEquivalence :: 'state 
\Rightarrow 'state \Rightarrow bool
\((\text{infixl } \approx_L 50)\)
where \(s \approx_L s' \equiv \forall V \in L. \text{state-val } s \; V = \text{state-val } s' \; V\)

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

lemma relevant-vars-Entry:
assumes \(V \in rv \text{ (Entry-)}\) and \((\text{High-}) \notin \text{backward-slice } S\)
shows \(V \in L\)
proof –
from \( V \in \text{rv} \ S \) \((-\text{Entry})\) obtain as \( n' \) where \( (-\text{Entry}) = as \rightarrow^* n' \) and \( n' \in \text{backward-slice} \ S \) and \( V \in \text{Use} \ n' \) and \( \forall nx \in \text{set}(\text{sourcenodes as}) \) \( V \notin \text{Def} \ nx \) by (erule \( \text{rvE} \))

from \((-\text{Entry}) = as \rightarrow^* n' \) 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

- 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 \( \langle (-\text{Entry}) = as \rightarrow^* n' \rangle \) obtain \( a' \) as \( \text{as} = a' \#as' \)
  
  and \( (-\text{High}) = as' \rightarrow^* n' \) by (erule \( \text{Entry-path-High-path} \))

  from \( \langle (-\text{Entry}) = as \rightarrow^* n' \rangle \) (as = a' #as') have sourcenode \( a' = (-\text{Entry}) \) by (fastforce elim: path.cases)

  show \( ?\text{thesis} \)

  proof (cases \( as' = [] \))

  - case True
    
    with \( \langle (-\text{High}) = as' \rightarrow^* n' \rangle \) have \( n' = (-\text{High}) \) by fastforce
    
    with \( \langle n' \in \text{backward-slice} \ S \rangle \) \( (-\text{High}) \notin \text{backward-slice} \ S \)
    
    have False by simp
    
    thus \( ?\text{thesis} \) by simp

  next

  - case False
    
    with \( \langle (-\text{High}) = as' \rightarrow^* n' \rangle \) have \( \text{hd} (\text{sourcenodes as'}) = (-\text{High}) \)
      
    by (rule \( \text{path-sourcenode} \))

    from False have \( \text{hd} (\text{sourcenodes as'}) \in \text{set} (\text{sourcenodes as'}) \)
      
    by (fastforce intro:hd-in-set simp:sourcenodes-def)

    with \( (\text{as} = a' \#as') \) have \( \text{hd} (\text{sourcenodes as'}) \in \text{set} (\text{sourcenodes as}) \)
      
    by (simp add:sourcenodes-def)

    with \( \langle \text{hd} (\text{sourcenodes as'}) = (-\text{High}) \rangle \) \( \forall nx \in \text{set} (\text{sourcenodes as}) \) \( V \notin \text{Def} \ nx \)
      
    have \( V \notin \text{Def} (-\text{High}) \) by fastforce
    
    hence \( V \notin \text{H} \) 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 \( \forall V \in \text{rv} \ S \) \( (-\text{Entry}) \), state-val \( s \) \( V = \text{state-val} \ s' \) \( V \)

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

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

show ?case
proof (cases \( L = \{ \} \))
  case True with \( \text{UseLow} \) show ?thesis by simp
next
  case False
  show ?thesis
proof (cases as')
  case Nil
  with \( \langle n - \text{as'} \rightarrow^* (-\text{Low}) \rangle \) have \( n = (-\text{Low}) \) by fastforce
  with \( \langle \text{valid-edge } a \langle \text{sourcenode } a = n \rangle \text{ have } \text{targetnode } a = (-\text{Exit}) \rangle \)
  by \( -(\text{rule Exit-successor-of-Low}, \text{simp}) \)
  from \( \text{Low-source-Exit-edge} \) obtain \( ax \) where \( \text{valid-edge } ax \)
    and \( \text{sourcenode } ax = (-\text{Low}) \) and \( \text{targetnode } ax = (-\text{Exit}) \)
    and \( \text{kind } ax = (\lambda s. \text{True}) \) by blast
  from \( \langle \text{valid-edge } a \langle \text{sourcenode } a = n \rangle \langle n = (-\text{Low}) \rangle \langle \text{targetnode } a = (-\text{Exit}) \rangle \)
    \( \langle \text{valid-edge } ax \langle \text{sourcenode } ax = (-\text{Low}) \rangle \langle \text{targetnode } ax = (-\text{Exit}) \rangle \)
  have \( a = ax \) by (fastforce dest: edge-det)
  with \( \langle \text{kind } ax = (\lambda s. \text{True}) \rangle \) have \( \text{kind } a = (\lambda s. \text{True}) \) by simp
  with \( \langle \text{targetnode } a = (-\text{Exit}) \rangle \langle \text{targetnode } a = n'' \rangle \langle n'' - \text{as'} \rightarrow^* (-\text{Low}) \rangle \)
  have \( (-\text{Low}) = (-\text{Exit}) \) by \( -(\text{rule path-Exit-source, auto}) \)
  with \( \text{False} \) have \( \text{False} \) by \( -(\text{drule Low-neq-Exit, simp}) \)
  thus ?thesis by simp
next
  case \( \langle \text{Cons } ax \text{ asx} \rangle \)
  with \( \langle n - \text{as'} \rightarrow^* (-\text{Low}) \rangle \) have \( n = \text{sourcenode } ax \) and \( \text{valid-edge } ax \)
    and \( \text{targetnode } ax - \text{asx'} \rightarrow^* (-\text{Low}) \) by (auto elim: path-split-Cons)
  show ?thesis
proof (cases \( \text{targetnode } ax = n'' \))
  case True
  with \( \langle \text{targetnode } ax - \text{asx} \rightarrow^* (-\text{Low}) \rangle \) have \( n'' - \text{asx} \rightarrow^* (-\text{Low}) \) by simp
  from \( \langle \text{valid-edge } ax \langle \text{valid-edge } a \langle n = \text{sourcenode } ax \rangle \langle \text{sourcenode } a = n \rangle \rangle \)
    \( \langle \text{True} \rangle \langle \text{targetnode } a = n'' \rangle \) have \( ax = a \) by (fastforce dest: edge-det)
  from \( \langle \text{preds } \text{(slice-kinds } S \text{ (a as))} \rangle \langle s \rangle \)
  have \( \text{preds1: preds } \text{(slice-kinds } S \text{ as)} \text{ (transfer } \text{(slice-kind } S \text{ a}) \rangle \langle s \rangle \)
    by (simp add: slice-kinds-def)
  from \( \langle \text{preds } \text{(slice-kinds } S \text{ as')} \rangle \langle s' \rangle \text{ Cons } \langle ax = a \rangle \)
  have \( \text{preds2: preds } \text{(slice-kinds } S \text{ asx)} \)
    \( \langle \text{transfer } \text{(slice-kind } S \text{ a}) \rangle \langle s' \rangle \)
    by (simp add: slice-kinds-def)
  from \( \langle \text{valid-edge } a \langle \text{sourcenode } a = n \rangle \langle \text{targetnode } a = n'' \rangle \rangle \)
    \( \langle \text{preds } \text{(slice-kinds } S \text{ (a as))} \rangle \langle \text{preds } \text{(slice-kinds } S \text{ as'}) \rangle \langle s' \rangle \)
    \( \langle ax = a \rangle \text{ Cons } \langle \forall V \in \text{rv } S \exists n. \text{state-val } s \text{ V } = \text{state-val } s' \text{ V} \rangle \)
  have \( \forall V \in \text{rv } S \langle n'' \rangle \text{ state-val } \text{(transfer } \text{(slice-kind } S \text{ a}) \rangle \langle s \rangle \text{ V } = 
\text{state-val } \text{(transfer } \text{(slice-kind } S \text{ a}) \rangle \langle s' \rangle \text{ V} \)
    by \( -(\text{rule rv-edge-slice-kinds, auto}) \)
  from \( \text{IH } [\text{OF } \langle n'' - \text{asx} \rightarrow^* (-\text{Low}) \rangle \text{ this } \text{preds1 preds2}] \)
In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, (-High-) ∈ backward-slice S, where (-Low-) ∈ S, makes sure that no high variable (which are all defined in (-High-)) can influence a low variable (which are all used in (-Low-)).

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

**Lemma nonInterference-path-to-Low:**

**Assumes**

s ≈_L s’ and (-High-) ∉ backward-slice S and (-Low-) ∈ S

and (-Entry-) − as→* (-Low-) and preds (kinds as) s

and (-Entry-) − as’→* (-Low-) and preds (kinds as’) s’

**Shows**

transfers (kinds as) s ≈_L transfers (kinds as’) s’

**Proof**

from (-Entry-) − as→* (-Low-) preds (kinds as) s)

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

and ∀ V ∈ Use (-Low-). state-val(transfers (slice-kinds S asx) s) V = state-val(transfers (kinds as) s) V

and slice-edges S as = slice-edges S asx

and (-Entry-) − asx→* (-Low-) by(erule fundamental-property-of-static-slicing)

from (-Entry-) − as’→* (-Low-) preds (kinds as’) s’ (-Low-) ∈ S

obtain asx’ where preds (slice-kinds S asx’) s’

and ∀ V ∈ Use (-Low-). state-val(transfers (slice-kinds S asx’) s’) V = state-val(transfers (kinds as’) s’) V

and slice-edges S as’ = slice-edges S asx’

and (-Entry-) − asx’→* (-Low-) by(erule fundamental-property-of-static-slicing)

from (s ≈_L s’ (-High-) ∉ backward-slice S)

have ∀ V ∈ rv S (-Entry-). state-val s V = state-val s’ V

by(rule lowEquivalence-relevant-nodes-Entry)

with (-Entry-) − asx→* (-Low-) (-Entry-) − asx’→* (-Low-) (-Low-) ∈ S

obtain preds (slice-kinds S asx) s’ preds (slice-kinds S asx’) s’

have ∀ V ∈ Use (-Low-). state-val(transfers (slice-kinds S asx) s) V = state-val (transfers (slice-kinds S asx’) s’) V

by -(rule rv-Low-Use-Low-auto)
\[
\begin{align*}
\text{with } & \forall V \in \text{Use } \langle -\text{Low} \rangle, \text{state-val}(\text{transfers } \langle \text{slice-kinds } S \text{ as } x \rangle s) \Rightarrow V = \\
& \text{state-val}(\text{transfers } \langle \text{kinds as } s \rangle s') \Rightarrow V \\
\forall V \in \text{Use } \langle -\text{Low} \rangle, & \text{state-val}(\text{transfers } \langle \text{slice-kinds } S \text{ as } x' \rangle s') \Rightarrow V = \\
& \text{state-val}(\text{transfers } \langle \text{kinds as'} s' \rangle s') \Rightarrow V \\
\text{show } & \text{thesis by}(\text{auto simp:lowEquivalence-def UseLow}) \\
\text{qed}
\end{align*}
\]

**Theorem** nonInterference-path:

assumes \(s \approx_L s'\) and \((-\text{High}) \notin \text{backward-slice } S\) and \((-\text{Low}) \in S\)
and \((-\text{Entry}) - as \rightarrow* (-\text{Exit})\) and \(\text{preds } \langle \text{kinds as } s \rangle\)
and \((-\text{Entry}) - as' \rightarrow* (-\text{Exit})\) and \(\text{preds } \langle \text{kinds as'} s' \rangle\)

**Proof**

from \((-\text{Entry}) - as \rightarrow* (-\text{Exit})\) obtain \(x x s\) where \(as = x#xs\)
and \((-\text{Entry})\) = sourcenode \(x\) and valid-edge \(x\)
and targetnode \(x\) \(-xs \rightarrow* (-\text{Exit})\)
apply(cases \(as = []\))
apply(simp, drule empty-path-nodes, drule Entry-noteq-Exit, simp)
by(erule path-split-Cons)
from \(\forall\text{valid-edge } x\) have valid-node (targetnode \(x\)) by simp
hence inner-node (targetnode \(x\))
proof(cases rule:valid-node-cases)
case Entry
with \(\forall\text{valid-edge } x\) have False by(rule Entry-target)
thus \(\text{thesis by simp}\)
next
case Exit
with \(\forall\text{targetnode } x \ -xs \rightarrow* (-\text{Exit})\): have \(xs = []\)
by -(rule path-Exit-source, simp)
from Entry-Exit-edge obtain \(z\) where valid-edge \(z\)
and sourcenode \(z\) = (-Entry) and targetnode \(z\) = (-Exit)
and \(\lambda s. False\) by blast
from (valid-edge \(x\)) (valid-edge \(z\)) (-Entry) = sourcenode \(x\)
(sourcenode \(z\) = (-Entry)) Exit \(\forall\text{targetnode } z\) = (-Exit):
have \(x = z\) by(fastforce intro:edge-det)
with \(\text{preds } \langle \text{kinds as } s \rangle\) s: \(\langle as = x#xs \rangle(xs = [])\) \(\langle \text{kind } z = (\lambda s. False)\rangle\)
have False by(simp add: kinds-def)
thus \(\text{thesis by simp}\)
qed simp

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

with (\(\text{targetnode y} \(-ys\rightarrowι\) (-Exit-)) obtain y' ys' where ys = ys'@[y]
and targetnode y \(-ys'\rightarrowι\) (-Low- and kind y' = (\(\lambda s. \text{True}\)))
by (fastforce elim: Exit-path-Low-path)
with (-Entry-) = sourcenode y (valid-edge y)
have (-Entry-) \(-y#ys'\rightarrowι\) (-Low-) by (fastforce intro: Cons-path)
from (as' = y#ys) (ys = ys'@[y]) have as' = (y#ys')@[y] by simp
with \(\text{preds} (\text{kinds as'})\) s' have \(\text{preds} (\text{kinds} (y#ys'))\) s'
by (simp add: kinds-def preds-split)
from (s \(\approx_L\) s') (-High-) \notin backward-slice S (-Low-) \(\in S\)
(-Entry-) \(-x#xs'\rightarrowι\) (-Low-) \(\langle\text{preds} (\text{kinds} (x#xs'))\) s
(-Entry-) \(-y#ys'\rightarrowι\) (-Low-) \(\langle\text{preds} (\text{kinds} (y#ys'))\) s'
have transfers (kinds (x#xs')) s \(\approx_L\) transfers (kinds (y#ys')) s'
by (rule nonInterference-path-to-Low)
with (as = x#xs) (xs = xs'@[x]) \(\langle\text{kind} x' = (\lambda s. \text{True})\rangle\)
(\(\text{as'} = y#ys\)) (ys = ys'@[y]) \(\langle\text{kind} y' = (\lambda s. \text{True})\rangle\)
show ?thesis by (simp add: kinds-def transfers-split)
qed

end

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

`locale NonInterferenceIntra = NonInterferenceIntraGraph sourcenode targetnode kind valid-edge Entry
Def Use state-val backward-slice Exit H L High Low + BackwardSlice-ref sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice sem identifies
for sourcenode :: 'edge ⇒ 'node and targetnode :: 'edge ⇒ 'node and kind :: 'edge ⇒ 'state edge-kind and valid-edge :: 'edge ⇒ bool and Entry :: 'node ('('Entry'-')') and Def :: 'node ⇒ 'var set and Use :: 'node ⇒ 'var set and state-val :: 'state ⇒ 'var and backward-slice :: 'node set ⇒ 'node set and sem :: 'com ⇒ 'state ⇒ 'com ⇒ 'state ⇒ bool (((1,(.-)/)) ⇒/ ((1,(.-)/))) [0,0,0,0] S1) 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 = \uparrow id
begin

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

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

`theorem nonInterference:
assumes s_1 \approx L, s_2 and (-High-) \notin backward-slice S and (-Low-) \in S
and valid-edge a and sourcenode a = (-High-) and targetnode a = n
and kind a = (λs. True)\_ and n = c and final c'
and (c,s_1) ⇒ (c',s_1') and (c,s_2) ⇒ (c',s_2')
shows s_1 \approx L, s_2`

`proof`

`from High-target-Entry-edge obtain ax where valid-edge ax
and sourcenode ax = (-Entry-) and targetnode ax = (-High-)
and kind ax = (λs. True)\_ by blast
from (n \triangleq c) (c,s_1) ⇒ (c',s_1')
obtain n_1 as_1 where n \not\rightarrow as_1 \rightarrow n_1 and transfers (kinds as_1) s_1 = s_1'
and preds (kinds as_1) s_1 and n_1 \triangleq c'
by (fastforce dest: fundamental-property)
from (n \not\rightarrow as_1 \rightarrow n_1) (valid-edge a) (sourcenode a = (-High-)) (targetnode a = n)
have (-High-) \not\rightarrow a as_1 \rightarrow n_1 by (rule Cons-path)
from final c' (n_1 \triangleq c')
obtain a_1 where valid-edge a_1 and sourcenode a_1 = n_1`
and targetnode $a_1 = \langle -Low- \rangle$ and kind $a_1 = \uparrow id$ by (fastforce dest: final-edge-Low)
hence $n_1 - [a_1] \rightarrow * \langle -Low- \rangle$ by (fastforce intro: path-edge)
with $\langle -High- \rangle - a \# as_1 \rightarrow * n_1$ have $\langle -High- \rangle = \langle -a \# as_1 @ [a_1] \rightarrow * \langle -Low- \rangle \rangle$
by (rule path-Append)
with $\langle valid-edge ax \rangle \langle source node ax = \langle -Entry- \rangle \rangle \langle target node ax = \langle -High- \rangle \rangle$
have $\langle -Entry- \rangle - ax \# (\langle a \# as_1 @ [a_1] \rangle \rightarrow *) \langle -Low- \rangle$ by $\langle -rule Cons-path \rangle$
from $\langle kind ax = (λs. True) \rangle \langle kind a = (λs. True) \rangle \langle preds (kinds as_1) s_1 \rangle$
\begin{itemize}
  \item $\langle kind a_1 = \uparrow id \rangle$ have preds (kinds $\langle ax \# (\langle a \# as_1 @ [a_1] \rangle) \rangle$) $s_1$
  \item by (simp add: kinds-def preds-split)
\end{itemize}
from $\langle n \triangleq c \rangle \langle \langle c, s_2 \rangle \Rightarrow (c', s_2') \rangle$
obtain $n_2 as_2$ where $n - as_2 \rightarrow * n_2$ and transfers (kinds $as_2$) $s_2 = s_2'$
and preds (kinds $as_2$) $s_2$ and $n_2 \triangleq c'$
by (fastforce dest: fundamental-property)
from $\langle n - as_2 \rightarrow * n_2 \rangle \langle valid-edge a \rangle \langle source node a = \langle -Low- \rangle \rangle \langle target node a = n \rangle$
have $\langle -High- \rangle - a \# as_2 \rightarrow * n_2$ by (rule Cons-path)
from $\langle final c' \langle c, s_2 \rangle \Rightarrow (c', s_2) \rangle$
obtain $n_2 as_2$ where valid-edge $a_2$ and source node $a_2 = n_2$
and targetnode $a_2 = \langle -Low- \rangle$ and kind $a_2 = \uparrow id$ by (fastforce dest: final-edge-Low)
hence $n_2 - [a_2] \rightarrow * \langle -Low- \rangle$ by (fastforce intro: path-edge)
with $\langle -High- \rangle - a \# as_2 \rightarrow * n_2$ have $\langle -High- \rangle - \langle a \# as_2 @ [a_2] \rightarrow * \langle -Low- \rangle \rangle$
by (rule path-Append)
with $\langle valid-edge ax \rangle \langle source node ax = \langle -Entry- \rangle \rangle \langle target node ax = \langle -High- \rangle \rangle$
have $\langle -Entry- \rangle - ax \# (\langle a \# as_2 @ [a_2] \rangle \rightarrow *) \langle -Low- \rangle$ by $\langle -rule Cons-path \rangle$
from $\langle kind ax = (λs. True) \rangle \langle kind a = (λs. True) \rangle \langle preds (kinds as_2) s_2 \rangle$
\begin{itemize}
  \item $\langle kind a_2 = \uparrow id \rangle$ have preds (kinds $\langle ax \# (\langle a \# as_2 @ [a_2] \rangle) \rangle$) $s_2$
  \item by (simp add: kinds-def preds-split)
\end{itemize}
from $\langle s_1 \approx_L s_2 \rangle \langle \langle -Low- \rangle \notin \text{backward-slice } S \rangle \langle \langle -Low- \rangle \in S \rangle$
\begin{itemize}
  \item $\langle -Entry- \rangle - ax \# (\langle a \# as_1 @ [a_1] \rangle) \rightarrow * \langle -Low- \rangle$ \langle preds (kinds $\langle ax \# (\langle a \# as_1 @ [a_1] \rangle) \rangle$) $s_1 \rangle$
  \item $\langle -Entry- \rangle - ax \# (\langle a \# as_2 @ [a_2] \rangle) \rightarrow * \langle -Low- \rangle$ \langle preds (kinds $\langle ax \# (\langle a \# as_2 @ [a_2] \rangle) \rangle$) $s_2 \rangle$
\end{itemize}
\begin{itemize}
  \item have transfers (kinds $\langle ax \# (\langle a \# as_1 @ [a_1] \rangle) \rangle$) $s_1 \approx_L$
  \item transfers (kinds $\langle ax \# (\langle a \# as_2 @ [a_2] \rangle) \rangle$) $s_2$
\end{itemize}
by (rule nonInterference-path-to-Low)
with $\langle kind ax = (λs. True) \rangle \langle kind a = (λs. True) \rangle \langle kind a_1 = \uparrow id \rangle \langle kind a_2 = \uparrow id \rangle$
\begin{itemize}
  \item transfers (kinds $as_1$) $s_1 = s_1'$ \langle transfers (kinds $as_2$) $s_2 = s_2' \rangle$
\end{itemize}
show $?thesis$ by (simp add: kinds-def transfers-split)
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** \( '\text{node} \ LDCFG\text{-node} = \text{Node} \ | \ \text{NewEntry} \ | \ \text{NewExit} \)**

**type-synonym** \( ( '\text{edge}, '\text{node}, '\text{state}) \ LDCFG\text{-edge} = '\text{node} \ LDCFG\text{-node} \times ( '\text{state edge-kind}) \times '\text{node} \ LDCFG\text{-node} \)**

#### 3.1.2 Lifting valid-edge

**inductive** \( \text{lift-valid-edge} :: ( '\text{edge} \Rightarrow \text{bool} ) \Rightarrow ( '\text{edge} \Rightarrow '\text{node} ) \Rightarrow ( '\text{edge} \Rightarrow '\text{node} ) \Rightarrow ( '\text{edge} \Rightarrow '\text{state edge-kind} ) \Rightarrow '\text{node} \Rightarrow '\text{node} \Rightarrow ( '\text{edge}, '\text{node}, '\text{state} ) \ LDCFG\text{-edge} \Rightarrow \text{bool} \)**

**for** \( \text{valid-edge} :: '\text{edge} \Rightarrow \text{bool} \text{ and src} :: '\text{edge} \Rightarrow '\text{node} \text{ and trg} :: '\text{edge} \Rightarrow '\text{node} \text{ and knd} :: '\text{edge} \Rightarrow '\text{state edge-kind} \text{ and E} :: '\text{node} \text{ and X} :: '\text{node} \)**

**where** \( \text{lve-edge} : [ \text{valid-edge} \ a; \ \text{src} \ a \neq E \lor \text{try} \ a \neq X ; \ e = ( \text{Node} \ ( \text{src} \ a) \ , \ \text{knd} \ a \ , \ \text{Node} \ ( \text{try} \ a)) ] ] \Rightarrow \text{lift-valid-edge valid-edge src try knd E X e} \)

| \( \text{lve-Entry-edge} : \ e = ( \text{NewEntry} \ ( \lambda s. \text{True})) \lor \text{Node} \ E \) \Rightarrow \text{lift-valid-edge valid-edge src try knd E X e} |

| \( \text{lve-Exit-edge} : \ e = ( \text{Node} \ X \ ( \lambda s. \text{True})) \lor \text{NewExit} \) \Rightarrow \text{lift-valid-edge valid-edge src try knd E X e} |

| \( \text{lve-Entry-Exit-edge} : \ e = ( \text{NewEntry} \ ( \lambda s. \text{False})) \lor \text{NewExit} \) \Rightarrow \text{lift-valid-edge valid-edge src try knd E X e} |

**lemma** \( \text{[simp]} : \neg \text{lift-valid-edge valid-edge src try knd E X (Node E, et, Node X)} \text{ by(auto elim:lift-valid-edge.cases)} \)**
### 3.1.3 Lifting Def and Use sets

**Inductive-set** lift-Def-set :: (′node ⇒ ′var set ⇒ ′node ⇒ ′var set ⇒ (′node LDCFG-node × ′var) set

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

where

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

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

**Abbreviation** lift-Def :: (′node ⇒ ′var set ⇒ ′node ⇒ ′var set ⇒ (′node LDCFG-node ⇒ ′var set

where

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

**Inductive-set** lift-Use-set :: (′node ⇒ ′var set ⇒ ′node ⇒ ′var set ⇒ (′node LDCFG-node × ′var) set

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

where

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

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

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

**Abbreviation** lift-Use :: (′node ⇒ ′var set ⇒ ′node ⇒ ′var set ⇒ (′node LDCFG-node ⇒ ′var set

where

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

### 3.2 The lifting lemmas

#### 3.2.1 Lifting the basic locales

**Abbreviation** src :: (′edge,′node,′state) LDCFG-edge ⇒ ′node LDCFG-node

where src a ≡ fst a

**Abbreviation** trg :: (′edge,′node,′state) LDCFG-edge ⇒ ′node LDCFG-node

where trg a ≡ snd(snd a)

**Definition** knd :: (′edge,′node,′state) LDCFG-edge ⇒ ′state edge-kind

where knd a ≡ fst(snd a)

---

14
lemma lift-CFG:
assumes wf : CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
state-val Exit
shows CFG src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
state-val Exit
by (rule wf)
show ?thesis
proof
  fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a 
  and trg a = NewEntry 
  thus False by (fastforce elim: lift-valid-edge.cases)
next
  fix a a' assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a 
  and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' 
  and src a = src a' and trg a = trg a' 
  thus a = a' 
proof (induct rule: lift-valid-edge.induct)
  case lve-edge thus \( \vdash \) 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 : CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
state-val Exit
shows CFG-wf src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry 
(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use
state-val Exit
by (rule wf)
interpret CFG:CFG src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry 
by (fastforce intro: lift-CFG wf)
show ?thesis
proof
  show lift-Def Def Entry Exit H L NewEntry = \{\} 
  lift-Use Use Entry Exit H L NewEntry = \{\} 
  by (fastforce elim: lift-Use-set.cases lift-Def-set.cases)
next
  fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V /∈ lift-Def Def Entry Exit H L (src a) and pred (kind a) s
thus state-val (transfer (kind a) s) V = state-val s V
proof (induct rule: lift-valid-edge.induct)
case lve-edge
  thus ?case by (fastforce intro: CFG-edge-no-Def-equal dest: lift-Def-node[of -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 (lve-edge a e)
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 (fastforce 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 \( \forall V \in \text{lift-Use} \) Use Entry Exit H L (src e).
state-val \( s \) \( V \) = state-val \( s' \) \( V \) \( \forall V \in H \)
have state-val \( s \) \( V \) = state-val \( s' \) \( V \) by simp
with \( e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)}) \)
\( \exists Q. \text{kind a} = (Q) \)
show \( \text{thesis} \) by (fastforce simp: knd-def)
next
case False
\{ fix \( V' \) assume \( V' \in \text{Use} \) (sourcenode a)
with \( e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)}) \)
have \( V' \in \text{lift-Use} \) Use Entry Exit H L (src e)
by (fastforce intro: lift-Use-node)
\}
with \( \forall V \in \text{lift-Use} \) Use Entry Exit H L (src e).
state-val \( s \) \( V \) = state-val \( s' \) \( V \)
by fastforce
from \( \text{valid-edge a} \) this \( \text{pred (kind e) s} \) \( \text{pred (kind e) s}' \)
\( e = (\text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)}) \)
have \( \forall V \in \text{Def} \) (sourcenode a), state-val \( (\text{transfer (kind a) s}) \) \( V \) =
state-val \( (\text{transfer (kind a) s}') \) \( V \)
by \( \text{erule CFG-edge-transfer-uses-only-Use, auto simp: knd-def} \)
from \( V \in \text{lift-Def} \) Def Entry Exit H L (src e), False
\( e = (\text{Node (sourcenode a), kind a, Node (targetnode a)}) \)
have \( V \in \text{Def} \) (sourcenode a) by (fastforce elim: lift-Def-set.cases)
with \( \forall V \in \text{Def} \) (sourcenode a), state-val \( (\text{transfer (kind a) s}) \) \( V \) =
state-val \( (\text{transfer (kind a) s}') \) \( V \)
\( e = (\text{Node (sourcenode a), kind a, Node (targetnode a)}) \)
show \( \text{thesis} \) by (simp add: knd-def)
qed
next
case (lve-Entry-edge e)
from \( V \in \text{lift-Def} \) Def Entry Exit H L (src e)
\( e = (\text{NewEntry}, (\lambda s. \text{True}) \), \text{Node Entry}) \)
have False by (fastforce elim: lift-Def-set.cases)
thus \( \text{?case by simp} \)
next
case (lve-Exit-edge e)
from \( V \in \text{lift-Def} \) Def Entry Exit H L (src e)
\( e = (\text{Node Exit}, (\lambda s. \text{True}) \), \text{NewExit}) \)
have False
by (fastforce elim: lift-Def-set.cases intro!: Entry-noteq-Exit simp: Exit-empty)
thus \( \text{?case by simp} \)
qed (simp add: knd-def)
next
case False
fix a s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \( \text{pred} (\text{knd} a) \) 
and \( \forall V \in \text{lift-Use} \) \( \text{Use} \) \( \text{Exit} \) \( H \) \( L \) \( (\text{src} a) \). \( \text{state-val} s V = \text{state-val} s' V \) 
thus \( \text{pred} (\text{knd} a) \) \( s' \) 
by \( \text{(induct rule:lift-valid-edge.induct,} 
\text{auto elim!:} \text{CFG-edge-Uses-pred-equal dest:lift-Use-node simp:knd-def}) \)

next 
fix \( a a' \) 
assume \( \text{lift-valid-edge valid-edge} \) \( \text{source-node} \) \( \text{target-node} \) \( \text{kind} \) \( \text{Entry} \) \( \text{Exit} \) \( a \) 
and \( \text{lift-valid-edge valid-edge} \) \( \text{source-node} \) \( \text{target-node} \) \( \text{kind} \) \( \text{Entry} \) \( \text{Exit} \) \( a' \) 
and \( \text{src} a = \text{src} a' \) \text{and} \( \text{try} a \neq \text{try} a' \) 
thus \( \exists Q Q'. \text{knd} a = (Q) \lor \land \text{knd} a' = (Q') \lor \land 
(\forall s. (Q s \rightarrow \neg Q' s) \land (Q' s \rightarrow \neg Q s)) \) 
proof \( \text{(induct rule:lift-valid-edge.induct)} \)

\text{case (lve-edge} a e) 
from \( \text{lift-valid-edge valid-edge} \) \( \text{source-node} \) \( \text{target-node} \) \( \text{kind} \) \( \text{Entry} \) \( \text{Exit} \) \( a' \) 
\text{valid-edge} a \( e = (\text{Node} \) \( \text{source-node} a \), \text{kind} a, \text{Node} \) \( \text{target-node} a \))\) 
\( \langle \text{src} e = \text{src} a' \rangle \langle \text{try} e \neq \text{try} a' \rangle \) 
show \?case 
proof \( \text{(induct rule:lift-valid-edge.induct)} \)

\text{case lve-edge thus ?case by}(\text{auto dest:deterministic simp:knd-def}) 
next 
\text{case (lve-Exit-edge} e') 
from \( e = (\text{Node} \) \( \text{source-node} a \), \text{kind} a, \text{Node} \) \( \text{target-node} a \))\) 
\( e' = (\text{Node} \) \( \text{Exit} \), \( \lambda s. \text{True} \), \text{NewExit}) \langle \text{src} e = \text{src} e' \rangle \) 
\text{have source-node} a = \text{Exit by simp} 
\text{with valid-edge} a \space{\text{have}} \text{False by}(\text{rule Exit-source}) 
thus \?case by simp 
qed auto 
qed \( \langle \text{fastforce elim:lift-valid-edge.cases simp:knd-def} \rangle + \)
qed 

\text{lemma lift-CFGExit:} 
\text{assumes} \text{wf:CFGExit-wf source-node target-node kind valid-edge Entry Def Use} 
\text{shows} \text{CFGExit src trg knd} \( (\text{lift-valid-edge valid-edge} \) \( \text{source-node} \) \( \text{target-node} \) \( \text{kind} \) \( \text{Entry} \) \( \text{Exit} \) \( \text{NewEntry} \)) 
\text{NewEntry NewExit} 
proof -- 
\text{interpret} \text{CFGExit-wf source-node target-node kind valid-edge Entry Def Use} 
\text{state-val Exit} 
by \( \text{rule} \text{wf} \) 
\text{interpret} \text{CFG:CFG src trg knd} \( \text{lift-valid-edge valid-edge} \) \( \text{source-node} \) \( \text{target-node} \) \( \text{kind} \) \( \text{Entry} \) \( \text{Exit} \) \( \text{NewEntry} \) 
by \( \text{fastforce intro:lift-CFG wf} \) 
show \?thesis 
proof 
fix a assume \text{lift-valid-edge valid-edge} \( \text{source-node} \) \( \text{target-node} \) \( \text{kind} \) \( \text{Entry} \) \( \text{Exit} \) \( a \)
and src a = NewExit
thus False by (fastforce elim:lift-valid-edge_cases)
next
from lve-Entry-Exit-edge
show ∃ a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
src a = NewEntry ∧ trg a = NewExit ∧ knd a = (λs. 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 = {} ∧
lift-Use Use Entry Exit H L NewExit = {} 
by (fastforce elim:lift-Use-set.cases lift-Def-set.cases)
qed
qed

3.2.2 Lifting wod-backward-slice

lemma lift-wod-backward-slice:
fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit and Def and Use and H and L
defines lve:lve ≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and lDef:lDef ≡ lift-Def Def Entry Exit H L
and lUse:lUse ≡ lift-Use Use Entry Exit H L
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
and H ∩ L = {} and H ∪ L = UNIV
shows NonInterferenceIntraGraph src try knd lve NewEntry lDef lUse state-val
proof –

interpret \( CFGExit\)-wf source node target kind valid-edge Entry Def Use state-val Exit

by (rule wf)

interpret \( CFGExit\)-wf:

\( CFGExit\)-wf src trg kind lve NewEntry lDef lUse state-val NewExit

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

from \( wf \ lve \) have \( CFG\): \( CFG \) src trg lve NewEntry lDef lUse state-val

by (fastforce intro: lift-CFG)

from \( wf \ lve \) lDef lUse have \( CFG\)-wf: \( CFG\)-wf src trg kind lve NewEntry lDef lUse state-val

by (fastforce intro: lift-CFG-wf)

show \( \text{thesis} \)

proof

fix \( n \ S \)

assume \( n \in CFG\)-wf.wod-backward-slice src trg lve lDef lUse S \)

with \( CFG\)-wf show \( CFG\).valid-node src trg lve n

by \( \neg \) (rule \( CFG\)-uf.wod-backward-slice-valid-node)

next

fix \( n \ S \) assume \( CFG\).valid-node src trg lve n and \( n \in S \)

with \( CFG\)-wf show \( n \in CFG\)-uf.wod-backward-slice src trg lve lDef lUse S

by \( \neg \) (rule \( CFG\)-uf.refl)

next

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

assume \( n' \in CFG\)-uf.wod-backward-slice src trg lve lDef lUse S

and \( CFG\)-uf.data-dependence src trg lve lDef lUse \( n \ V \ n' \)

with \( CFG\)-wf show \( n \in CFG\)-uf.wod-backward-slice src trg lve lDef lUse S

by \( \neg \) (rule \( CFG\)-uf.dd-closed)

next

fix \( n \ S \)

from \( CFG\)-wf

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

\( \bigcup (CFG\)-uf.wod-backward-slice src trg lve lDef lUse S) = \{ m \} \)

by (rule \( CFG\)-uf.obs-singleton)

thus finite

\( (CFG\).obs src trg lve n (CFG\)-uf.wod-backward-slice src trg lve lDef lUse S) = \}

by (rule \( CFG\)-uf.obs-singleton)

next

fix \( n \ S \)

from \( CFG\)-wf

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

\( \bigcup (CFG\)-uf.wod-backward-slice src trg lve lDef lUse S) = \{ m \} \)

by (rule \( CFG\)-uf.obs-singleton)
thus \( \text{card} \ (CFG_{obs} \ src \ trg \ lve \ n) \)

\[ (CFG_{wf}.wod-backward-slice \ src \ trg \ lve \ IUse \ S)) \leq 1 \]

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

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

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

by (fastforce simp:knd-def)

next

fix a assume lve a and src a = Node Exit

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

next

from IDef show IDef (Node Entry) = H

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

next

from Entry-not-eq-Exit IUse show IUse (Node Entry) = H

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

next

from Entry-not-eq-Exit IUse show IUse (Node Exit) = L

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

next

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

next

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

qed

qed

3.2.3 Lifting PDG-BS with standard-control-dependence

lemma lift-Postdomination:

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

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

and \( \text{inner} : CFG_{Exit.inner-node sourcennode targetnode valid-edge Entry Exit nx} \)
shows Postdomination src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry NewExit

proof —
interpret Postdomination sourcenode targetnode kind valid-edge Entry Exit
by(rule pd)
interpret CFG Exit-ps : CFG Exit-pf src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val NewExit)
by(fastforce intro:lift-CFG Exit-ps pf)

from pf 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)

next

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 [] n

by(fastforce intro:CFG:empty-path[OF CFG] simp:CFG:valid-node-def[OF CFG])

thus `thesis by blast

next

case NewExit

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,(λs. False),NewExit) by(fastforce intro:lve-Entry-Exit-edge)

with NewExit have CFG: path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry [(NewEntry,(λs. False),NewExit)] n

by(fastforce intro:CFG:Cons-path[OF CFG] CFG:empty-path[OF CFG]
simp:CFG:valid-node-def[OF CFG])

thus `thesis by blast

next

case (Node m)

with Entry-Exit-edge (CFG:valid-node src trg)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) n

have valid-node m
by(auto elim:lift-valid-edge.cases
simp:CFG:valid-node-def[OF CFG] valid-node-def)

thus `thesis

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

case Entry
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry,(λs. 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, (λs. 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: lift-valid-edge.lve-edge simp: inner-node-def)
  hence path: CFG.path src trg
    (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
    (Node (sourcenode ax)) [(Node (sourcenode ax), kind ax, Node Exit)]
    (Node Exit)
    by (fastforce intro: CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG]
      simp: CFG.valid-node-def[OF CFG])
  have edge: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
    (NewEntry, (λs. 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 = ⟨λax. [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
and \( \text{targetnode } x = \text{sourcenode } ax \) by (auto elim: path-split-snoc)

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

hence \( \text{targetnode } x \neq \text{Exit} \) by clarsimp
with \( \langle \text{valid-edge } x \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \) THEN sgm
have \( \langle \text{valid-edge } \text{sourcenode targetnode kind Entry Exit} \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \)
by (fastforce intro: lift-valid-edge. lve-edge)

hence path: CFG. path src trg
(\( \langle \text{valid-edge } \text{sourcenode targetnode kind Entry Exit} \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \))
by (fastforce intro: CFG. Cons-path[OF CFG])
from IH[\( \langle \text{Entry} \rightarrow \ast \text{sourcenode } x \rangle \langle \text{valid-edge } x \rangle \)] obtain es
where CFG. path src trg
(\( \langle \text{valid-edge } \text{sourcenode targetnode kind Entry Exit} \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \)) by blast
with path have CFG. path src trg
(\( \langle \text{valid-edge } \text{sourcenode targetnode kind Entry Exit} \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \)) by (rule CFG. path-Append[OF CFG])
thus ?case by blast
then obtain es where CFG. path src trg
(\( \langle \text{valid-edge } \text{sourcenode targetnode kind Entry Exit} \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \)) by blast
with inner have \( \langle \text{valid-edge } \text{sourcenode targetnode kind Entry Exit} \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \)
by (fastforce intro: CFG. Cons-path[OF CFG])
with Node Exit show ?thesis by fastforce
next
case inner
from \( \langle \text{valid-node } m \rangle \) obtain as \( \langle \text{Entry} \rightarrow \ast \text{m} \rangle \)
by (fastforce dest: Entry-path)
with inner have \( \exists \text{es. CFG. path src trg} \)
(\( \langle \text{valid-edge } \text{sourcenode targetnode kind Entry Exit} \rangle \langle \text{targetnode } x = \text{sourcenode } ax \rangle \))
(\( \langle \text{targetnode } x = \text{sourcenode } ax \rangle \))
proof (induct arbitrary: m rule: rev-induct)
case Nil
from (Entry −[]→∗ m) have m = Entry by fastforce
with lve-Entry-edge have CFG.path src try
  (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
  (Node Entry) [] (Node m)
by (fastforce intro: CFG.empty-path[OF CFG] simp: CFG.valid-node-def[OF CFG])
thus ?case by blast

next
case (snoc x xs)
  note IH = (∃ m. [inner-node m; Entry −xs→∗ m])
  |= 3 es. CFG.path src try
  (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
  (Node Entry) es (Node m)
from (Entry −xs@[x]→∗ m) have Entry −xs→∗ source-node x and valid-edge x and m = target-node x by (auto elim: path-split-snoc)
with (inner-node m)
  have edge: lift-valid-edge valid-edge source-node target-node kind Entry Exit
    (Node (source-node x), kind x, Node m)
  by (fastforce intro: lve-edge simp: inner-node-def)
  hence path: CFG.path src try
    (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
    (Node (source-node x)) [((Node (source-node 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 (source-node x) by simp
thus ?case proof (cases source-node x rule: valid-node-cases)
  case Entry
  with edge have CFG.path src try
    (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
    (Node Entry) [([Node Entry, kind x, Node m]) (Node m)]
  apply − apply (rule CFG.Cons-path[OF CFG])
  apply (rule CFG.empty-path[OF CFG])
  by (auto simp: CFG.valid-node-def[OF CFG])
  thus ?thesis by blast
next
case Exit
  with (valid-edge x) have False by (rule Exit-source)
  thus ?thesis by simp
next
case inner
from IH[OF this (Entry −xs→∗ source-node x)] obtain es
  where CFG.path src try
    (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
    (Node Entry) es (Node (source-node x)) by blast
  with path have CFG.path src try
    (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(Node Entry) \( es[@((\text{Node (sourcenode } x, \text{kind } x, \text{Node } m))] \) (Node m)
by \( \text{(rule CFG.path-Append[OF CFG])} \)
thus \(?thesis\) by blast
qed

then obtain \( es \) where path:CFG.path src try
(Node Entry) \( es \) (Node m) by blast
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,\( \langle \lambda s. \text{True} \rangle \),Node Entry) by(fastforce intro:lve-Entry-edge)
from this path Node have CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry ((NewEntry,\( \langle \lambda s. \text{True} \rangle \),Node Entry)#\( es \)) \( n \)
by(fastforce intro:CFG.Cons-path[OF CFG])
thus \(?thesis\) by blast
qed

next

fix \( n \) assume CFG.valid-node src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) \( n \)
show \( \exists as. \) CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
\( n \) as NewExit
proof(cases \( n \))
case NewEntry
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(NewEntry,\( \langle \lambda s. \text{False} \rangle \),NewExit) by(fastforce intro:lve-Entry-Exit-edge)
with NewEntry have CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
\( n \) [(NewEntry,\( \langle \lambda s. \text{False} \rangle \),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,\( \langle \lambda s. \text{False} \rangle \),NewExit) by(fastforce intro:lve-Entry-Exit-edge)
with NewExit have CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
\( n \) [] NewExit
by(fastforce intro:CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF CFG])
thus \(?thesis\) by blast
next
case \( (\text{Node } m) \)
with Entry-Exit-edge (CFG.valid-node src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) \( n \))
have valid-node \( m \)
by(auto elim:lift-valid-edge.cases
\[ \text{simp: } \text{CFG.valid-node-def[OF CFG] valid-node-def} \]

\text{thus } \text{thesis}

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

\text{case Entry}

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

\text{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)}}

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

\text{hence path:CFG.path src try}

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

(Node Exit) [(Node Exit,(\lambda s. True),NewExit)] (NewExit)

by(fastforce intro:CFG.Cons-path[OF CFG] CFG.empty-path[OF CFG] simp:CFG.valid-node-def[OF CFG])

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

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

\text{from this :valid-edge ax have \exists s. CFG.path src try}

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

(Node (targetnode ax)) es (Node Exit)

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

\text{case Nil}

\text{from (targetnode ax \rightarrow* Exit) have targetnode ax = Exit by fastforce}

\text{hence CFG.path src try}

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

(Node (targetnode ax)) [] (Node Exit)

apply simp apply(rule CFG.empty-path[OF CFG])

by(auto intro:lve-Exit-edge simp:CFG.valid-node-def[OF CFG])

\text{thus ?case by blast}

\text{next}

\text{case (Cons x xs)}

\text{note IH = } (\forall ax. [\text{targetnode ax \rightarrow x \rightarrow* Exit; valid-edge ax}] \implies \exists es. CFG.path src try}

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

(Node (targetnode ax)) es (Node Exit))

\text{from (targetnode ax \rightarrow x \rightarrow* Exit)}

\text{have targetnode x \rightarrow x \rightarrow* Exit and valid-edge x and sourcenode x = targetnode ax by(auto elim:path-split-Cons)}

\{ \text{assume sourcenode x = Entry with (valid-edge ax) (sourcenode x = targetnode ax)}

\text{have False by (rule Entry-target,simp+) } \}

\text{hence sourcenode x \neq Entry by clarsimp}

\text{with (valid-edge x) (sourcenode x = targetnode ax)[THEN sym]}

\text{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)}}

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

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{(Node (targetnode x)) es (Node Exit) by blast} \]

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

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{(Node (targetnode ax))} \]

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

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

qed

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

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{(Node (targetnode ax)) es (Node Exit) by blast} \]

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

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{(Node Entry) ((Node Entry, kind ax, Node (targetnode ax))#es) (Node Exit)} \]

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

with \text{path have} \( \text{CFG}.\text{path} \text{ src trg} \)

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{(Node Entry) ((Node Entry, kind ax, Node (targetnode ax))#es) (Node Exit)} \]

\[ \text{by (fastforce intro:lve-Exit-edge)} \]

\[ \text{with Exit Node show \(?\text{thesis by fastforce}\)} \]

next

\[ \text{case Exit} \]

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

\[ \text{(Node Exit,(\lambda s. True)\text{, NewExit}) by (fastforce intro:lve-Exit-edge)} \]

\[ \text{with \text{Exit Node have} \( \text{CFG}.\text{path} \text{ src trg} \)} \]

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{n ((Node Exit,(\lambda s. True)\text{, NewExit})) NewExit} \]

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

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

next

\[ \text{case inner} \]

\[ \text{from \( \langle \text{valid-node m} \rangle \text{ obtain as where} \text{ m - as→* Exit} \)} \]

\[ \text{by (fastforce dest:Exit-path)} \]

\[ \text{with inner have \( \exists \text{ es. CFG.path src trg} \)} \]

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{(Node m) es (Node Exit)} \]

proof\((\text{induct as arbitrary:m})\)

\[ \text{case Nil} \]

\[ \text{from m - [-→* Exit]} \]

\[ \text{have m = Exit by fastforce} \]

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

\[ \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)} \]

\[ \text{(Node m) [] (Node Exit)} \]

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

next

case (Cons x xs)

note IH = \(\forall m. [\text{inner-node } m; m - xs \rightarrow^* \text{Exit}]\)

\(\Rightarrow \exists es. \text{CFG.path src trg}\)

(lift-valid-edge valid-edge source-node target-node kind Entry Exit)

(Node m) es (Node Exit):

from \((m - x \# xs \rightarrow^* \text{Exit})\) have target-node x - xs \rightarrow^* \text{Exit}

and valid-edge x and \(m = \text{sourcenode } x\) by (auto elim: path-split-Cons)

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

have edge: lift-valid-edge valid-edge source-node target-node kind Entry Exit

(Node m, kind x, Node (target-node x))

by (fastforce intro: lve-edge simp: inner-node-def)

from (valid-edge x) have valid-node (target-node x) by simp

thus ?case

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

case Entry

with (valid-edge x) have False by (rule Entry-target)

thus ?thesis by simp

next

case Exit

with edge have CFG.path src trg

(lift-valid-edge valid-edge source-node target-node kind Entry Exit)

(Node m) \[\langle \text{Node m, kind x, Node (target-node x)\rangle} \) (Node Exit)

apply apply (rule CFG.Cons-path[OF CFG])

apply (rule CFG.empty-path[OF CFG])

by (auto simp: CFG.valid-node-def[OF CFG])

thus ?thesis by blast

next

case Inner

from IH[OF this \(\langle\text{target-node } x - xs \rightarrow^* \text{Exit}\rangle\)] obtain es

where CFG.path src trg

(lift-valid-edge valid-edge source-node target-node kind Entry Exit)

(Node (target-node x)) es (Node Exit) by blast

with edge have CFG.path src trg

(lift-valid-edge valid-edge source-node target-node kind Entry Exit)

(Node m) \(\langle \text{Node m, kind x, Node (target-node x)\rangle} \# es\) (Node Exit)

by (fastforce intro: CFG.Cons-path[OF CFG])

thus ?thesis by blast

qed

then obtain es where path: CFG.path src trg

(lift-valid-edge valid-edge source-node target-node kind Entry Exit)

(Node m) es (Node Exit) by blast

have lift-valid-edge valid-edge source-node target-node kind Entry Exit

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

hence CFG.path src trg

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

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

\[\text{with path Node have CFG.path src try}\]
\quad (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
\quad n\ (es\@[(\text{Node Exit}, (\lambda s. \text{True}), \text{NewExit})]) \text{ NewExit}

by (fastforce intro: CFG.path-Append[OF CFG])

thus ?thesis by blast

qed

lemma lift-PDG-scd:

assumes PDG\textcdot PDG source-node target-node kind valid-edge Entry Def Use state-val
\quad Exit
\quad (Postdomination.standard-control-dependence source-node target-node valid-edge Exit)
\quad and \text{pd}:Postdomination source-node target-node kind valid-edge Entry Exit
\quad and inner: CFGExit.inner-node source-node target-node valid-edge Entry Exit nx

shows PDG src try knd
\quad (lift-valid-edge valid-edge source-node target-node kind Entry Exit) \text{ NewEntry}
\quad (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
\quad (Postdomination.standard-control-dependence src try)
\quad (lift-valid-edge valid-edge source-node target-node kind Entry Exit) \text{ NewExit}

proof

\text{interpret PDG source-node target-node kind valid-edge Entry Def Use state-val Exit}
\quad Postdomination.standard-control-dependence source-node target-node
\quad valid-edge Exit

by (rule PDG)

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

from wf pd inner have pd\textcdot Postdomination src try knd
\quad (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
\quad NewEntry NewExit

by (rule lift-Postdomination)

from wf have CFG: CFG src try
\quad (lift-valid-edge valid-edge source-node target-node kind Entry Exit) \text{ NewEntry}

by (rule lift-CFG)

from wf have CFG-wf: CFG-wf src try knd
\quad (lift-valid-edge valid-edge source-node target-node kind Entry Exit) \text{ NewEntry}
\quad (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
\quad (lift-valid-edge valid-edge source-node target-node kind Entry Exit)
\quad NewEntry NewExit

by (rule lift-CFGExit)

from wf have CFGExit-wf: CFGExit-wf src try knd
\begin{verbatim}
(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 = { } \land
       lift-Use Use Entry Exit H L NewEntry = { } by(rule CFG-wf.Entry-empty)
next
  fix a V s
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and V \notin lift-Def Def Entry Exit H L (src a) and pred (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 asms: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  \forall V \in lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
  pred (knd a) s pred (knd a) s'
  with CFG-wf show \forall V \in lift-Def Def Entry Exit H L (src a).
       state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
  by(rule CFG-wf.CFG-edge-transfer-uses-only-Use)
next
  fix a s s'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and pred (knd a) s
  and \forall V \in lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
  with CFG-wf show pred (knd a) s' by(rule CFG-wf.CFG-edge-Uses-pred-equal)
next
  fix a a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
  and src a = src a' and try a \neq try a'
  with CFG-wf show \exists Q Q'. knd a = (Q) \land \land knd a' = (Q') \land
       (\forall s. (Q s \rightarrow \neg Q' s) \land (Q' s \rightarrow \neg Q s))
  by(rule CFG-wf.deterministic)
next
\end{verbatim}

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 ∧ kind a = (λs. False) ✓
  by (rule CFGExit.Entry-Exit-edge)
next
  from CFGExit-wf
  show lift-Def Def Entry Exit H L NewExit = {} ∧
        lift-Use Use Entry Exit H L NewExit = {}
  by (rule CFGExit-wf.Exit-empty)
next
  fix n n'
  assume scd: Postdomination.standard-control-dependence src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          NewExit n n'
  show n' ≠ NewExit
  proof (rule ccontr)
    assume ¬ n' ≠ NewExit
    hence n' = NewExit by simp
    with scd pd' show False
      by (fastforce intro: Postdomination.Exit-not-standard-control-dependent)
  qed
next
  fix n n'
  assume Postdomination.standard-control-dependence src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          NewExit n n'
  thus ∃ as. CFG.path src trg
          (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
          n as n' ∧ as ≠ []
  by (fastforce simp: Postdomination.standard-control-dependence-def[OF pd'])
  qed
qed

lemma lift-PDG-standard-backward-slice:
fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
  and Def and Use and H and L
defines lve::{lve} ≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  and lDef::lDef ≡ lift-Def Def Entry Exit H L
  and lUse::lUse ≡ lift-Use Use Entry Exit H L
assumes PDG: PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
          (Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)

and pd::Postdomination sourcenode targetnode kind valid-edge Entry Exit
and inner::CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
and $H \cap L = \{ \}$ and $H \cup L = \text{UNIV}$

shows NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val
  (PDG.PDG-BS src trg lve lDef lUse
   (Postdomination.standard-control-dependence src trg lve NewExit))
NewExit $H \ L$ (Node Entry) (Node Exit)

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

proof
  fix n S
  assume n \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.PDG-ddep-edge[OF PDG'] simp: PDG.PDG-BS-def[OF PDG']
next
  fix n S V
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 V
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 src trg lve lDef lUse)
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 = NewExit ∧ knd a = (λs. True)
  by (fastforce simp: knd-def)
next
  fix a assume lve a and src a = Node Exit
  with lve show trg a = NewExit by (fastforce elim: lift-valid-edge.cases)
next
  from lDef show lDef (Node Entry) = H
  by (fastforce elim: lift-Def-set.cases intro: lift-Def-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Entry) = H
  by (fastforce elim: lift-Use-set.cases intro: lift-Use-High)
next
  from Entry-noteq-Exit lUse show lUse (Node Exit) = L
  by (fastforce elim: lift-Use-set.cases intro: lift-Use-Low)
next
  from H ∩ L = {} show H ∩ L = {} .
next
  from H ∪ L = UNIV show H ∪ L = UNIV .
qed

3.2.4 Lifting PDG-BS with weak-control-dependence

lemma lift-StrongPostdomination:
assumes \( wf:CFGExit-wf \) sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
and \( spd:StrongPostdomination \) sourcenode targetnode kind valid-edge Entry Exit
and \( inner:CFGExit.inner-node \) sourcenode targetnode valid-edge Entry Exit nx
shows \( StrongPostdomination \) src trg knd (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry NewExit

proof –
interpret \( StrongPostdomination \) sourcenode targetnode kind valid-edge Entry Exit
by (rule spd)
have \( pd:Postdomination \) sourcenode targetnode kind valid-edge Entry Exit
by (unfold-locales)
interpret \( pd':Postdomination \) src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
NewEntry NewExit
by (fastforce intro:wf inner lift-Postdomination pd)
interpret \( CFGExit-wf:CFGExit-wf \) src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L state-val NewExit
by (fastforce intro:lift-CFGExit-wf wf)
from \( wf \) have \( CFG:CFG \) src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
by (rule lift-CFG)
show \(?thesis\)
proof
fix \( n \) assume \( CFG.valid-node \) src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) \( n \)
show finite
\( \{ n'. \exists a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' \land src a' = n \land trg a' = n' \} \)
proof (cases \( n \))
case NewEntry
hence \( \{ n'. \exists a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' \land src a' = n \land trg a' = n' \} = \{ NewExit,Node \} \)
by (auto elim:lift-valid-edge.cases intro:lift-valid-edge.intros)
thus \(?thesis\) by simp
next
case NewExit
hence \( \{ n'. \exists a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' \land src a' = n \land trg a' = n' \} = \{ \} \)
by fastforce
thus \(?thesis\) by simp
next
case \( \) (Node \( m \))
with \( Entry-Exit-edge \) \( \) (CFG.valid-node \) src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) \( n \)
have valid-node \( m \)
by (auto elim:lift-valid-edge.cases
simp:CFG.valid-node-def[OF CFG] valid-node-def)
lemma lift-PDG-wcd:
assumes PDG: PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
(StrongPostdomination.weak-control-dependence sourcenode targetnode valid-edge Exit)
and spd: StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
shows PDG src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
(StrongPostdomination.weak-control-dependence src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit)
proof –
interpret PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
StrongPostdomination.weak-control-dependence sourcenode targetnode

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 \{m'. \exists a'. lift-valid-edge valid-edge sourcenode targetnode kind
Entry Exit a' \land src a' = n \land trg a' = n'\} \subseteq
(\{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
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
qed
valid-edge Exit

by (rule PDG)
have \( wf : CFG \text{Exit-wf} \) sourcenode targetnode kind valid-edge Entry Def Use state-val Exit by (unfold-locales)
from \( wf \) spd inner have spd \( ': \text{StrongPostdomination} \) src try knd
(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 \( CFG : CFG \) src try knd
(lift-valid-edge valid-edge source node target node kind Entry Exit)
NewEntry NewExit
by (rule lift-CFG)
from \( wf \) have \( CFG \text{Exit} : CFG \text{Exit} \) src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry NewExit
by (rule lift-CFGExit)
from \( wf \) have \( CFG \text{Exit-wf} : CFG \text{Exit-wf} \) src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry NewExit-wf
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 = \{\} \land
lift-Use Use Entry Exit H L NewEntry = \{\}
by (rule CFG-wf.Entry-empty)
next
fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V \notin lift-Def Def Entry Exit H L (src a) and pred (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) and knd a' = (Q') and
∀ s. (Q s → ¬Q' s) ∧ (Q' s → ¬Q s)
by (rule CFG-wf, deterministic)

next
fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and src a = NewExit
with CFGExit show False by (rule CFGExit, Exit-source)

next
from CFGExit
show ∃ a. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a ∧
src a = NewEntry ∧ trg a = NewExit ∧ knd a = (λs. False)
by (rule CFGExit, Entry-Exit-edge)

next
from CFGExit-wf
show lift-Def Def Entry Exit H L NewExit = {} ∧
lift-Use Use Entry Exit H L NewExit = {}
by (rule CFGExit-wf, Exit-empty)

next
fix n n'
assume wcd: StrongPostdomination.weak-control-dependence src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit 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 \( \exists \text{as}. \ CFG \text{path src trg} \)

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

\( n \text{ as } n' \land as \neq [] \)

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

qed

lemma lift-PDG-weak-backward-slice:

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

defines lve:lve \( \equiv \) lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and lDef:lDef \( \equiv \) lift-Def Def Entry Exit H L
and lUse:lUse \( \equiv \) lift-Use Use Entry Exit H L

assumes PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
(PDG.weak-control-dependence sourcenode targetnode valid-edge Exit)
and spd:StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nz
and \( H \cap L = \{\} \) and \( H \cup L = \text{UNIV} \)

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

(PDG.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

(PDG.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

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

by (fastforce intro:lift-PDG-wcd)

from wf spd inner have spd':StrongPostdomination src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry NewExit

by (rule lift-StrongPostdomination)

from wf lve have CFG:CFG src trg lve NewEntry

by (fastforce intro:lift-CFG)

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

from \( wf \) lve have \( CFG\Exit\-\( CFG\Exit \) src trg knd lve NewEntry NewExit
by (fastforce intro: lift-\( CFG\Exit \))

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

show \( \) thesis
proof
fix \( n \) S
assume \( n \in PDG.PDG\-BS \) src trg lve lDef lUse
(StrongPostdomination.weak-control-dependence src trg lve NewExit) \( S \)
with \( PDG' \) show \( CFG.\) valid-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
(StrongPostdomination.weak-control-dependence src trg lve NewExit) \( S \)
by (fastforce intro: PDG.PDG-path-nil \[ OF PDG' \] simp: PDG.PDG-BS-def \[ OF PDG' \])

PDG'

next
fix \( n' \) S \( n \) V
assume \( n' \in PDG.PDG\-BS \) src trg lve lDef lUse
(StrongPostdomination.weak-control-dependence src trg lve NewExit) \( S \)
and \( CFG\-wf.\) data-dependence src trg lve lDef lUse \( n \) V \( n' \)
thus \( n \in PDG.PDG\-BS \) src trg lve lDef lUse
(StrongPostdomination.weak-control-dependence src trg lve NewExit) \( S \)
by (fastforce intro: PDG.PDG-path-append \[ OF PDG' \] PDG.PDG-path-ddep \[ OF PDG' \]
PDG'

split: if-split-asm)

next
fix \( n \) S
interpret \( PDGx: PDG \) src trg knd lve NewEntry lDef lUse state-val NewExit
StrongPostdomination \( weak\-control\-dependence \) src trg lve NewExit
by (rule PDG')

interpret \( spdx: StrongPostdomination \) src trg knd lve NewEntry NewExit
by (fastforce intro: spd' simp: lve)

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

from \( WeakControlDependencePDG.\) obs-singleton \[ OF wcd \]

have \( \exists m. \) 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 \( n \)
\((PDG.PDG\-BS src trg lve lDef lUse\)
\[(\text{StrongPostdomination.weak-control-dependence src trg lve NewExit}) \ S \) = \\
\{\} \\
\text{by (fastforce simp: WeakControlDependencePDG.PDG-BS-w-def[OF \ wcd])} \\
\text{thus finite (CFG.obs src trg lve n)} \\
(\text{PDG.PDG-BS src trg lve lDef lUse}) \\
(\text{(StrongPostdomination.weak-control-dependence src trg lve NewExit}) \ S \) \\
\text{by fastforce} \\
next \\
fix \ n \ S \\
\text{interpret PDGx:PDG src trg knd lve NewEntry lDef lUse state-val NewExit} \\
\text{StrongPostdomination.weak-control-dependence src trg lve NewExit} \\
\text{by (rule PDG')} \\
\text{interpret spdx:StrongPostdomination src trg knd lve NewEntry NewExit} \\
\text{by (fastforce intro:spd' simp:hee)} \\
\text{have \ wcd:WeakControlDependencePDG src trg knd lve NewEntry} \\
lDef lUse state-val NewExit \text{ by (unfold-locales)} \\
\text{from WeakControlDependencePDG.obs-singleton[OF \ wcd]} \\
\text{have (}\exists \ m. \CFG.obs src trg lve n) \\
(\text{PDG.PDG-BS src trg lve lDef lUse}) \\
(\text{(StrongPostdomination.weak-control-dependence src trg lve NewExit}) \ S \) = \\
\{m\} \lor \\
\text{CFG.obs src trg lve n} \\
(\text{PDG.PDG-BS src trg lve lDef lUse}) \\
(\text{(StrongPostdomination.weak-control-dependence src trg lve NewExit}) \ S \) = \\
\{\} \\
\text{by (fastforce simp: WeakControlDependencePDG.PDG-BS-w-def[OF \ wcd])} \\
\text{thus card (CFG.obs src trg lve n)} \\
(\text{PDG.PDG-BS src trg lve lDef lUse}) \\
(\text{(StrongPostdomination.weak-control-dependence src trg lve NewExit}) \ S \) \leq \\
I \\
\text{by fastforce} \\
next \\
fix \ a \ assume \ lve \ a \ and \ src \ a = \text{NewEntry} \\
with \ lve \ show \ try \ a = \text{NewExit} \lor \ try \ a = \text{Node Entry} \\
\text{by (fastforce elim:lift-valid-edge.cases)} \\
next \\
from \ lve-Entry-edge lve \\
\text{show } \exists \ a. \ lve \ a \land \ src \ a = \text{NewEntry} \land \ try \ a = \text{Node Entry} \land \ knd \ a = (\lambda s. \text{True}) \land \\
\text{by (fastforce simp:knd-def)} \\
next \\
fix \ a \ assume \ lve \ a \ and \ try \ a = \text{Node Entry} \\
with \ lve \ show \ src \ a = \text{NewEntry} \text{ by (fastforce elim:lift-valid-edge.cases)} \\
next \\
fix \ a \ assume \ lve \ a \ and \ try \ a = \text{NewExit} \\
with \ lve \ show \ src \ a = \text{NewEntry} \lor \ src \ a = \text{Node Exit} \\
\text{by (fastforce elim:lift-valid-edge.cases)} \\
next \\
from \ lve-Exit-edge lve
show \ exist a. lve a \and src a = Node Exit \and trg a = NewExit \and knd a = (\lambda s. 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 \cap L = \{\}; show H \cap L = \{\}.
next
  from :H \cup L = UNIV; show H \cup L = UNIV.
qed
qed

end

4 Information Flow for While

theory NonInterferenceWhile imports
  Slicing.SemanticsWellFormed
  Slicing.StaticControlDependences
  LiftingIntra
begin

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

4.1 Lifting labels-nodes and Defining final

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

lemmas WCFG-path-induct[consumes 1, case-names empty-path Cons-path] = CFG.path.induct[OF While-CFG-aux]
lemma lift-valid-node:
  assumes CFG.valid-node sourcenode targetnode (valid-edge prog) n
  shows CFG.valid-node src trg
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
  (Node n)
proof
  from (CFG.valid-node sourcenode targetnode (valid-edge prog) n)
  obtain a where valid-edge prog a and n = sourcenode a ∨ n = targetnode a
    by (fastforce simp:While-CFG.valid-node-def)
  from (n = sourcenode a ∨ n = targetnode a)
  show ?thesis
proof
  assume n = sourcenode a
  show ?thesis
proof (cases sourcenode a = Entry)
  case True
  have lift-valid-edge (valid-edge prog) sourcenode targetnode kind Entry Exit
    (NewEntry,(λs. True))_.Node Entry
    by (fastforce intro:lve-Entry-edge)
  with While-CFGExit-wf-aux[of prog] (n = sourcenode a) True show ?thesis
    by (fastforce simp:CFG.valid-node-def[OF lift-CFG1])
  next
  case False
  with (valid-edge prog a) (n = sourcenode a ∨ n = targetnode a)
  have lift-valid-edge (valid-edge prog) sourcenode targetnode kind Entry Exit
    (Node (sourcenode a),kind a,Node (targetnode a))
    by (fastforce intro:lve-edge)
  with While-CFGExit-wf-aux[of prog] (n = sourcenode a) show ?thesis
    by (fastforce simp:CFG.valid-node-def[OF lift-CFG1])
  qed
next
  assume n = targetnode a
  show ?thesis
proof (cases targetnode a = Exit)
  case True
  have lift-valid-edge (valid-edge prog) sourcenode targetnode kind Entry Exit
    (Node Exit,(λs. True))_.NewExit
    by (fastforce intro:lve-Exit-edge)
  with While-CFGExit-wf-aux[of prog] (n = targetnode a) True show ?thesis
    by (fastforce simp:CFG.valid-node-def[OF lift-CFG1])
  next
  case False
  with (valid-edge prog a) (n = sourcenode a ∨ n = targetnode a)
  have lift-valid-edge (valid-edge prog) sourcenode targetnode kind Entry Exit
    (Node (sourcenode a),kind a,Node (targetnode a))
    by (fastforce intro:lve-edge)
  with While-CFGExit-wf-aux[of prog] (n = targetnode a) show ?thesis
  qed
lemma lifted-CFG-fund-prop:
assumes labels-LDCFG-nodes prog n c and ⟨c, s⟩ →∗ ⟨c', s'⟩
shows ∃ n' as CFG.path src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
n as n' ∧ transfers (CFG.kinds knd as) s = s' ∧
preds (CFG.kinds knd as) s ∧ labels-LDCFG-nodes prog n' c'
proof
  from ⟨labels-LDCFG-nodes prog n c⟩ obtain nx where n = Node nx
  and labels-nodes prog nx c by (cases n) auto
  from ⟨labels-nodes prog nx c⟩ ⟨⟨c, s⟩ →∗ ⟨c', s'⟩⟩
  obtain n' as where prog ⊢ nx − as →∗ n' and transfers (CFG.kinds kind as) s = s'
  and preds (CFG.kinds kind as) s ∧ labels-nodes prog n
  proof (induct arbitrary: n s c rule: WCFG-path-induct)
  case (empty-path n nx)
    from ⟨CFG.valid-node sourcenode targetnode (valid-edge prog) n⟩
    have valid-node:CFG.valid-node src trg
      (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
      (Node nx) es ⟨Node n'⟩ ∧ transfers (CFG.kinds kind as) s = s' ∧
      preds (CFG.kinds kind as) s
    proof
      (induct arbitrary: n s c rule: WCFG-path-induct)
      case (empty-path n nx)
        from ⟨CFG.valid-node sourcenode targetnode (valid-edge prog) n⟩
        have valid-node:CFG.valid-node src trg
          (lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
          (Node nx)
        by (rule lift-valid-node)
        have CFG.kinds kind
          ([ ]::(w-node LDCFG-node × state edge-kind × w-node LDCFG-node) list) =
          by (simp add: CFG.kinds-def[OF lift-CFG[OF While-CFGExit-wf-aux]])
        with ⟨transfers (CFG.kinds kind []) s = s'⟩ ∧ preds (CFG.kinds kind []) s
        valid-node
        show ?case
          by (fastforce intro: CFG.empty-path[OF lift-CFG[OF While-CFGExit-wf-aux]])
        simp: While-CFG.kinds-def)
    next
    case (Cons-path n' as n' a nx)
\[\text{note IH} = \{ \neg n \in s \text{ c. } \exists \text{ transfers (CFG.kinds kind as) s = s'} \}
\]
\[\text{preds (CFG.kinds kind as) s; n = LDCFG-node.Node n''} \]
\[\text{labels-nodes prog n'' c; labels-nodes prog n' c'} \]
\[\implies \exists \text{ es. CFG.path src trg} \]
\[(\text{lift-valid-edge (valid-edge prog) source-node target-node kind (Entry) (Exit)})(LDCFG-node.Node n'') \wedge \text{transfers (CFG.kinds kind as) s = s'} \land \text{preds (CFG.kinds kind as) s} \]
\[\text{from (transfers (CFG.kinds kind (a \not= as)) s = s')} \]
\[\text{have transfers (CFG.kinds kind as) (transfer (kind a) s) = s'} \]
\[\text{by(simp add:While-CFG.kinds-def)} \]
\[\text{from (preds (CFG.kinds kind (a \not= as)) s) \}
\[\text{have preds (CFG.kinds kind as) (transfer (kind a) s) \}
\[\text{and pred (kind a) s by(simp-all add:While-CFG.kinds-def)} \]
\[\text{show \(\Box\) case \}
\[\text{proof(cases source-node a = (Entry)) \]
\[\text{case True} \]
\[\text{with (source-node a = nx) (labels-nodes prog nx c) have False by simp} \]
\[\text{thus \(\Box\) thesis by simp} \]
\[\text{next} \]
\[\text{case False} \]
\[\text{with (valid-edge prog a)} \]
\[\text{have edge:lift-valid-edge (valid-edge prog) source-node target-node kind} \]
\[\text{Entry Exit (Node (source-node a),kind a,Node (target-node a))} \]
\[\text{by(fastforce intro:live-edge)} \]
\[\text{from (prog \(\vdash\) n'' as \(\rightarrow^{*}\) n') \}
\[\text{have CFG.valid-node source-node target-node (valid-edge prog) n''} \]
\[\text{by(rule While-CFG.path-valid-node)} \]
\[\text{then obtain c'' where labels-nodes prog n'' c'')} \]
\[\text{proof(cases rule:While-CFGExit.valid-node-cases) \]
\[\text{case Entry} \]
\[\text{with (target-node a = n''; valid-edge prog a) have False by fastforce} \]
\[\text{thus \(\Box\) thesis by simp} \]
\[\text{next} \]
\[\text{case Exit} \]
\[\text{with (prog \(\vdash\) n'' as \(\rightarrow^{*}\) n') have n' = (Entry)} \text{ by fastforce} \]
\[\text{with (labels-nodes prog n' c') have False by fastforce} \]
\[\text{thus \(\Box\) thesis by simp} \]
\[\text{next} \]
\[\text{case inner} \]
\[\text{then obtain l'' where \(\Box\) simp:n'' = (- l'' -) by(cases n'')} \text{ auto} \]
\[\text{with (valid-edge prog a) (target-node a = n')} \text{ have l'' \< \#:prog} \]
\[\text{by(fastforce intro:WCFG-target-label-less-num-nodes simp:valid-edge-def)} \]
\[\text{then obtain c'' where labels prog l'' c'')} \]
\[\text{by(fastforce dest:less-num-inner-nodes-label)} \]
\[\text{with that show \(\Box\) thesis by fastforce} \]
\[\text{qed} \]
\[\text{from IH[OF \(\Box\) transfers (CFG.kinds kind as) (transfer (kind a) s) = s'} \]
\[\text{\(\Box\) preds (CFG.kinds kind as) (transfer (kind a) s) - this} \]
\[\text{\(\Box\) labels-nodes prog n' c')]} \]
obtain es where CFG.path src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-)) (LDCFG-node.Node n′′) es (LDCFG-node.Node n′)
and transfers (CFG.kinds knd es) (transfer (kind a) s) = s'
and preds (CFG.kinds knd es) (transfer (kind a) s) by blast
with ⟨targetnode a = n′′⟩ ⟨sourcenode a = nx⟩ edge
have path:CFG.path src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode
kind (-Entry-) (-Exit-))
(LDCFG-node.Node n′′) ((Node (sourcenode a),kind a,Node (targetnode a)))#es)
(LDCFG-node.Node nx) ((Node (sourcenode a),kind a,Node (targetnode a)))
by (fastforce intro:CFG.Cons-path[OF lift-CFG[OF While-CFGExit-wf-aux]])
from edge have knd (Node (sourcenode a),kind a,Node (targetnode a)) = kind a
by (simp add:knd-def)
with ⟨transfers (CFG.kinds knd es) (transfer (kind a) s) = s′⟩
⟨preds (CFG.kinds knd es) (transfer (kind a) s)⟩ ⟨pred (kind a) s⟩
have transfers
(CFG.kinds knd ((Node (sourcenode a),kind a,Node (targetnode a)))#es)) s
= s'
and preds
(CFG.kinds knd ((Node (sourcenode a),kind a,Node (targetnode a)))#es)) s
by (auto simp:CFG.kinds-def[OF lift-CFG[OF While-CFGExit-wf-aux]])
with path show thesis by blast
qed
qed
with ⟨n = Node nx⟩ ⟨labels-LDCFG-nodes prog (Node n′) c⟩
show thesis by fastforce
qed

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

lemma final-edge:
labels-nodes prog n Skip ⟹ prog ⊢ n ⊢ id → (-Exit-)
proof (induct prog arbitrary:n)
case Skip
from ⟨labels-nodes Skip n Skip⟩ have n = ⟨0⟩
by (cases n)(auto elim:labels.cases)
thus ⟨case ⟨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 (\text{Seq } c_1 c_2)

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

from \( \langle \text{labels-nodes } (c_1; c_2) \text{ n Skip}\) obtain \( l \) where \( n = (l - l -) \)
and \( l \geq #:c_1 \) and \( \text{labels-nodes } c_2 \text{ (} l - #:c_1 - \text{) Skip} \)
by (cases \( n \))(auto elim:labels.cases)

from IH2[\( \langle \text{OF } \text{labels-nodes } c_2 \text{ (} l - #:c_1 - \text{) Skip}\) ] have \( c_2 \vdash (l - #:c_1 -) \downarrow \text{id} \rightarrow (\text{Exit-}c) . \)

with \( l \geq #:c_1 \) have \( c_1; c_2 \vdash (l - #:c_1 -) \odot #:c_1 - \uparrow \text{id} \rightarrow (\text{Exit-}) \odot #:c_1 
\)
by (fastforce intro: WCFG-SeqSecond)

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

next
case (\text{Cond } b c_1 c_2)

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

note IH2 = (\forall n. \text{labels-nodes } c_1 \text{ n Skip } \implies c_2 \vdash n \downarrow \text{id} \rightarrow (\text{Exit-}c) )

from \( \langle \text{labels-nodes } (b \text{ c_1 else } c_2) \text{ n Skip}\) obtain \( l \) where \( n = (l - l -) \) and disj:\( (l \geq 1 \wedge \text{labels-nodes } c_1 \text{ (} l - 1 - \text{) Skip} ) \vee \\
(l \geq #:c_1 + 1 \wedge \text{labels-nodes } c_2 \text{ (} l - #:c_1 - 1 - \text{) Skip} ) \)
by (cases \( n \))(fastforce elim:labels.cases)+

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

proof
assumee \( 1 \leq l \wedge \text{labels-nodes } c_1 \text{ (} l - l - \text{) Skip} \)

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

from IH1[\( \langle \text{OF } \text{labels-nodes } c_1 \text{ (} l - l - \text{) Skip}\) ] have \( c_1 \vdash (l - l -) \downarrow \text{id} \rightarrow (\text{Exit-}c) . \)

with \( l \leq l \) have \( \text{if (} b \text{ c_1 else } c_2 \vdash (l - l -) \odot 1 - \uparrow \text{id} \rightarrow (\text{Exit-}) \odot 1 
\)
by (fastforce intro: WCFG-CondThen)

with \( n = (l - l -) \) show \( \leq l \) case by (simp add:id-def)

next
assumee \( #:c_1 + 1 \leq l \wedge \text{labels-nodes } c_2 \text{ (} l - #:c_1 - 1 - \text{) Skip} \)

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

from IH2[\( \langle \text{OF } \text{labels-nodes } c_2 \text{ (} l - #:c_1 - 1 - \text{) Skip}\) ] have \( c_2 \vdash (l - #:c_1 - 1 -) \downarrow \text{id} \rightarrow (\text{Exit-}c) . \)

with \( #:c_1 + 1 \leq l \) have \( \text{if (} b \text{ c_1 else } c_2 \vdash (l - #:c_1 - 1 -) \odot ( #:c_1 + 1 ) 
\)
\- \uparrow \text{id} \rightarrow (\text{Exit-}) \odot ( #:c_1 + 1 ) 
\)
by (fastforce intro: WCFG-CondElse)

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

next
case (\text{While } b c)

from \( \langle \text{labels-nodes } (\text{while (} b \text{ c) n Skip}\) have \( n = (l - l -) 
\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 =
\[\text{lift-wod-backward-slice}[\text{OF While-CFGExit-wf-aux HighLowDistinct HighLowUnIV}]\]

**Lemma** \text{WODNonInterference}:

NonInterferenceIntra src trg knd
\[\text{lift-valid-edge (valid-edge prog) source node target node kind}\]
\[\text{(-Entry-) (-Exit-)}\]
NewEntry (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
\[(\text{lift-Use (Uses prog) (-Entry-) (-Exit-) H L}) id\]
\[\text{CFG-wf, wod-backward-slice src trg}\]
\[(\text{lift-valid-edge (valid-edge prog) source node target node kind}\]
\[\text{(-Entry-) (-Exit-)}\]
\[\text{lift-Def (Defs prog) (-Entry-) (-Exit-) H L}\]
\[\text{lift-Use (Uses prog) (-Entry-) (-Exit-) H L}\]
\[\text{red labels-LDCFG-nodes prog}\]
NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final

**Proof** –

\text{interpret} NonInterferenceIntraGraph src trg knd
\[\text{lift-valid-edge (valid-edge prog) source node target node kind}\]
\[\text{(-Entry-) (-Exit-)}\]
NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
\[\text{lift-Use (Uses prog) (-Entry-) (-Exit-) H L id}\]
\[\text{CFG-wf, wod-backward-slice src trg}\]
\[(\text{lift-valid-edge (valid-edge prog) source node target node kind}\]
\[\text{(-Entry-) (-Exit-)}\]
\[\text{lift-Def (Defs prog) (-Entry-) (-Exit-) H L}\]
\[\text{lift-Use (Uses prog) (-Entry-) (-Exit-) H L}\]
\[\text{NewExit H L LDCFG-node.Node (-Entry-)} (LDCFG-node.Node (-Exit-))\]
by(rule WODNonInterferenceGraph)

\text{interpret} BackwardSlice-wf src trg knd
\[\text{lift-valid-edge (valid-edge prog) source node target node kind}\]
\[\text{(-Entry-) (-Exit-)}\]
NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
\[\text{lift-Use (Uses prog) (-Entry-) (-Exit-) H L id}\]
\[\text{CFG-wf, wod-backward-slice src trg}\]
\[(\text{lift-valid-edge (valid-edge prog) source node target node kind}\]
\[\text{(-Entry-) (-Exit-)}\]
\[\text{lift-Def (Defs prog) (-Entry-) (-Exit-) H L}\]
\[\text{lift-Use (Uses prog) (-Entry-) (-Exit-) H L}\]
\[\text{red labels-LDCFG-nodes prog}\]
NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-))

**Proof** (unfold-locales)

\text{fix} n c s c' s'
\text{assume} labels-LDCFG-nodes prog n c \text{ and } \langle c, s \rangle \leftrightarrow \langle c', s' \rangle
\text{thus } \exists n' \text{ as } CFG.path src trg
\[\text{lift-valid-edge (valid-edge prog) source node target node kind} (-Entry-) (-Exit-)\]
\[n \text{ as } n' \wedge \text{transfers (CFG.kinds knd as) } s = s' \wedge\]
\[\text{preds (CFG.kinds knd as) } s \wedge \text{labels-LDCFG-nodes prog n' c'}\]
by(rule lifted-CFG-fund-prop)

qed
show ?thesis
proof (unfold-locales)
  fix c n
  assume final c and labels-LDCFG-nodes prog n c
  from final c have [simp]:c = Skip by (cases c) auto
  from labels-LDCFG-nodes prog n c obtain nx where [simp]:n = Node nx
  and labels-nodes prog nx Skip by (cases n) auto
  from labels-nodes prog nx Skip have prog |- nx —⇑ id → (_Exit_)
    by (rule final-edge)
  then obtain a where valid-edge prog a and sourcenode a = nx
    and kind a = ▲ id and targetnode a = (_Exit_)
    by (auto simp: valid-edge-def)
  with labels-nodes prog nx Skip
  show ∃ a. lift-valid-edge (valid-edge prog) sourcenode targetnode
       kind (_Entry_) (_Exit_) a ∧
    src a = n ∧ trg a = LDCFG-node.Node (_Exit_) ∧ knd a = ▲ id
    by (rule-tac x = (Node nx, ▲ id, Node (_Exit_)) in exI)
     (auto intro!: lve-edge simp: knd-def valid-edge-def)
  qed
  qed

4.3 Semantic Non-Interference for Standard Control Dependence

lemma inner-node-exists: ∃ n. CFGExit.inner-node sourcenode targetnode
  (valid-edge prog) (_Entry_) (_Exit_) n
proof –
  have prog |- (_Entry_) —(λs. 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)
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-fund-prop)
qed
show ?thesis
proof (unfold-locales)
fix c n
assume final c and labels-LDCFG-nodes prog n c
from final c have [simp]:c = Skip by (cases c) auto
from labels-LDCFG-nodes prog n c obtain nx where [simp]:n = Node nx
and labels-nodes prog nx Skip by (cases n) auto
from labels-nodes prog nx Skip have prog ⊢ nx −⇑ id → (-Exit-)
by (rule final-edge)
then obtain a where valid-edge prog a and sourcenode a = nx
and kind a = ↑ id and targetnode a = (-Exit-)
by (auto simp: valid-edge-def)
with (labels-nodes prog nx Skip)
show ∃ a. lift-valid-edge (valid-edge prog) sourcenode targetnode
kind (-Entry-) (-Exit-) a ∧
src a = n ∧ trg a = LDCFG-node. Node (-Exit-) ∧ knd a = ↑ id
by (rule-tac x = (Node nx, ↑ id, Node (-Exit-)) in exI)
(auto intro! : lve-edge simp: knd-def valid-edge-def)
qed
qed

4.4 Semantic Non-Interference for Weak Control Dependence

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

lemma WCDNonInterference:
NonInterferenceIntra src trg knd
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
  (-Entry-) (-Exit-))
NewEntry (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
(lift-Use (Uses prog) (-Entry-) (-Exit-) H L) id
(PDG.PDG-BS src trg
  (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
    (-Entry-) (-Exit-))
  (lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
  (lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
  (StrongPostdomination.weak-control-dependence src trg
    (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
      (-Entry-) (-Exit-) NewExit))
  reds (labels-LDCFG-nodes prog)
NewExit H L (LDCFG-node. Node (-Exit-)) (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)
(StrongPostdomination.weak-control-dependence src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-) NewExit)
NewExit H L LDCFG-node.
Node (-Entry-) LDCFG-node.
Node (-Exit-)

by (fastforce intro:WCDNonInterferenceGraph)

interpret BackwardSlice-wf src trg knd
lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-)
NewEntry lift-Def (Defs prog) (-Entry-) (-Exit-) H L
lift-Use (Uses prog) (-Entry-) (-Exit-) H L id

PDG_PDG-BS src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-))
(lift-Def (Defs prog) (-Entry-) (-Exit-) H L)
(lift-Use (Uses prog) (-Entry-) (-Exit-) H L)
(StrongPostdomination.weak-control-dependence src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind
(-Entry-) (-Exit-) NewExit) reds labels-LDCFG-nodes prog

proof (unfold-locales)
fix n c s c' s'
assume labels-LDCFG-nodes prog n c and ⟨c,s⟩ →∗ ⟨c',s'⟩
thus ∃ n' as. CFG.path src trg
(lift-valid-edge (valid-edge prog) sourcenode targetnode kind (-Entry-) (-Exit-))
n as n' ∧ transfers (CFG.kinds knd as) s = s' ∧
preds (CFG.kinds knd as) s ∧ labels-LDCFG-nodes prog n' c'
by (rule lifted-CFG-fund-prop)

qed
show ?thesis
proof (unfold-locales)
fix n c
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 ∧
\[
\text{src} \ a = n \land \ \text{trg} \ a = \text{LDCFG-node.} \text{Node} \ (-\text{Exit}) \land \ \text{knd} \ a = \uparrow \text{id}
\]

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

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

qed

qed

end

end

References


