Slicing Guarantees Information Flow
Noninterference

Daniel Wasserrab
October 11, 2017

Abstract
In this contribution, we show how correctness proofs for intra-\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 determines if a given program leaks secret information to public entities. The major group are so called IFC type systems, where well-typed means that the respective program is secure. Several IFC type systems have been verified in proof assistants, e.g. see \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 HRB Slicing guarantees IFC Noninterference

theory NonInterferenceInter
imports HRB-Slicing,FundamentalProperty
begin

1
2.1 Assumptions of this Approach

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

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

locale NonInterferenceInterGraph =
SDG sourcenode targetnode kind valid-edge Entry
get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
for sourcenode \::\ edge \Rightarrow \node and targetnode \::\ edge \Rightarrow \node
and kind \::\ edge \Rightarrow \(\lambda\ var,\ val,\ ret,\ pname\) edge-kind
and valid-edge \::\ edge \Rightarrow bool
and Entry \::\ node \(('\text{-Entry}'\)) and get-proc \::\ node \Rightarrow \node
and get-return-edges \::\ edge \Rightarrow \node set
and procs \::\ \node \Rightarrow \node set list and Main \::\ \node
and Exit::\node \('\text{-Exit}'\))
and Def \::\ node \Rightarrow \node set and Use ::\node \Rightarrow \node set
and ParamDefs ::\ node \Rightarrow \node set and ParamUses ::\ node \Rightarrow \node set list +
fixes \(H\) ::\var set
fixes \(L\) ::\var set
fixes \(High\) ::\node \('\text{-High}''\))
fixes \(Low\) ::\node \('\text{-Low}''\))
assumes Entry-edge-Exit-or-High:
\[\text{valid-edge}\ a;\ \text{sourcenode}\ a =\ (-\text{Entry}-)\] \[\Rightarrow\ \text{targetnode}\ a =\ (-\text{Exit})\lor\ \text{targetnode}\ a =\ (-\text{High})\]
and High-target-Entry-edge:
\[\exists\ a.\ \text{valid-edge}\ a\land\ \text{sourcenode}\ a =\ (-\text{Entry})\land\ \text{targetnode}\ a =\ (-\text{High})\land\ \text{kind}\ a =\ (\lambda s.\ \text{True})\]
and Entry-predeccessor-of-High:
\[\text{valid-edge}\ a;\ \text{targetnode}\ a =\ (-\text{High})\] \[\Rightarrow\ \text{sourcenode}\ a =\ (-\text{Entry})\]
and Exit-edge-Entry-or-Low:
\[\text{valid-edge}\ a;\ \text{targetnode}\ a =\ (-\text{Exit})\] \[\Rightarrow\ \text{sourcenode}\ a =\ (-\text{Entry})\lor\ \text{sourcenode}\ a =\ (-\text{Low})\]
and Low-source-Exit-edge:
\[\exists\ a.\ \text{valid-edge}\ a\land\ \text{sourcenode}\ a =\ (-\text{Low})\land\ \text{targetnode}\ a =\ (-\text{Exit})\land\ \text{kind}\ a =\ (\lambda s.\ \text{True})\]
and Exit-successor-of-Low:
\[\text{valid-edge}\ a;\ \text{sourcenode}\ a =\ (-\text{Low})\] \[\Rightarrow\ \text{targetnode}\ a =\ (-\text{Exit})\]
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 \not= \{\} shows (-Low-) \not= (-Exit-)
proof
  assume (-Low-) = (-Exit-)
  have Use (-Exit-) = \{\} by fastforce
  with UseLow \langle L \not= \{\} \rangle \langle (-Low-) = (-Exit-) \rangle show False by simp
qed

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

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

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

lemma get-proc-High:
  get-proc (-High-) = Main
proof –
  from High-target-Entry-edge obtain a where valid-edge a
  and sourcenode a = (-Entry-) and targetnode a = (-High-) and intra-kind (kind a) by (fastforce simp:intra-kind-def)
  from \langle valid-edge a \rangle \langle intra-kind (kind a) \rangle
  have get-proc (sourcenode a) = get-proc (targetnode a) by (rule get-proc-intra)
  with \langle sourcenode a = (-Entry-) \rangle \langle targetnode a = (-High-) \rangle get-proc-Entry
  show \?thesis by simp
qed
lemma Entry-path-High-path:
assumes \((\text{-Entry})\) \(\rightarrow as \rightarrow n\) and inner-node \(n\)
obtains \(a'\) \(\text{as'}\) where \(as = a' \# \text{as'}\) and \((\text{-High})\) \(\rightarrow as' \rightarrow n\)
and kind \(a' = (\lambda s. \text{True})\) \(\checkmark\)
proof (atomize-elim)
from \((\text{-Entry})\) \(\rightarrow as \rightarrow n\) (inner-node \(n\))
show \(\exists a'\) \(\text{as'}\). \(as = a' \# \text{as'}\) \& \((\text{-High})\) \(\rightarrow as' \rightarrow n\) \& kind \(a' = (\lambda s. \text{True})\) \(\checkmark\)
proof (induct \(\equiv (\text{-Entry})\) as \(n\) rule: path-induct)
case (Cons-path \(n''\) as \(n'\) \(a\))
from \((n'' \rightarrow as \rightarrow n'b\) (inner-node \(n'b\)) \(n'' \neq (\text{-Exit})\)
by (fastforce simp: inner-node-def)
with (valid-edge \(a\)) (sourcenode \(a\) = (\text{-Entry})\) (targetnode \(a\) = \(n''\))
have \(n'' = (\text{-High})\) by (drule Entry-edge-Exit-or-High, auto)
from High-target-Entry-edge
obtain \(a'\) where valid-edge \(a'\) and sourcenode \(a'\) = (\text{-Entry})
and targetnode \(a'\) = (\text{-High}) and kind \(a' = (\lambda s. \text{True})\) \(\checkmark\)
by blast
with (valid-edge \(a\)) (sourcenode \(a\) = (\text{-Entry})\) (targetnode \(a\) = \(n''\))
\((n'' = (\text{-High})\)
have \(a = a'\) by (auto dest: edge-det)
with \((n'' \rightarrow as \rightarrow n'b\) \(n'' = (\text{-High})\) \(\equiv\) kind \(a' = (\lambda s. \text{True})\)
show \(\equiv\) case by blast
qed fastforce
qed

lemma Exit-path-Low-path:
assumes \(n \rightarrow as \rightarrow (\text{-Exit})\) and inner-node \(n\)
obtains \(a'\) \(\text{as'}\) where \(as = as' \# [a']\) and \(n \rightarrow as' \rightarrow (\text{-Low})\)
and kind \(a' = (\lambda s. \text{True})\) \(\checkmark\)
proof (atomize-elim)
from \((n \rightarrow as \rightarrow (\text{-Exit}))\)
show \(\exists as' \text{a'}. \ as = as' \# [a'] \& n \rightarrow as' \rightarrow (\text{-Low}) \& kind \(a' = (\lambda s. \text{True})\) \(\checkmark\)
proof (induct as rule: rev-induct)
case Nil
with \(\text{inner-node} \ n\) show \(\equiv\) case by fastforce
next
case \((\text{snc} \ a' \ \text{as'}\)\)
from \((n \rightarrow as' [a'] \rightarrow (\text{-Exit}))\)
have \(n \rightarrow as' \rightarrow \text{sourcenode} \ a' \ and \ valid-edge \ a' \ and \ targetnode \ a' = (\text{-Exit})\)
by (auto elim: path-split-snoc)
\{ assume sourcenode \(a' = (\text{-Entry})\)
with \((n \rightarrow as' \rightarrow \text{sourcenode} \ a'\) have \(n = (\text{-Entry})\)
by (blast intro!: path-Entry-target)
with \(\text{inner-node} \ n\) have False by (simp add: inner-node-def) \}
with (valid-edge \(a'\) \(\text{targetnode} a' = (\text{-Exit})\) \(\equiv\) sourcenode \(a' = (\text{-Low})\)
by (blast dest!: Edge-edge-Entry-or-Low)
from Low-source-Exit-edge
obtain ax where valid-edge \(ax\) and sourcenode \(ax = (\text{-Low})\)
and targetnode ax = (\text{-Exit-}) and kind ax = (\lambda s. \text{True}) \\
by blast \\
with (valid-edge a') (targetnode a' = (\text{-Exit-}) (sourcenode a' = (\text{-Low-})) \\
have a' = ax by (fastforce intro:edge-det) \\
with (n as'\rightarrow \ast sourcenode a') (sourcenode a' = (\text{-Low-}) (kind ax = (\lambda s. \text{True})) \\
show \text{?case by blast} \\
qed \\
qed \\

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

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

2.2 Low Equivalence \\
In classical noninterference, an external observer can only see public values, in our case the L-variables. If two states agree in the values of all L-variables, these states are indistinguishable for him. Low equivalence groups those states in an equivalence class using the relation $\approx_L$: \\
definition lowEquivalence :: ('var => 'val) list => ('var => 'val) list => bool 
(infixl $\approx_L$ 50) 
where s $\approx_L$ s' $\equiv \forall V \in L. hd s V = hd s' V$

The following lemmas connect low equivalent states with relevant variables as necessary in the correctness proof for slicing. \\
lemma relevant-vars-Entry: 
assumes V \in rv S (CFG-node (\text{-Entry-})) and (\text{-High-}) $\notin [HRB-slice S]_{CFG}$ 
shows V \in L 
proof 
from (V \in rv S (CFG-node (\text{-Entry-}))): obtain as n' 
where (\text{-Entry-}) \rightarrow_1 \ast parent-node n' 
and n' \in HRB-slice S and V \in UseSDG n' 
and $\forall n'', valid-SDG-node n'' \land parent-node n'' \in set (sourcenodes as)$ 
\rightarrow V \notin DefSDG n'' by (fastforce elim:rvE') 
from (\text{-Entry-}) \rightarrow_1 \ast parent-node n' \have valid-node (parent-node n') 
by (fastforce intro:path-valid-node simp:intra-path-def) 
thus \text{?thesis} 
proof(cases parent-node n' rule:valid-node-cases) 
\text{case Entry} 
with (V \in UseSDG n') \have False 
by \text{-(drule SDG-Use-parent-Use, simp add:Entry-empty)} 
thus \text{?thesis by simp}
next
  case Exit
  with \langle \exists V \in \text{Use}_{SDG} n' \rangle \text{ have } \text{False}
  by \neg \text{(drule SDG-Use-parent-Use,simp add:Exit-empty)}
  thus ?thesis by simp
next
  case inner
  with \langle \text{(-Entry-)} as' \rightarrow \top \text{ parent-node } n' \rangle \text{ obtain } a' as' \text{ where } as = a' \# as'
  and \langle \text{(-High-)} as' \rightarrow \top \text{ parent-node } n' \rangle
  by (fastforce elim:Entry-path-High-path simp:intra-path-def)
  from \langle \text{(-Entry-)} as' \rightarrow \top \text{ parent-node } n' \rangle \langle as = a' \# as' \rangle
  have sourcenode a' = \langle \text{(-Entry-)} \rangle
  proof (cases as' = [])
  case True
  with \langle \text{(-High-)} as' \rightarrow \top \text{ parent-node } n' \rangle \text{ have } parent-node n' = \langle \text{(-High-)} \rangle
  by (fastforce simp:intra-path-def)
  with \langle n' \in \text{HRB-slice } S \rangle \langle \text{(-High-)} \notin \lfloor \text{HRB-slice } S \rfloor_{CFG} \rangle
  have False
  thus ?thesis by simp
next
  case False
  with \langle \text{(-High-)} as' \rightarrow \top \text{ parent-node } n' \rangle \text{ have } hd \langle \text{sourcenodes } as' \rangle = \langle \text{(-High-)} \rangle
  by (fastforce intro:path-source-node simp:intra-path-def)
  from False have hd \langle \text{sourcenodes } as' \rangle \in \text{set } \langle \text{sourcenodes } as' \rangle
  by (fastforce intro:hd-in-set simp:sourcenodes-def)
  with \langle as = a' \# as' \rangle \text{ have } hd \langle \text{sourcenodes } as' \rangle \in \text{set } \langle \text{sourcenodes } as' \rangle
  by (simp add:sourcenodes-def)
  from \langle hd \langle \text{sourcenodes } as' \rangle = \langle \text{(-High-)} \rangle \rangle
  have valid-node \langle hd \langle \text{sourcenodes } as' \rangle \rangle \text{ by simp}
  have valid-SDG-node \langle CFG-node \langle \text{(-High-)} \rangle \rangle \text{ by simp}
  with \langle hd \langle \text{sourcenodes } as' \rangle = \langle \text{(-High-)} \rangle \rangle
  \langle hd \langle \text{sourcenodes } as' \rangle \in \text{set } \langle \text{sourcenodes } as' \rangle \rangle
  \forall n'', \text{ valid-SDG-node } n'' \land \text{ parent-node } n'' \in \text{set } \langle \text{sourcenodes } as' \rangle
  \to V \notin \text{Def}_{SDG} n''
  have V \notin \text{Def } \langle \text{(-High-)} \rangle
  by (fastforce dest:CFG-Def-SDG-Def[\text{OF } \text{valid-node } \langle hd \langle \text{sourcenodes } as' \rangle \rangle])
  hence V \notin \text{H by simp add:DefHigh}
  thus ?thesis by (rule not-High-Low)
qed
qed

lemma lowEquivalence-relevant-nodes-Entry:
assumes s \approx_L s' \land \langle \text{(-High-)} \rangle \notin \lfloor \text{HRB-slice } S \rfloor_{CFG}
shows $\forall V \in \text{rv} S \ (\text{CFG-node (-Entry-)}). \ \text{hd} s \ V = \text{hd} s' \ V$

proof

fix $V$ assume $V \in \text{rv} S \ (\text{CFG-node (-Entry-})$

with $\langle \text{HRB-slice} S \rangle_{\text{CFG}}$ have $V \in L$ by -(rule relevant-vars-Entry)

with $(s \approx_L s')$ show $\text{hd} s \ V = \text{hd} s' \ V$ by(simp add:lowEquivalence-def)

qed

2.3 The Correctness Proofs

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

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

lemma slpa-rev-Low-Use-Low:

assumes $\text{CFG-node (-Low-) \in S}$

shows $\{\text{same-level-path-aux} \ cs \ as; \ \text{upd-cs} cs \ as = []; \ \text{same-level-path-aux} \ cs \ as'$;

$\forall c \in \text{set cs}. \ \text{valid-edge} c; \ m \text{-as}\rightarrow\text{as'} (-\text{Low-}); \ m \text{-as'}\rightarrow\text{as} (-\text{Low-});$

$\forall i < \text{length} cs. \ \forall V \in \text{rv} S \ (\text{CFG-node (sourcenode (cs[i]))}).$

$\text{fst} (s!S\text{uc} i) \ V = \text{fst} (s^n\text{Suc} i) \ V; \ \forall i < \text{Suc} (\text{length} cs). \ \text{snd} (s!i) = \text{snd} (s^n!i);$\n
$\forall V \in \text{rv} S \ (\text{CFG-node} m). \ \text{state-val} s \ V = \text{state-val} s' \ V;$

$\text{preds} \ (\text{slice-kinds} S \ as) \ s; \ \text{preds} \ (\text{slice-kinds} S \ as') \ s';$

$\text{length} s = \text{Suc} (\text{length} cs); \ \text{length} s' = \text{Suc} (\text{length} cs)]$

$\implies \forall V \in \text{Use} \ (-\text{Low-}). \ \text{state-val} \ (\text{transfers} \ (\text{slice-kinds} S \ as) \ s) \ V =$

$\text{state-val} \ (\text{transfers} \ (\text{slice-kinds} S \ as') \ s') \ V$

proof(induct arbitrary;$m \text{ as'} \ s' \text{ rule:slpa-induct})

case (slpa-empty cs)

from $(m \text{-[]}\rightarrow\text{(-Low-)})$ have $m = (-\text{Low-})$ by fastforce

from $(m \text{-[]}\rightarrow\text{(-Low-)})$ have valid-node $m$

by(rule path-valid-node)+

{ fix $V$ assume $V \in \text{Use} \ (-\text{Low-})$

moreover

from $(\text{valid-node} m) \ (m = (-\text{Low-})$ have $(-\text{Low-}) \text{-[]}\rightarrow\text{(-Low-)}$

by(fastforce intro:empty-path simp:intra-path-def)

moreover

from $(\text{valid-node} m) \ (m = (-\text{Low-})$ have $CFG-node (-\text{Low-}) \in S$

by(fastforce intro:HRB-slice-refl)

ultimately have $V \in \text{rv} S \ (CFG-node m)$

using $(m = (-\text{Low-})$)

by(auto intro!:rdl CFG-Use-SDG-Use simp:sourcenodes-def) }

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

show $\text{thesis}$

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

case True with UseLow show $\text{thesis}$ by simp

next
\begin{verbatim}
case False
from (m − m#) →* (Low-); \langle m = (Low-) \rangle have as' = []
proof (induct m as' m'≡(Low-) rule:path.induct)
case (Cons-path m'' as' a m)
  from (valid-edge a) (sourcenode a = m) \langle m = (Low-) \rangle
  have targetnode a = (Exit-) by (rule Exit-successor-of-Low,simp+)
  with (targetnode a = m'→* m''−→* (Low-))
  have (Low-) = (Exit-) by (drule path-Exit-source,auto)
  with False have False by (drule Low-neq-Exit,simp)
  thus ?case by simp
qed simp
with \forall V ∈ Use (Low-). V ∈ rv S (CFG-node m)
\forall V ∈ rv S (CFG-node m). state-val s V = state-val s' V \Nil
show ?thesis by (auto simp:slice-kinds-def)
qed
next
case (slpa-intra cs a as)
note IH = (\forall m as' s'). [upd-cs cs as \Nil]; same-level-path-aux cs as'
\forall a \in set cs. valid-edge a; m − as→* (Low-); m − as'→* (Low-);
\forall i < length cs. \forall V ∈ rv S (CFG-node (sourcenode (cs i))).
fst (s ! Suc i) V = fst (s' ! Suc i) V;
\forall i < Suc (length cs). snd (s ! i) = snd (s' ! i);
\forall V ∈ rv S (CFG-node m). state-val s V = state-val s' V;
preds (slice-kinds S as) s; preds (slice-kinds S as') s';
length s = Suc (length cs); length s' = Suc (length cs)
\implies \forall V ∈ Use (Low-). state-val (transfers(slice-kinds S as) s) V =
state-val (transfers(slice-kinds S as') s') V
note rhs = (\forall i < length cs. \forall V ∈ rv S (CFG-node (sourcenode (cs i))).
fst (s ! Suc i) V = fst (s' ! Suc i) V)
from (m − a #) as→* (Low-) have sourcenode a = m and valid-edge a
and targetnode a = as→* (Low-) by (auto elim:path-split-Cons)
show ?case
proof (cases L = \Nil)
case True with UseLow show ?thesis by simp
next
case False
show ?thesis
proof (cases as'\Nil)
case Nil
with (m − m#) →* (Low-) have m = (Low-) by fastforce
with (valid-edge a) (sourcenode a = m) have targetnode a = (Exit-)
by (rule Exit-successor-of-Low,simp+)
from Low-source-Exit-edge obtain a' where valid-edge a'
and sourcenode a' = (Low-) and targetnode a' = (Exit-)
and kind a' = (\text{as}, \text{True}) by blast
from (valid-edge a) (sourcenode a = m) \langle m = (Low-) \rangle
\langle targetnode a = (Exit-) \rangle (valid-edge a') (sourcenode a' = (Low-))
\langle targetnode a' = (Exit-) \rangle
have a = a' by (fastforce dest:edge-det)
\end{verbatim}
with \(\langle \text{kind} \ a' = (\lambda s. \ True) \rangle\) have \(\text{kind} \ a = (\lambda s. \ True)\) by simp
with \((\langle \text{targetnode} \ a = \langle \text{Exit} \rangle \ Association \ a' \rightarrow \ (\langle \text{Low} \rangle) \rangle)\)
have \((\langle \text{Low} \rangle = (\langle \text{Exit} \rangle) \ by \ -(\text{drule \ path-Exit-source,auto})\) with False have False by -(drule \ Low-neq-Exit,simp)
thus \(\langle \text{thesis} \rangle\) by simp
next
  case (\langle \text{Cons} \ ax \ ax \rangle)
with \((\langle m \rightarrow \langle \text{as'} \rightarrow \langle \text{as} \rangle \rangle \rangle)\) have \(\langle \text{sourcecode} \ ax = m \ and \ valid-edge \ ax \rangle\)
and \(\langle \text{targetnode} \ ax = \langle \text{as} \rightarrow \langle \text{as} \rangle \rangle \ by \ (\text{auto \ elim:split-cons} \rangle\) from \(\langle \langle \text{preds} \ (\langle \text{slice-kinds} \ S \ (a \ # \ as) \rangle) \rangle \) obtain \(\langle \text{cfs} \ where \ \langle \text{simp} \rangle : s = \text{cf} \ # \ cfs \ by \ (\text{cases} \ s \) \ (\text{auto \ simp:slice-kinds-def} \rangle\) from \(\langle \langle \text{preds} \ (\langle \text{slice-kinds} \ S \ as \rangle) \ s \ \langle \text{as'} = \ ax \ # \ ax \rangle \) obtain \(\langle \text{cfs'} \ where \ \langle \text{simp} \rangle : s' = \text{cf'} \ # \ cfs' \by \ (\text{cases} \ s \) \ (\text{auto \ simp:slice-kinds-def} \rangle\) have intra-kind \(\langle \text{kind} \ ax \rangle\)
proof \(\langle \text{cases} \ \text{kind} \ ax \ \text{rule:edge-kind-cases} \rangle\) case (Call \(Q \ r \ p \ fs\) ) have False
proof \(\langle \text{cases} \ \text{sourcecode} \ ax \ \in \ \text{[HRB-slice} \ S]_{\text{CFG}} \rangle\)
case True
with \(\langle \text{intra-kind} \ (\langle \text{kind} \ a \rangle) \rangle\) have \(\text{slice-kinds} \ S \ a = \text{kind} \ a\)
by -(rule slice-intra-kind-in-slice)
from \(\langle \text{valid-edge} \ ax \rangle\) have \(\langle \text{kind} \ ax = Q; r \rightarrow \langle \text{ps} \rangle \rangle\) have unique:exists a'. valid-edge a' \ /
\(\langle \text{sourcecode} \ ax \ \text{and} \ \text{slice-kind} \ a' = \text{sourcecode} \ ax \ \text{and} \ \text{intra-kind} (\langle \text{kind} \ a' \rangle) \ by \ (\text{rule \ call-only-one-intra-edge} \rangle\) from \(\langle \text{valid-edge} \ ax \rangle\) have \(\langle \text{kind} \ ax = Q; r \rightarrow \langle \text{ps} \rangle \rangle\) obtain \(x\) where \(x \ \in \ \text{get-return-edges} \ ax \ by \ (\text{fastforce \ dest:call-return-edge-call} \rangle\)
with \(\langle \text{valid-edge} \ ax \rangle\) obtain \(\langle \text{a'} \ \text{where} \ \text{valid-edge} \ a' \rangle\)
and \(\text{sourcecode} \ ax \ a' = \text{sourcecode} \ ax \ and \ \text{kind} \ a' = (\lambda \ cf. \ False)\) by \(\langle \text{fastforce \ dest:call-return-node-edge} \rangle\)
with \(\langle \text{valid-edge} \ ax \rangle \ (\text{sourcecode} \ ax = m) \ (\text{sourcecode} \ ax \ \text{and} \ \text{slice-kinds} \ S \ (a \ # \ as) \ s)\) have False by \(\langle \text{cases} \ s \) \ (\text{auto \ simp:slice-kinds-def} \rangle\)
thus \(\langle \text{thesis} \rangle\) by simp
next
  case False
with \(\langle \text{kind} \ ax = Q; r \rightarrow \langle \text{ps} \rangle \rangle \ (\text{sourcecode} \ ax = m) \ (\text{sourcecode} \ ax = m)\) have \(\text{slice-kinds} \ S \ ax = (\lambda \ cf. \ False); r \rightarrow \langle \text{ps} \rangle \)
by \(\langle \text{fastforce \ intro:split-kind-Call} \rangle\)
with \(\langle \text{as'} = \ ax \ # \ axx \rangle\) obtain \(\langle \text{preds} \ (\langle \text{slice-kinds} \ S \ as' \rangle) \ s' \rangle\)
have False by \(\langle \text{cases} \ s' \) \ (\text{auto \ simp:slice-kinds-def} \rangle\)
thus \(\langle \text{thesis} \rangle\) by simp
qed
thus \(\langle \text{thesis} \rangle\) by simp
next
  case (Return \(Q \ p \ f))
from ⟨valid-edge ax⟩ :kind ax = Q←p⌉ ⟨valid-edge a⟩ : intra-kind (kind a) ⟨source-node a = m⟩ ⟨source-node ax = m⟩
have False by ⟨(drule return-edges-only,auto simp:intra-kind-def)⟩
thus ?thesis by simp
qed simp
with ⟨same-level-path-aux cs asx'⟩ (asx' = ax#asx)
have same-level-path-aux cs asx by ⟨(fastforce simp:intra-kind-def)⟩
show ?thesis
proof ⟨cases target-node a = target-node ax⟩
case True
with ⟨valid-edge a⟩ ⟨valid-edge ax⟩ ⟨source-node a = m⟩ ⟨source-node ax = m⟩
have a = ax by ⟨(fastforce intro:edge-def)⟩
with ⟨valid-edge a⟩ ⟨intra-kind (kind a)⟩ ⟨source-node a = m⟩
∀ V ∈ rv ⟨CFG-node m⟩. state-val s V = state-val s' V
⟨preds (slice-kinds S (a ≠ as))⟩ s
⟨preds (slice-kinds S as)⟩ s' (as' = ax # asx)
have rvː∀ V ∈ rv ⟨CFG-node (target-node a)⟩.
state-val ⟨transfer (slice-kind S a) s⟩ V =
state-val ⟨transfer (slice-kind S a) s'⟩ V
by ⟨(rule rv-edge-slice-kinds,auto)⟩
from ⟨upd-cs cs (a ≠ as) = []⟩ ⟨intra-kind (kind a)⟩
have upd-cs cs as = [] by ⟨(fastforce simp:intra-kind-def)⟩
from ⟨target-node ax → ax#→∗ (-Low-)⟩ (a = ax)
have target-node a → ax#→∗ (-Low-) by simp
from ⟨valid-edge a⟩ ⟨intra-kind (kind a)⟩
obtain cfx
where cfx:transfer (slice-kind S a) s = cfx#cfs ∧ snd cfx = snd cf
apply ⟨cases cf⟩
apply ⟨cases source-node a ∈ [HRB-slice S]CFG⟩ apply auto
apply ⟨(fastforce dest:slice-intra-kind-in-slice simp:intra-kind-def)⟩
apply ⟨(auto simp:intra-kind-def)⟩
apply ⟨(drule slice-kind-Upd)⟩ apply auto
by ⟨(erule kind-Predicate-notin-slice-slice-kind-Predicate)⟩ auto
from ⟨valid-edge a⟩ ⟨intra-kind (kind a)⟩
obtain cfx'
where cfx':transfer (slice-kind S a) s' = cfx'#cfs' ∧ snd cfx' = snd cf'
apply ⟨cases cf'⟩
apply ⟨cases source-node a ∈ [HRB-slice S]CFG⟩ apply auto
apply ⟨(fastforce dest:slice-intra-kind-in-slice simp:intra-kind-def)⟩
apply ⟨(auto simp:intra-kind-def)⟩
apply ⟨(drule slice-kind-Upd)⟩ apply auto
by ⟨(erule kind-Predicate-notin-slice-slice-kind-Predicate)⟩ auto
with cfx ⟨∀ i < Suc (length cs), snd (s*i) = snd (s'!i)⟩
snd ⟨transfer (slice-kind S a) s ! i⟩ =
snd ⟨transfer (slice-kind S a) s' ! i⟩
by auto ⟨case-tac i,auto⟩
from ⟨rsv cfx cfx'⟩ have ⟨rsv:∀ i < length cs.
∀ V ∈ rv S ⟨CFG-node (source-node (cs ! i))⟩⟩.

10
\[\text{fst} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s ! \ Suc \ i) \ V =\]
\[\text{fst} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s' ! \ Suc \ i) \ V\]
\[\text{by fastforce}\]

\text{from} \ (\text{preds} \ (\text{slice-kinds} \ S \ (a \ # \ as)) \ s)\]
\text{have} \ (\text{preds} \ (\text{slice-kinds} \ S \ as))\]
\[\text{by}(\text{simp add: slice-kinds-def})\]
\text{moreover}\]
\text{from} \ (\text{preds} \ (\text{slice-kinds} \ S \ as') \ s') \ (as' = a x \ # \ ax) \ (a = ax)\]
\text{have} \ (\text{preds} \ (\text{slice-kinds} \ S \ ax)) \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s')\]
\[\text{by}(\text{simp add: slice-kinds-def})\]
\text{moreover}\]
\text{from} \ (\text{valid-edge} \ a; \ (\text{intra-kind} \ (\text{kind} \ a))\]
\text{have} \ length \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s) = \text{length} \ s\]
\[\text{by}(\text{cases sourcenode} \ ax \ \in \ [\text{HRB-slice} \ S]_{\text{CFG}})\]
\text{(auto dest: slice-intra-kind-in-slice slice-kind-Upd)}\]
\text{elim:kind-Predicate-notin-slice-slice-kind-Predicate simp:intra-kind-def)}\]
\text{with} \ (\text{length} \ s = \text{Suc} \ (\text{length} \ cs))\]
\text{have} \ length \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s) = \text{Suc} \ (\text{length} \ cs)\]
\[\text{by simp}\]
\text{moreover}\]
\text{from} \ (a = ax) \ (\text{valid-edge} \ a; \ (\text{intra-kind} \ (\text{kind} \ a))\]
\text{have} \ length \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s') = \text{length} \ s'\]
\[\text{by}(\text{cases sourcenode} \ ax \ \in \ [\text{HRB-slice} \ S]_{\text{CFG}})\]
\text{(auto dest: slice-intra-kind-in-slice slice-kind-Upd)}\]
\text{elim:kind-Predicate-notin-slice-slice-kind-Predicate simp:intra-kind-def)}\]
\text{with} \ (\text{length} \ s' = \text{Suc} \ (\text{length} \ cs))\]
\text{have} \ length \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s') = \text{Suc} \ (\text{length} \ cs)\]
\[\text{by simp}\]
\text{moreover}\]
\text{from} \ IH | OF \ (\text{upd-cs} \ cs \ as = []) \ (\text{same-level-path-aux} \ cs \ ax)\]
\[\forall \ c | \text{set} \ cs \ \text{valid-edge} \ c; \ (\text{targetnode} \ ax \ \rightarrow^{*} \ (\text{-Low-}));\]
\[\text{(targetnode} \ ax \ \rightarrow^{*} \ (\text{-Low-})); \ \text{res'} \ \text{snds} \ \text{re calculation}]\]
\[a = ax \ # \ ax) \ (a = ax)\]
\text{show \ ?thesis \ by}(\text{simp add: slice-kinds-def})\]
\text{next}\]
\text{case} \ False\]
\text{from} \ \forall \ i < \text{Suc}(\text{length} \ cs). \ \text{snd} \ (s!i) = \text{snd} \ (s'!i)\]
\text{have} \ \text{snd} \ (\text{hd} \ s) = \text{snd} \ (\text{hd} \ s') \ \text{by}(\text{erule-tac} \ x=0 \ \text{in allE}) \ \text{fastforce}\]
\text{with} \ (\text{valid-edge} \ a) \ (\text{valid-edge} \ ax) \ (\text{source-node} \ ax = m)\]
\[\text{source-node} \ ax = m) \ (as' = ax \ # \ ax) \ False\]
\[\text{(intra-kind} \ (\text{kind} \ a)) \ (\text{intra-kind} \ (\text{kind} \ ax))\]
\[\text{preds} \ (\text{slice-kinds} \ S \ (a \ # \ as)) \ s;\]
\[\text{preds} \ (\text{slice-kinds} \ S' \ (a' \ # \ as)) \ s'\]
\[\forall \ V \ \in \text{rv} \ S \ (\text{CFG-node} \ m). \ \text{state-val} \ s \ V = \text{state-val} \ s' \ V\]
\[\text{length} \ s = \text{Suc} \ (\text{length} \ cs); \ \text{length} \ s' = \text{Suc} \ (\text{length} \ cs)\]
\text{have} \ False \ \text{by}(\text{fastforce intro!:re-branching-edges-slice-kinds-False[of \ ax]}\]
\text{thus} \ ?thesis \ \text{by simp}\]
\text{qed}\]
\text{qed}
qed

next
case (slpa-Call cs a as Q r p fs)

note IH = \( i \land m as' s s' \).

\[ \text{upd-CS} (a \not= cs) as = []; \, \text{same-level-path-aux} (a \not= cs) as'; \]
\( \forall c \in \text{set} (a \not= cs). \, \text{valid-edge} c; \, m \not\rightarrow s' (\text{Low}); \, m \not\rightarrow s' (\text{Low}); \)
\( \forall i < \text{length} (a \not= cs). \, \forall V \in \text{rv} S \, (\text{CFG-node} (\text{source-node} ((a \not= cs) ! i))). \)

\( \text{fst} (s ! \text{Suc} i) V = \text{fst} (s' ! \text{Suc} i) V; \)
\( \forall i < \text{Suc} \, (\text{length} (a \not= cs)). \, \text{snd} (s ! i) = \text{snd} (s' ! i); \)
\( \forall V \in \text{rv} S \, (\text{CFG-node} m). \, \text{state-val} \, V = \text{state-val} s' V; \)
\( \text{preds} (\text{slice-kinds} S \, \text{as}) s; \, \text{preds} (\text{slice-kinds} S \, \text{as'}) s'; \)
\( \text{length} s = \text{Suc} \, (\text{length} (a \not= cs)); \, \text{length} s' = \text{Suc} \, (\text{length} (a \not= cs)) \]
\( \Rightarrow \forall V \in \text{Use} (\text{Low}). \, \text{state-val} (\text{transfers} (\text{slice-kinds} S \, \text{as}) s) V = \text{state-val} (\text{transfers} (\text{slice-kinds} S \, \text{as'}) s') V; \)

note rvs = \( \forall i < \text{length} cs. \, \forall V \in \text{rv} S \, (\text{CFG-node} (\text{source-node} (cs ! i))). \)

\( \text{fst} (s ! \text{Suc} i) V = \text{fst} (s' ! \text{Suc} i) V \)

from \((m \not\rightarrow s' as' \not\rightarrow s (\text{Low})); \, \text{have} \, \text{source-node} a = m \, \text{and} \, \text{valid-edge} a \)
\( \, \text{and} \, \text{target-node} a \not\rightarrow s (\text{Low}) \, \text{by} (\text{auto elim: path-split-Cons}) \)

from \( \forall c \in \text{set} cs. \, \text{valid-edge} c \, (\text{valid-edge} a) \)
\( \, \text{have} \, \forall c \in \text{set} (a \not= cs), \, \text{valid-edge} c \, \text{by simp} \)

show \?case

proof (cases L = \{\})

case True with UseLow show \?thesis by simp

next

case False

show \?thesis

proof (cases as')

case Nil

with \((m \not\rightarrow s' \not\rightarrow s (\text{Low})); \, \text{have} \, m = (\text{Low}) \, \text{by fastforce} \)

with \(\text{valid-edge} a; \, \text{source-node} a = m \, \text{have} \, \text{target-node} a = (\text{Exit}) \)
\( \, \text{by} (\text{rule Exit-successor-of-Low simp+}) \)

from \text{Low-source-Exit-edge} \, \text{obtain} \, a' \, \text{where} \, \text{valid-edge} a' \)
\( \, \text{and} \, \text{source-node} a' = (\text{Low}) \, \text{and} \, \text{target-node} a' = (\text{Exit}) \)
\( \, \text{and} \, \text{kind} a' = (\text{Lex True}) \, \text{by blast} \)

from \(\text{valid-edge} a; \, \text{source-node} a = m \); \(m = (\text{Low}) \)
\( \, \text{ obtain} \, (\text{target-node} a = (\text{Exit}) \, \text{valid-edge} a' \, \text{source-node} a' = (\text{Low}) \)
\( \, \text{target-node} a' = (\text{Exit}) \)

\( \, \text{have} \, a = a' \, \text{by fastforce dest: edge-det} \)

with \(\text{kind} a' = (\text{Lex True}) \, \text{have} \, \text{kind} a = (\text{Lex True}) \, \text{by simp} \)

with \(\text{target-node} a = (\text{Exit}) \, \text{target-node} a \not\rightarrow (\text{Low}) \)

\( \, \text{have} \, (\text{Low}) = (\text{Exit}) \, \text{by} (\text{drule path-Exit-source auto}) \)

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

thus \?thesis by simp

next

case (Cons a x asx)

with \((m \not\rightarrow s' \not\rightarrow s (\text{Low})); \, \text{have} \, \text{source-node} a = m \, \text{and} \, \text{valid-edge} a \)
\( \, \text{and} \, \text{target-node} a \not\rightarrow (\text{Low}) \, \text{by (auto elim: path-split-Cons}) \)

from \(\text{preds} (\text{slice-kinds} S \, (a \not= as)) s \)

obtain cf cfs where \(\text{simp}: s = \text{cf cfs} \, \text{by} (\text{cases s}) (\text{auto simp: slice-kinds-def}) \)
from \(\langle \text{preds (slice-kinds } S \text{ as') } s' \rangle \langle \text{as'} = ax \# asx \rangle\)

obtain \(cf' cf's'\) where \([\text{simp}]: s' = cf' \# cf's'\)
by \((\text{cases } s')(\text{auto simp:slice-kinds-def})\)

have \(\exists Q r p fs. \text{kind } ax = Q: r \rightarrow p fs\)
proof \((\text{cases kind } ax \text{ rule:edge-kind-cases})\)

\textbf{case Intra}

have False
proof \((\text{cases source node } ax \in [HRB-slice } S]_{CFG})\)

\textbf{case True}

with \(\langle \text{intra-kind (kind } ax)\rangle\)

have slice-kind \(S\ ax = \text{kind } ax\)
by \(\neg (\text{rule slice-intra-kind-in-slice})\)

from \(\text{valid-edge } a\ (\text{kind } a = Q: r \rightarrow p fs)\)
have unique: \(\exists a'. \text{valid-edge } a' \land \text{source node } a' = \text{source node } a \land \text{intra-kind (kind } a')\)
by \(\text{(rule call-only-one-intra-edge)}\)

from \(\text{valid-edge } a\ (\text{kind } a = Q: r \rightarrow p fs)\) obtain \(x\)
where \(x \in \text{get-return-edges } a\) by \(\text{(fastforce dest: get-return-edge-call)}\)

with \(\text{valid-edge } a\) obtain \(a'\) where \(\text{valid-edge } a'\)
and source node \(a' = \text{source node } a\) and kind \(a' = (\lambda ax. \text{False})\)
by \(\text{(fastforce dest: call-return-node-edge)}\)

with \(\text{valid-edge } ax\) \(\langle \text{source node } ax = m\rangle\) \(\langle \text{source node } a = m\rangle\)
\(\langle \text{intra-kind (kind } ax)\rangle\) unique

have \(a' = ax\) by \(\text{(fastforce simp: intra-kind-def)}\)

with \(\langle \text{kind } a' = (\lambda ax. \text{False})\rangle\)
\(\langle \text{slice-kind } S \ ax = \text{kind } ax\rangle\) \(\langle \text{as'} = ax \# asx \rangle\)
\(\langle \text{preds (slice-kinds } S \text{ as') } s' \rangle\)
have False by \(\text{(simp add: slice-kinds-def)}\)
thus \(?thesis by simp\)

\textbf{next}

\textbf{case False}

with \(\langle \text{kind } a = Q: r \rightarrow p fs\rangle\) \(\langle \text{source node } ax = m\rangle\) \(\langle \text{source node } a = m\rangle\)

have slice-kind \(S\ a = (\lambda ax. \text{False}) : r \rightarrow p fs\)
by \(\text{(fastforce intro:slice-kind-Call)}\)

with \(\langle \text{preds (slice-kinds } S \ (a \# as) \rangle \ s\rangle\)

have False by \(\text{(simp add: slice-kinds-def)}\)
thus \(?thesis by simp\)

qed

thus \(?thesis by simp\)

\textbf{next}

\textbf{case (Return } Q' \ p' f')

from \(\langle \text{valid-edge } ax\rangle\) \(\langle \text{kind } ax = Q' : p \rightarrow f'\rangle\) \(\langle \text{valid-edge } a\rangle\)
\(\langle \text{kind } a = Q: r \rightarrow p fs\rangle\)

\(\langle \text{source node } a = m\rangle\) \(\langle \text{source node } ax = m\rangle\)

have False by \(\neg (\text{drule return-edges-only,auto})\)
thus \(?thesis by simp\)

qed simp

have source node \(a \in \lfloor HRB\text{-slice } S \rfloor_{CFG}\)

proof \(\text{(rule ccontr)}\)

assume source node \(a \notin \lfloor HRB\text{-slice } S \rfloor_{CFG}\)
from this \(\langle \text{kind } a = Q: r \rightarrow p fs\rangle\)
have slice-kind $S$ $a$ = (λ$e$. False):$r$→$_p$f$s$
  by (rule slice-kind-Call)
  with ⟨$\langle$ slice-kinds $S$ (a ≠ as)$\rangle$ $s$⟩
  show False by (simp add: slice-kinds-def)
qed
with ⟨$\langle$ slice-kinds $S$ (a ≠ as)$\rangle$ $s$⟩ ⟨kind $a$ = $Q$:r→$_p$f$s$⟩
have pred (kind a) $s$
  by (fastforce dest: slice-kind-Call-in-slice simp: slice-kinds-def)
from ⟨source-node a ∈ [HRB-slice $S$]$_{CFG}$⟩
  ⟨source-node a ∈ $m$⟩ (source-node ax = $m$)
have source-node ax ∈ [HRB-slice $S$]$_{CFG}$ by simp
with ⟨$\langle$ as' = ax # axs$\rangle$ $\langle$ slice-kinds $S$ as$'$ $s'$⟩
  $\exists$ Q $r$ $p$ $fs$. kind ax = $Q$:r→$_p$f$s$
have pred (kind ax) $s'$
  by (fastforce dest: slice-kind-Call-in-slice simp: slice-kinds-def)
{ fix V assume $V$ ∈ Use (source-node a)
  from ⟨valid-edge a$'$⟩ have source-node a →$\rightarrow$*$\rightarrow$ source-node a
    by (fastforce intro: empty-path simp: intra-path-def)
  with ⟨source-node a ∈ [HRB-slice $S$]$_{CFG}$⟩
    ⟨valid-edge a$'$⟩ $\langle$ $V$ ∈ Use (source-node a)$\rangle$
  have $V$ ∈ rv $S$ (CFG-node (source-node a))
by (auto intro!: rv! CFG-Use-SDG-Use simp: SDG-to-CFG-set-def source-nodes-def)
}
with $\forall$ V ∈ rv $S$ (CFG-node-m). state-val s $V$ = state-val $s'$ $V$
  ⟨source-node a = $m$⟩
have Use! $V$ ∈ Use (source-node a). state-val s $V$ = state-val $s'$ $V$ by simp
from $\forall i$< Suc (length cs). snd ($s$ ! $i$) = snd ($s'$ ! $i$)
have snd ($hd$ $s$) = snd ($hd$ $s'$) by fastforce
with ⟨valid-edge a$'$⟩ ⟨kind $a$ = $Q$:r→$_p$f$s$⟩ ⟨valid-edge ax⟩
  $\exists$ Q $r$ $p$ $fs$. kind ax = $Q$:r→$_p$f$s$
  ⟨source-node a = $m$⟩ (source-node ax = $m$)
  ⟨pred (kind a) $s$⟩ ⟨pred (kind ax) $s'$⟩ $\langle$ Use $\langle$ length $s$ = Suc (length cs)$\rangle$
  ⟨length $s'$ = Suc (length cs)$\rangle$
have [simp]: ax = $a$ by (fastforce intro!: CFG-equal-Use-equal-call)
from ⟨same-level-path-aux cs as$'$⟩ ⟨as' = as#axs$⟩ ⟨kind $a$ = $Q$:r→$_p$f$s$⟩
  $\exists$ Q $r$ $p$ $fs$. kind ax = $Q$:r→$_p$f$s$
have same-level-path-aux (a ≠ cs) axs by simp
from ⟨target-node ax – axs→# (Low-1)⟩ have target-node a – axs→# (Low-1)
by simp
from ⟨kind $a$ = $Q$:r→$_p$f$s$⟩ ⟨upd-cs cs (a ≠ as) = []⟩
have upd-cs (a ≠ cs) as = [] by simp
from ⟨source-node a ∈ [HRB-slice $S$]$_{CFG}$⟩ ⟨kind $a$ = $Q$:r→$_p$f$s$⟩
have slice-kind: slice-kind $S$ $a$ =
  $Q$:r→$_p$ (csp (target-node a) (HRB-slice $S$) $fs$)
by (rule slice-kind-Call-in-slice)
from $\forall i$< Suc (length cs). snd ($s$ ! $i$) = snd ($s'$! $i$) slice-kind
have snds $\forall i$< Suc (length (a ≠ cs)).
  snd (transfer (slice-kind $S$ a) $s$ ! $i$) =
  snd (transfer (slice-kind $S$ a) $s'$! $i$)
by auto (case-tac i, auto)
from ⟨valid-edge a⟩ (kind a = Q:r→p fs) obtain ins outs
  where ⟨p, ins, outs⟩ ∈ set procs by (fastforce dest!: callee-in-procs)
with ⟨valid-edge a⟩ (kind a = Q:r→p fs)
have length ⟨ParamUses (sourcenode a)⟩ = length ins
  by (fastforce intro: ParamUses-call-source-length)
with ⟨valid-edge a⟩
  have ∀ i < length ins. ∀ V ∈ ⟨ParamUses (sourcenode a)⟩ i V.
  V ∈ Use (sourcenode a)
  by (fastforce intro: ParamUses-in-Use)
with ∀ V ∈ Use (sourcenode a). state-val s V = state-val s' V
have ∀ i < length ins. ∀ V ∈ ⟨ParamUses (sourcenode a)⟩ i V.
  state-val s V = state-val s' V
  by fastforce
with ⟨valid-edge a⟩ (kind a = Q:r→p fs) ⟨p, ins, outs⟩ ∈ set procs
  ⟨pred (kind a) s⟩ ⟨pred (kind ax) s’⟩
have ∀ i < length ins. ⟨params fs (fst (hd s))⟩ i = ⟨params fs (fst (hd s’))⟩ i
  by (fastforce intro: CFG-call-edge-params)
from ⟨valid-edge a⟩ (kind a = Q:r→p fs) ⟨p, ins, outs⟩ ∈ set procs
have length fs = length ins by (rule CFG-call-edge-length)
{ fix i assume i < length fs
  with ⟨length fs = length ins⟩ have i < length ins by simp
  from ⟨i < length fs⟩ have ⟨params fs (fst cf)⟩ i = ⟨fs!i⟩ ⟨fst cf⟩
    by (rule params-nth)
  moreover
  from ⟨i < length fs⟩ have ⟨params fs (fst cf’)⟩ i = ⟨fs!i⟩ ⟨fst cf’⟩
    by (rule params-nth)
  ultimately have ⟨fs!i⟩ ⟨fst (hd s)⟩ = ⟨fs!i⟩ ⟨fst (hd s’)⟩
    using ⟨i < length ins⟩
    ⟨∀ i < length ins. ⟨params fs (fst (hd s))⟩ i = ⟨params fs (fst (hd s’))⟩ i⟩
    by simp }
hence ∀ i < length fs. ⟨fs ! i⟩ ⟨fst cf⟩ = ⟨fs ! i⟩ ⟨fst cf’⟩ by simp
{ fix i assume i < length fs
  with ∀ i < length fs. ⟨fs ! i⟩ ⟨fst cf⟩ = ⟨fs ! i⟩ ⟨fst cf’⟩
  have ⟨fs ! i⟩ ⟨fst cf⟩ = ⟨fs ! i⟩ ⟨fst cf’⟩ by simp
  have ⟨(csppa (targetnode a) (HRB-slice S) 0 fs)⟩ i ⟨fst cf⟩ =
    ⟨(csppa (targetnode a) (HRB-slice S) 0 fs)⟩ i ⟨fst cf’⟩
  proof (cases Formal-in ⟨targetnode a, i + 0⟩ ∈ HRB-slice S)
  case True
    with ⟨i < length fs⟩
    have ⟨csppa (targetnode a) (HRB-slice S) 0 fs⟩ i = fs!i
      by (rule csppa-Formal-in-in-slice)
    with ⟨(fs ! i) ⟨fst cf⟩ = ⟨fs ! i⟩ ⟨fst cf’⟩⟩ show ?thesis by simp
next
  case False
    with ⟨i < length fs⟩
    have ⟨csppa (targetnode a) (HRB-slice S) 0 fs⟩ i = empty
      by (rule csppa-Formal-in-notin-slice)
    thus ?thesis by simp
  qed }
**hence** eq:\(\forall i < \text{length } fs\).  
\((\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)!i)(\text{fst } cf) =  
\((\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)!i)(\text{fst } cf')  
\text{by (simp add: csp-def)}

{ fix \(i\) assume \(i < \text{length } fs\)  
**hence** (\text{params } (\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)  
\((\text{fst } cf')!i\) =  
\((\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)!i)(\text{fst } cf')  
\text{by (fastforce intro: params-nth)}

**moreover**  
from \((i < \text{length } fs)\)  
**have** (\text{params } (\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)  
\((\text{fst } cf')!i\) =  
\((\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)!i)(\text{fst } cf')  
\text{by (fastforce intro: params-nth)}

**ultimately**  
**have** (\text{params } (\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)  
\((\text{fst } cf')!i\) =  
(\text{params } (\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)(\text{fst } cf')!i)  
\text{using eq } (i < \text{length } fs) \text{ by simp }

**hence** params (\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)(\text{fst } cf) =  
params (\text{csp}(\text{targetnode } a) (\text{HRB-slice } S) fs)(\text{fst } cf')  
\text{by (simp add: list-eq-iff-nth-eq)}

with slice-kind \(((p, ins, outs) \in \text{set procs})\)

**obtain** cfr \textbf{where} [\text{simp}]:  
\text{transfer} (\text{slice-kind } S a) (cf \# cfs) = cfx \# cf \# cfs  
\text{transfer} (\text{slice-kind } S a) (cf' \# cfs') = cfz \# cf' \# cfs'  
\text{by auto}

**hence** rv: \(\forall V \in rv S \ (\text{CFG-node } (\text{targetnode } a))\).  
\text{state-val} (\text{transfer} (\text{slice-kind } S a) s) V =  
\text{state-val} (\text{transfer} (\text{slice-kind } S a) s') V \text{ by simp}

from \(\forall V \in rv S \ (\text{CFG-node } m). \text{ state-val } s V = \text{state-val } s' V\)  
\langle source-node a = m \rangle

**have** res': \(\forall i < \text{length } (a \neq cs)\).  
\(\forall V \in rv S \ (\text{CFG-node } (\text{source-node } ((a \neq cs) ! i)))\).  
\(\text{fst} ((\text{transfer} (\text{slice-kind } S a) s) ! \text{Suc } i) V =  
\text{fst} ((\text{transfer} (\text{slice-kind } S a) s') ! \text{Suc } i) V  
\text{by auto (case-tac i, auto)}

from \(\langle \text{preds } (\text{slice-kinds } S (a \neq as)) s\rangle\)

**have** preds (\text{slice-kinds } S as)  
(\text{transfer} (\text{slice-kind } S a) s) \text{ by (simp add: slice-kinds-def)}

**moreover**  
from \(\langle \text{preds } (\text{slice-kinds } S \text{ as} s') s' (as' = ax \# asx)\rangle\)

**have** preds (\text{slice-kinds } S \text{ asx})  
(\text{transfer} (\text{slice-kind } S a) s') \text{ by (simp add: slice-kinds-def)}

**moreover**  
from \(\langle \text{length } s = \text{Suc } (\text{length } cs)\rangle\)

**have** length (\text{transfer} (\text{slice-kind } S a) s) =  
\text{Suc } (\text{length } (a \neq cs)) \text{ by simp}
moreover
from \(\text{length } s' = \text{Succ}(\text{length } cs)\)

have \(\text{length } (\text{transfer } \text{slice-kind } S a) s' = \text{Succ}(\text{length } (a \neq cs))\) by simp

moreover
from \(IH[\text{OF } \langle \text{upd-cs } (a \neq cs) \rangle = []; \text{same-level-path-aux } (a \neq cs) \text{ as } s;\)

\(\forall c \in \text{set } (a \neq cs), \text{valid-edge } c; \text{targetnode } a \to as\to (\text{-Low-});\)

\(\forall \text{targetnode } a \to as\to (\text{-Low-});\) \(\text{vs'} \text{ nds } \text{rv calculation}; \langle as' = ax \# axs\rangle\)

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

qed

qed

next

case \((\text{slpa-Return } cs a as Q p f c' cs')\)

note \(IH = \langle \forall m as' s s', \langle \text{upd-cs } cs' as = []; \text{same-level-path-aux } cs' as';\)

\(\forall c \in \text{set } cs', \text{valid-edge } c; m \to as\to (\text{-Low-}); m \to as'\to (\text{-Low-});\)

\(\forall i < \text{length } cs', \forall V \in \text{rv } S \text{(CFG-node } (\text{source-node } (cs' i))\).

\(\text{fst } (s ! \text{Succ } i) V = \text{fst } (s' ! \text{Succ } i) V;\)

\(\forall i < \text{Succ } (\text{length } cs'). \text{snd } (s ! i) = \text{snd } (s' ! i);\)

\(\forall V \in \text{rv } S \text{(CFG-node } m), \text{state-val } s V = \text{state-val } s' V;\)

\(\text{preds } (\text{slice-kinds } S as) s; \text{preds } (\text{slice-kinds } as' s) s;\)

\(\text{length } s = \text{Succ } (\text{length } cs'); \text{length } s' = \text{Succ } (\text{length } cs');\)

\(\implies \forall V \in \text{Use } (\text{-Low-}), \text{state-val } (\text{transfers } (\text{slice-kinds } as') s) V = \text{state-val } (\text{transfers } (\text{slice-kinds } S as') s') V';\)

note \(\text{vs} = \langle \forall i < \text{length } cs, \forall V \in \text{rv } S \text{(CFG-node } (\text{source-node } (cs ! i))\).

\(\text{fst } (s ! \text{Succ } i) V = \text{fst } (s' ! \text{Succ } i) V\)

from \((m \to a \neq as\to (\text{-Low-});\) have \(\text{source-node } a = m \text{ and } \text{valid-edge } a\)

and \(\text{target-node } a \to as\to (\text{-Low-})\) by \(\text{auto elim:path-split-Cons}\)

from \(\forall c \in \text{set } cs, \text{valid-edge } c; \langle cs = c' \# cs'\rangle\)

have \(\text{valid-edge } c' \text{ and } \forall c \in \text{set } cs'. \text{valid-edge } c\) by simp-all

show \(?\text{case}\)

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

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

next

case \(\text{False}\)

show \(?\text{thesis}\)

proof(cases \(as'\))

case \(\text{Nil}\)

with \((m \to as'\to (\text{-Low-});)\) have \(m = (\text{-Low-})\) by fastforce

with \(\langle \text{valid-edge } a; \text{source-node } a = m\rangle\) have \(\text{target-node } a = (\text{-Exit-})\)

by \(\text{-(rule Exit-successor-of-Low,simp+)}\)

from \(\text{Low-source-Exit-edge obtain } a' \text{ where } \text{valid-edge } a'\)

and \(\text{source-node } a' = (\text{-Low-}) \text{ and } \text{target-node } a' = (\text{-Exit-})\)

and \(\text{kind } a' = (\lambda s. \text{True})\) by blast

from \(\langle \text{valid-edge } a; \text{source-node } a = m\rangle\) \(m = (\text{-Low-})\)

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

\(\langle \text{target-node } a' = (\text{-Exit-})\rangle\)

have \(a = a'\) by \(\text{fastforce dest:edge-det}\)

with \(\langle \text{kind } a' = (\lambda s. \text{True})\rangle\) have \(\text{kind } a = (\lambda s. \text{True})\) by simp

with \(\langle \text{target-node } a = (\text{-Exit-})\rangle\) \(\langle \text{target-node } a \to as\to (\text{-Low-})\rangle\)
have ($\text{-Low}$) = ($\text{-Exit}$) by $(\text{drule path-Exit-source,auto})$

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

thus $\text{thesis}$ by simp

next
case (Cons $ax$ $asx$)

with $(m \rightarrow as' \rightarrow^+ (\text{-Low})$) have $\text{source-node} ax = m$ and $\text{valid-edge} ax$

and $\text{target-node} ax \rightarrow^+ (\text{-Low})$ by $(\text{auto elim:path-split-Cons})$

from ($\text{valid-edge} a$) ($\text{valid-edge} ax$) ($\text{kind} a = Q \rightarrow p f$)

have $\exists Q f$, $\text{kind} ax = Q \rightarrow p f$ by $(\text{auto dest:return-edges-only})$

with $(\text{same-level-path-axv} cs as' a)$ ($as' = ax \rightarrow asx$) ($cs = c' \rightarrow cs'$)

have $ax \in \text{get-return-edges} c'$ and $(\text{same-level-path-axv} cs as' ax)$ by auto

from ($\text{valid-edge} c'$) ($ax \in \text{get-return-edges} c'$) ($a \in \text{get-return-edges} c'$)

have $\exists (cs)$: $ax = a$ by $(\text{rule get-return-edges-unique})$

from $(\text{target-node} ax \rightarrow as \rightarrow^+ (\text{-Low}))$ have $\text{target-node} a \rightarrow as \rightarrow^+ (\text{-Low})$

by simp

from $(\text{upd-cs} cs (a \not= as)) = []$ ($\text{kind} a = Q \rightarrow p f$) ($cs = c' \rightarrow cs'$)

$(a \in \text{get-return-edges} c')$

have $(\text{upd-cs} cs a)$ as $[]$ by simp

from $(\text{length} s = \text{Suc} (\text{length} cs)$) ($cs = c' \rightarrow cs'$)

obtain $cf cfx cfs$ where $s = cf \rightarrow cfx \rightarrow cfs$

by $(\text{cases} s, \text{auto, case-tac list, fastforce+})$

from $(\text{length} s' = \text{Suc} (\text{length} cs))$ ($cs = c' \rightarrow cs'$)

obtain $cf cfx' cfs'$ where $s' = cf' \rightarrow cfx' \rightarrow cfs'$

by $(\text{cases} s', \text{auto, case-tac list, fastforce+})$

from $\text{res} (\text{cs} = c' \rightarrow cs'$) ($s = cf \rightarrow cfx \rightarrow cfs$) ($s' = cf' \rightarrow cfx' \rightarrow cfs'$)

have $\exists \forall i < \text{length} cs'$.

$\forall V \in \text{rv} S (\CFG-node (\text{source-node} (cs' \rightarrow i)))$

$fst ((\text{cfx} \rightarrow cfs)) \rightarrow \text{Suc} i V = fst ((\text{cfx'} \rightarrow cfs')) \rightarrow \text{Suc} i V$

and $\forall V \in \text{rv} S (\CFG-node (\text{source-node} c'))$

$(\text{fst cfx}) V = (\text{fst cfx'}) V$

by auto

from ($\text{valid-edge} c'$) ($a \in \text{get-return-edges} c'$)

obtain $Qx rx px fsx$ where $\text{kind} c' = Qx:rx \rightarrow px fsx$

by $(\text{fastforce dest!:only-call-get-return-edges})$

have $\forall V \in \text{rv} S (\CFG-node (\text{target-node} a))$

$V \in \text{rv} S (\CFG-node (\text{source-node} c'))$

proof

fix $V$ assume $V \in \text{rv} S (\CFG-node (\text{target-node} a))$

from ($\text{valid-edge} c'$) ($a \in \text{get-return-edges} c'$)

obtain $a' \rightarrow \text{edge:valid-edge} a' \rightarrow \text{source-node} a' = \text{source-node} c'$

$\text{target-node} a' = \text{target-node} a \rightarrow \text{intra-kind} (\text{kind} a')$

by $(\text{drule call-return-node-edge,auto simp:intra-kind-def})$

from $(\forall V \in rv S (\CFG-node (\text{target-node} a)))$

obtain as $n'$ where $\text{target-node} a \rightarrow as \rightarrow^+ \text{parent-node} n'$

and $n' \in \text{HRB-slice} S$ and $V \in \text{Use_SDG} n'$

and $\forall n''$. valid-SDG-node $n'' \rightarrow \text{parent-node} n'' \in \text{set} (\text{source-nodes as})$

$\rightarrow V \notin \text{Def_SDG} n''$ by $(\text{fastforce elim:rvE})$
proof

\begin{enumerate}
\item \begin{align*}
\text{from } \langle \text{targetnode } a \rightarrow \ast \text{ parent-node } n \rangle \text{ edge } & \\
\text{have } \text{source-node } c' \rightarrow a' \# \ast \rightarrow \ast \text{ parent-node } n' & \\
\text{by} (\text{fastforce intro: Cons-path simp: intra-path-def})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{from } \langle \text{valid-edge } c' \rangle \langle \text{kind } c' = Qx:x \leftarrow pxfsx \rangle & \text{ have } \text{Def } (\text{source-node } c') = \\
\text{by} (\text{rule call-source-Def-empty})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{hence } \forall n''. \text{ valid-SDG-node } n'' \land \text{parent-node } n'' = \text{source-node } c' & \\
\rightarrow V \notin \text{Def SDG } n'' \text{ by} (\text{fastforce dest: SDG-Def-parent-Def})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{with } \langle \text{source-node } a' = \text{source-node } c' \rangle & \\
\text{have } \forall n''. \text{ valid-SDG-node } n'' \land \text{parent-node } n'' \in \text{set } (\text{source-nodes } (a'\#as))
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\rightarrow V \notin \text{Def SDG } n'' \text{ by} (\text{fastforce simp: source-nodes-def})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{with } \langle \text{source-node } c' \rightarrow a' \# \ast \rightarrow \ast \text{ parent-node } n' \rangle & \\
\langle n' \in \text{HRB-slice } S \rangle (V \in \text{Use SDG } n')
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{show } V \in r乎 S (\text{CFG-node } (\text{source-node } c')) & \\
\text{by} (\text{fastforce intro: r乎I})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{qed}
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{show } \%\text{thesis}
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{proof}(\text{cases source-node } a \in [\text{HRB-slice } S]_{\text{CFG}})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{case } \text{True}
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{from } \langle \text{valid-edge } c' \rangle \langle \text{a } \in \text{get-return-edges } c' \rangle & \\
\text{have } \text{get-proc } (\text{target-node } c') = \text{get-proc } (\text{source-node } a) & \\
\text{by } \sim\text{drule intra-proc-additional-edge}, \text{auto dest: get-proc-intra simp: intra-kind-def})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{moreover}
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{from } \langle \text{valid-edge } c' \rangle \langle \text{kind } c' = Qx:x \leftarrow pxfsx \rangle & \\
\text{have } \text{get-proc } (\text{target-node } c') = px & \text{by} (\text{rule get-proc-call})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{moreover}
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{from } \langle \text{valid-edge } a \rangle \langle \text{kind } a = Q \leftarrow pf \rangle & \\
\text{have } \text{get-proc } (\text{source-node } a) = p & \text{by} (\text{rule get-proc-return})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{ultimately have } \langle \text{simp: } px = p \text{ by } simp \rangle
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{from } \langle \text{valid-edge } c' \rangle \langle \text{kind } c' = Qx:x \leftarrow pxfsx \rangle & \\
\text{obtain } \text{ins } \text{outs where } (p, \text{ins, outs}) \in \text{set proc}s & \\
\text{by} (\text{fastforce dest!: callee-in-procs})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{with } \langle \text{source-node } a \in [\text{HRB-slice } S]_{\text{CFG}} \rangle
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\langle \text{valid-edge } a \rangle \langle \text{kind } a = Q \leftarrow pf \rangle & \\
\text{have } \text{slice-kind: slice-kind } S a = & \\
Q \leftarrow p (\text{cf } c' \#, \text{rspp } (\text{target-node } a) (\text{HRB-slice } S) \text{ outs } cf \# cf) & \\
\text{by} (\text{rule slice-kind-Return-in-slice})
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{with } \langle \text{s }\in \text{ cf }\# \text{ cfs}\# \text{ cfs}\rangle \langle \text{s'} \in \text{ cf'}\# \text{ cfs}\# \text{ cfs}\rangle & \\
\text{have } \text{s:transfer } (\text{slice-kind } S a) s = & \\
(\text{rspp } (\text{target-node } a) (\text{HRB-slice } S) \text{ outs } (\text{fst cfs}) (\text{fst cf}), & \\
\text{snd cfs})\# \text{ cfs} & \\
\text{and } \text{s':transfer } (\text{slice-kind } S a) s' = & \\
(\text{rspp } (\text{target-node } a) (\text{HRB-slice } S) \text{ outs } (\text{fst cfs'}) (\text{fst cf'}), & \\
\text{snd cfs'})\# \text{ cfs'} & \\
\text{by } \text{simp-all}
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \begin{align*}
\text{with } \text{r乎1 have } \text{r乎1} \forall i < \text{length } cs'. & \\
\forall V \in r乎 S (\text{CFG-node } (\text{source-node } (\text{cs'} ! i)))
\end{align*}
\end{enumerate}
\[
\text{fst} \left(\langle\text{transfer} \ (\text{slice-kind} \ S \ a) \ s\rangle ! \text{Suc} \ i\right) V = \\
\text{fst} \left(\langle\text{transfer} \ (\text{slice-kind} \ S \ a) \ s'\rangle ! \text{Suc} \ i\right) V
\]
by \text{fastforce}

\text{from} \text{slice-kind} \ \forall \ i < \text{Suc} \ (\text{length} \ cs), \ \text{snd} \ (s!i) = \text{snd} \ (s'!i) \langle cs = c' \#
\]
\text{have} \ \langle s = cf#cfx#cf$_1$s \rangle \langle s' = cf'#cfx'#cf$_2$s' \rangle

\text{apply} \ \text{auto} \ \text{apply}(\text{case-tac} \ i) \ \text{apply} \ \text{auto}
by(\text{erule-tac} \ x = \text{Suc} \ (\text{Suc nat}) \ \text{in} \ \text{allE}) \ \text{auto}

\text{have} \ \forall \ V \in rv \ S \ (\text{CFG-node} \ (\text{targetnode a})), \\
\text{rspp} \ (\text{targetnode a}) \ (\text{HRB-slice} \ S) \ \text{outs}
(fst cfz) (fst cf) \ V = \\
\text{rspp} \ (\text{targetnode a}) \ (\text{HRB-slice} \ S) \ \text{outs}
(fst cfz') (fst cf') \ V

\text{proof}(\text{cases} \ V \in \ \text{set} \ (\text{ParamDefs} \ (\text{targetnode a})))
\text{case} \ \text{True}
then \ \text{obtain} \ i \ \text{where} \ i < \text{length} \ (\text{ParamDefs} \ (\text{targetnode a}))
and \ (\text{ParamDefs} \ (\text{targetnode a}))[i] = V
by(\text{fastforce} \ \text{simp:in-set-cone nth})
\text{from} \ \langle\text{valid-edge} \ a \ \langle\text{kind} \ a = \text{Q} \leftarrow p f \rangle \ \langle(p, \text{ins, outs}) \in \text{set procs} \rangle
\text{have} \ \text{length} (\text{ParamDefs} \ (\text{targetnode a})) = \text{length} \ \text{outs}
by(\text{fastforce} \ \text{intro:ParamDefs-return-target-length})
\text{show} \ \?\text{thesis}
\text{proof}(\text{cases} \ \text{Actual-out}(\text{targetnode a, i}) \in \text{HRB-slice} \ S)
\text{case} \ \text{True}
with \ i < \text{length} \ (\text{ParamDefs} \ (\text{targetnode a})), \ \langle\text{valid-edge} \ a \ \langle\text{length} (\text{ParamDefs} \ (\text{targetnode a})) = \text{length} \ \text{outs} \rangle \ \langle\text{ParamDefs} \ (\text{targetnode a})[i] = V\rangle \ \langle\text{then} \ \text{sym} \rangle
\text{have} \ \text{rspp-eq} : \text{rspp} \ (\text{targetnode a}) \\
(\text{HRB-slice} \ S) \ \text{outs} (fst cfz) (fst cf) \ V = \\
(fst cf)(\text{outs}i)
\text{rspp} \ (\text{targetnode a}) \\
(\text{HRB-slice} \ S) \ \text{outs} (fst cfz') (fst cf') \ V = \\
(fst cf')(\text{outs}i)
by(\text{auto} \ \text{intro:rspp-Actual-out-in-slice})
\text{from} \ \langle\text{valid-edge} \ a \ \langle\text{kind} \ a = \text{Q} \leftarrow p f \rangle \ \langle(p, \text{ins, outs}) \in \text{set procs} \rangle
\text{have} \ \forall \ V \in \ \text{set} \ \text{outs}, \ V \in \ \text{Use} \ (\text{source node} a) \ \text{by} \ \text{fastforce} \ \text{dest:outs-in-Use}
\text{have} \ \forall \ V \in \ \text{Use} \ (\text{source node} a), \ V \in rv \ S \ (\text{CFG-node} \ m)
\text{proof}
fix \ V \ \text{assume} \ V \in \ \text{Use} \ (\text{source node} a)
\text{from} \ (\text{valid-edge} \ a \ \langle\text{source node} a = m \rangle)
have parent-node (CFG-node m) −→,* parent-node (CFG-node m) by (fastforce intro:empty-path simp:intra-path-def)
with (sourcenode a ∈ HRB-slice S CFG)
  V ∈ Use (sourcenode a) (sourcenode a = m) (valid-edge a)
show V ∈ rv S (CFG-node m)
  by −(rule refl,
        auto intro!:CFG-Use-SDG-Use simp:SDG-to-CFG-set-def sourcenodes-def)
qed
with ∀ V ∈ set outs. V ∈ Use (sourcenode a)
have ∃ V ∈ set outs. V ∈ rv S (CFG-node m) by simp
with ∀ V ∈ rv S (CFG-node m). state-val s V = state-val s′ V
  (s = cf′#cfx#cfs) (s′ = cf′#cfx′#cfs)
have ∀ V ∈ set outs. (fst cf) V = (fst cf′) V by simp
with (i < length (ParamDefs (targetnode a)))
  (length(ParamDefs (targetnode a)) = length outs)
have (fst cf)(outs!i) = (fst cf′)(outs!i) by fastforce
with rspp-eq show ?thesis by simp
next
case False
with (i < length (ParamDefs (targetnode a))) (valid-edge a)
  (length(ParamDefs (targetnode a)) = length outs)
  ((ParamDefs (targetnode a))!i = V;[THEN sym]
have rspp-eq:(rspp (targetnode a)
  (HRB-slice S) outs (fst cfz) (fst cf)) V =
  (fst cfz)((ParamDefs (targetnode a))!i)
  (rspp (targetnode a)
  (HRB-slice S) outs (fst cfz′) (fst cf′)) V =
  (fst cfz′)((ParamDefs (targetnode a))!i)
  by (auto intro!:rspp-Actual-out-notin-slice)
from ∀ V ∈ rv S (CFG-node (sourcenode c′)).
  (fst cfz) V = (fst cfz′) V
  (∀ V ∈ rv S (CFG-node (targetnode a))).
  V ∈ rv S (CFG-node (sourcenode c′))
  ((ParamDefs (targetnode a))!i = V;[THEN sym]
have (fst cfz) (ParamDefs (targetnode a) ! i) =
  (fst cfz′) (ParamDefs (targetnode a) ! i) by fastforce
with rspp-eq show ?thesis by fastforce
qed
next
case False
with (V ∈ rv S (CFG-node (sourcenode c′)).
  (fst cfz) V = (fst cfz′) V
  (∀ V ∈ rv S (CFG-node (targetnode a))).
  V ∈ rv S (CFG-node (sourcenode c′))
show ?thesis by (fastforce simp:rspp-def map-merge-def)
qed
with \(sz \cdot sz'\)

have \(rv \cdot \forall V \in rv S \ (CFG\text{-node } (targetnode a))\).
  state-val \((\text{transfer } (\text{slice-kind } S a) s)\) \(V = \)
  state-val \((\text{transfer } (\text{slice-kind } S a) s')\) \(V \)
  by fastforce
from \((\text{preds } (\text{slice-kinds } S (a \# ax)) s)\)
have \(\text{preds } (\text{slice-kinds } S as)\)
  \((\text{transfer } (\text{slice-kind } S a) s)\)
  \(\text{by}(\text{simp add: slice-kinds-def})\)
moreover
from \((\text{preds } (\text{slice-kinds } S as') s') \langle \text{as'} = ax#ax \rangle\)
have \(\text{preds } (\text{slice-kinds } S ax)\)
  \((\text{transfer } (\text{slice-kind } S a) s')\)
  \(\text{by}(\text{simp add: slice-kinds-def})\)
moreover
from \((\text{length } s = \text{Suc } (\text{length } cs)) \langle cs = c' \# cs' \rangle \text{ sz} \)
have \(\text{length } (\text{transfer } (\text{slice-kind } S a) s) = \text{Suc } (\text{length } cs')\)
  \(\text{by}(\text{simp, simp add: s = cf # cf # cf s})\)
moreover
from \((\text{length } s' = \text{Suc } (\text{length } cs)) \langle cs = c' \# cs' \rangle \text{ sz'} \)
have \(\text{length } (\text{transfer } (\text{slice-kind } S a) s') = \text{Suc } (\text{length } cs')\)
  \(\text{by}(\text{simp, simp add: s' = cf'#cf#cf s'})\)
moreover
from \(HF[\text{OF } \langle \text{upd-cs } cs' \rangle \text{ as } = [] \langle \text{same-level-path-aux } cs' \text{ ax} \rangle\)
  \(\forall c \in \text{set } cs'. \text{ valid-edge } c \text{ (targetnode } a \rightarrow \ast \ (\text{-Low})): \)
  \(\text{targetnode } a \rightarrow \ast \ (\text{-Low}): \text{ res }' \text{ snds } rv' \text{ calculation} \langle \text{as'} = ax#ax \rangle\)
show \(\text{thesis } \text{by}(\text{simp add: slice-kinds-def})\)

next

case False
from this \(\langle \text{kind } a = Q \text{ cf} \rangle\)
have \(\text{slice-kind } s = \text{Suc } (\text{slice-kind } S a)\)
  \(a = (\lambda cf. \text{ True}) \mapsto_p (\lambda cf'. cf')\)
  \(\text{by}(\text{rule slice-kind-Return})\)
with \(s = \text{cf # cf # cf s}\)
have \(\text{snds } (\text{slice-kind } S a) s = \text{cf # cf # cf s}\)
  \((\text{transfer } (\text{slice-kind } S a) s') = \text{cf # cf # cf s'})\)
  \(\text{by simp-all}\)
from \(\langle \text{slice-kind } \forall i < \text{Suc } (\text{length } cs). \text{ snd } (s ! i) = \text{snd } (s' ! i) \rangle\)
have \(\langle cs = c' \# cs', s = \text{cf # cf # cf s}, (s' = \text{cf # cf # cf s'})\rangle\)
  \(\text{by simp-all}\)

next

from \(\forall V \in rv S \ (CFG\text{-node } (source-node cs' ! i))\)
  \(\text{fst } ((\text{transfer } (\text{slice-kind } S a) s) \in \text{Suc } i) \text{ V} = \)
  \(\text{fst } ((\text{transfer } (\text{slice-kind } S a) s') \in \text{Suc } i) \text{ V} \)
  \(\text{by fastforce}\)
from \(\forall V \in rv S \ (CFG\text{-node } (targetnode a))\)
  \(V \in rv S \ (CFG\text{-node } (source-node c'))\)
\(\forall V \in rv S \ (CFG\text{-node } (source-node c'))\).
\[(\text{fst } cfx) \ V = (\text{fst } cfx') \ V\]

**have** \( \forall V \in \text{rv} \ S \ \text{(CFG-node (targetnode a)).} \\
\text{state-val} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s) \ V = \text{state-val} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s') \ V \ \text{by simp} \\
\text{from} \langle \text{preds} \ (\text{slice-kinds} \ S \ (a \ # \ as)) \ s \rangle \\
\text{have} \ \langle \text{preds} \ (\text{slice-kinds} \ S \ as) \rangle \\
\text{(transfer} \ (\text{slice-kind} \ S \ a) \ s) \ \\
\text{by(simp add:slice-kinds-def)} \\
\text{moreover} \\
\text{from} \langle \text{preds} \ (\text{slice-kinds} \ S \ as') \ s' \rangle \ \langle \text{as'} = \ ax\#axs \rangle \\
\text{have} \ \langle \text{preds} \ (\text{slice-kinds} \ S \ ax) \rangle \\
\text{(transfer} \ (\text{slice-kind} \ S \ a) \ s') \ \\
\text{by(simp add:slice-kinds-def)} \\
\text{moreover} \\
\text{from} \langle \text{length} \ s = \text{Suc} \ (\text{length} \ cs) \rangle \ \langle cs = c' \# \ cs' \rangle \\
\text{have} \ \langle \text{length} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s) = \text{Suc} \ (\text{length} \ cs') \rangle \\
\text{by(simp simp add:s = cf\#cfx\#cfs)} \\
\text{moreover} \\
\text{from} \langle \text{length} \ s' = \text{Suc} \ (\text{length} \ cs) \rangle \ \langle cs = c' \# \ cs' \rangle \\
\text{have} \ \langle \text{length} \ (\text{transfer} \ (\text{slice-kind} \ S \ a) \ s') = \text{Suc} \ (\text{length} \ cs') \rangle \\
\text{by(simp simp add:s' = cf'\#cfx'\#cfs')} \\
\text{moreover} \\
\text{from} \langle \text{IH} \OF \langle \text{upd-cs cs'} \ as = [] \rangle \ (\text{same-level-path-aux cs'} \ asx) \
\forall c \in \text{set} \ cs'. \ \text{valid-edge} \ c \ \langle \text{targetnode} \ a \ - as - \rightarrow* \ (-\text{Low-}) \rangle \\
\langle \text{targetnode} \ a \ - ax - \rightarrow* \ (-\text{Low-}) \rangle \ \text{res'} \ \text{snfs rv'} \ \text{calculation} \ \langle \text{as'} = \ ax\#axs \rangle \rangle \\
\text{show} \ \langle \text{thesis by(simp add:slice-kinds-def)} \rangle \\
\text{qed} \\
\text{qed} \\
\text{qed}

**lemma** \( \text{rv-Low-Use-Low:} \\
\text{assumes} \ m - \text{as} - \rightarrow* \ (-\text{Low-}) \ \text{and} \ m - \text{as'} - \rightarrow* \ (-\text{Low-}) \ \text{and} \ \text{get-proc} \ m = \text{Main} \)

\text{and } \forall V \in \text{rv} \ S \ \text{(CFG-node m).} \ \text{cf} \ V = \text{cf'} \ V \\
\text{and} \ \langle \text{preds} \ (\text{slice-kinds} \ S \ as) \ [\langle \text{cf,undefined} \rangle] \rangle \\
\text{and} \ \langle \text{preds} \ (\text{slice-kinds} \ S \ as') \ [\langle \text{cf',undefined} \rangle] \rangle \\
\text{and} \ \langle \text{CFG-node} \ (-\text{Low-}) \ \in \ S \rangle \\
\text{shows} \ \forall V \in \text{Use} \ (-\text{Low-}). \\
\text{state-val} \ (\text{transfers(slice-kinds} \ S \ as) \ [\langle \text{cf,undefined} \rangle]) \ V = \\
\text{state-val} \ (\text{transfers(slice-kinds} \ S \ as') \ [\langle \text{cf',undefined} \rangle]) \ V \\
\text{proof(cases as)} \\
\text{case} \ \text{Nil} \\
\text{with} \langle m - \text{as} - \rightarrow \ (-\text{Low-}) \rangle \ \text{have} \ \langle \text{valid-node} \ m \ \text{and} \ m = (-\text{Low-}) \rangle \\
\text{by(auto intro:path-valid-node simp:vp-def)}
\{ \\
\text{fix} \ V \ \text{assume} \ V \in \text{Use} \ (-\text{Low-}) \\
\text{moreover} \\
\text{from} \langle \text{valid-node} \ m \rangle \langle m = (-\text{Low-}) \rangle \ \text{have} \ (-\text{Low-}) - [] - \rightarrow _* \ (-\text{Low-}) \\
\text{by(fastforce intro:empty-path simp:intra-path-def)}
\}

23
moreover
from (valid-node m) (m = (Low-)) (CFG-node (Low-) ∈ S)
have CFG-node (Low-) ∈ HRB-slice S
  by (fastforce intro!:HRB-slice-refl)
ultimately have V ∈ rv S (CFG-node m) using (m = (Low-))
    by (auto intro!:ril CFG-Use-SDG-Use simp:sourcenodes-def)

hence ∀ V ∈ Use (Low-). V ∈ rv S (CFG-node m) by simp

show ?thesis
proof (cases L = {})
  case True with UseLow show ?thesis by simp

next
  case False
from (m − as'−→^* (Low-)) have m − as'−→^* (Low-) by (simp add:wp-def)
from (m − as'−→^* (Low-)) (m = (Low-)) have as' = []

proof (induct m as' m'≡ (Low-) rule: path-induct)
  case (Cons-path m'' as a m)
  from (valid-edge a) (sourcenode a = m) (m = (Low-))
  have targetnode a = (Exit-) by (rule Exit-successor-of-Low,simp+)
  with (targetnode a = m'' (m'' − as−→^* (Low-))
  have (Low-) = (Exit-) by (drule path-Exit-source,auto)
  with False have False by (drule Low-neq-Exit,simp)
  thus ?case by simp

qed simp

with Nil (∀ V ∈ rv S (CFG-node m). cf V = cf' V)
  (∀ V ∈ Use (Low-). V ∈ rv S (CFG-node m))
  show ?thesis by (fastforce simp:slice-kinds-def)

qed

next
  case (Cons ax asx)
  with (m − as'−→^* (Low-)) have sourcenode ax = m and valid-edge ax
    and targetnode ax − asx−→^* (Low-)
    by (auto elim:path-split-Cons simp:wp-def)
  show ?thesis
  proof (cases L = {})
    case True with UseLow show ?thesis by simp
  next
    case False
    show ?thesis
  proof (cases as')
    case Nil
      with (m − as'−→^* (Low-)) have m = (Low-) by (fastforce simp:wp-def)
  with (valid-edge ax) (sourcenode ax = m) have targetnode ax = (Exit-)
    by (rule Exit-successor-of-Low,simp+)
  from Low-source-Exit-edge obtain a' where valid-edge a'
    and sourcenode a' = (Low-) and targetnode a' = (Exit-)
    and kind a' = (λs. True) by blast
  from (valid-edge ax) (sourcenode ax = m) (m = (Low-))
    (targetnode ax = (Exit-)) (valid-edge a') (sourcenode a' = (Low-))
    (targetnode a' = (Exit-))
have \( ax = a' \) by (fastforce dest: edge-det)
with (kind \( a' = (\lambda s. \text{True}) \)) have kind \( ax = (\lambda s. \text{True}) \) by simp
with (targetnode \( ax = (-\text{Exit}) \)) have \( \text{targetnode } ax - \text{as} \rightarrow (-\text{Low}) \).

have \( (-\text{Low}) = (-\text{Exit}) \) by \( \neg (\text{drule-Exit-source, auto}) \)
with False have False by \( \neg (\text{drule Low-neq-Exit, simp}) \)
thus \( \text{?thesis by simp} \)

next
case \( (\text{Cons } ax' \text{ asx}') \)
from \( (m \rightarrow as \rightarrow as \rightarrow -\text{Low}) \) have valid-path-aux [] as and \( m \rightarrow as \rightarrow (-\text{Low}) \)
by (simp-all add: vp-def valid-path-def)
from this \( \langle as = ax' \# \text{asx}' \rangle \) (get-proc \( m = \text{Main} \))
have same-level-path-aux [] as \( \land \) upd-cs [] as = []
by \( \neg (\text{rule vpa-Main-slpa[of - } m (-\text{Low})), \) (fastforce intro!;get-proc-Low simp: valid-call-list-def+)

hence same-level-path-aux [] as and upd-cs [] as = [] by simp-all
from \( (m \rightarrow as \rightarrow as \rightarrow -\text{Low}) \) have valid-path-aux [] as' \( \land \) \( m \rightarrow as' \rightarrow -\text{Low} \)

by (simp-all add: vp-def valid-path-def)
from this \( \langle as' = ax' \# \text{asx}' \rangle \) (get-proc \( m = \text{Main} \))
have same-level-path-aux [] as' \( \land \) upd-cs [] as' = []
by \( \neg (\text{rule vpa-Main-slpa[of - } m (-\text{Low})), \) (fastforce intro!;get-proc-Low simp: valid-call-list-def+)

hence same-level-path-aux [] as' by simp
from \( \langle \text{same-level-path-aux [] as} \rangle \) (upd-cs [] as = [])
\( \langle \text{same-level-path-aux [] as'} \rangle \) (\( m \rightarrow as \rightarrow -\text{Low}) \) \( \land \) \( m \rightarrow as' \rightarrow -\text{Low} \)
\( \forall V \in \text{rv } S \) \( \langle \text{CFG-node } m \rangle \). \( \langle \text{cf = cf'} \rangle \) \( \langle \text{CFG-node } (-\text{Low}) \in S \rangle \)
\( \langle \text{preds (slice-kinds } S \text{ as }) \rangle \) \( \langle [\text{cf, undefined}] \rangle \)
\( \langle \text{preds (slice-kinds } S \text{ as'}) \rangle \) \( \langle [\text{cf', undefined}] \rangle \)

show \( \text{?thesis by } \neg (\text{erule slpa-rv-Low-use-Low, auto}) \)
qed
qed

lemma nonInterference-path-to-Low:
assumes \( [\text{cf}] \approx_L [\text{cf}'] \) \( \langle \text{-High} \rangle \) \( \notin \langle \text{HRB-slice } S \rangle C_{\text{CFG}} \)
and \( \langle \text{CFG-node } (-\text{Low}) \in S \rangle \)
and \( \langle \text{-Entry} \rightarrow as \rightarrow as \rightarrow -\text{Low} \rangle \) \( \langle \text{preds (kinds } as \rangle \rangle \) \( \langle [\text{cf, undefined}] \rangle \)
and \( \langle \text{-Entry} \rightarrow as' \rightarrow as' \rightarrow -\text{Low} \rangle \) \( \langle \text{preds (kinds } as') \rangle \) \( \langle [\text{cf', undefined}] \rangle \)
shows map fst (transfers (kinds } as \rangle \rangle \langle [\text{cf, undefined}] \rangle) \approx_L
map fst (transfers (kinds } as') \rangle \rangle \langle [\text{cf', undefined}] \rangle)

proof -
from \( \langle \text{-Entry} \rightarrow as \rightarrow as \rightarrow -\text{Low} \rangle \) \( \langle \text{preds (kinds } as \rangle \rangle \) \( \langle [\text{cf, undefined}] \rangle \)
\( \langle \text{CFG-node } (-\text{Low}) \in S \rangle \)
obtain \( \text{asx where } \langle \text{preds (slice-kinds } S \text{ asx} \rangle \rangle \) \( \langle [\text{cf, undefined}] \rangle \)
and \( \forall V \in \text{Use } (-\text{Low}) \).
state-val (transfers (slice-kinds } S \text{ asx} \rangle \rangle \langle [\text{cf, undefined}] \rangle) \rangle V =
state-val (transfers (kinds } as \rangle \rangle \langle [\text{cf, undefined}] \rangle) \rangle V
and slice-edges S [] as = slice-edges S [] asx

25
\[
\text{and transfers (kinds as') } [(\text{cf}', \text{undefined})] \neq []
\]
\[
\text{and (-Entry) } \rightarrow_{\ast}^{\ast} (\text{-Low-})
\]
\[
\text{by (erule fundamental-property-of-static-slicing)}
\]
\[
\text{from } (-\text{Entry}) \rightarrow_{\ast}^{\ast} (\text{-Low-}) \vdash \text{preds (kinds as') } [(\text{cf}', \text{undefined})]
\]
\[
\text{CFG-node } (-\text{Low-}) \in S
\]
\[
\text{obtain } \text{as}' \text{ where } \text{preds (slice-kinds S asx') } [(\text{cf}', \text{undefined})]
\]
\[
\text{and } \forall V \in \text{Use } (-\text{Low-}).
\]
\[
\text{state-val transfers(slice-kinds S asx') } [(\text{cf}', \text{undefined})] \text{ } V =
\]
\[
\text{state-val transfers(kinds as') } [(\text{cf}', \text{undefined})] \text{ } V
\]
\[
\text{and slice-edges S } [] \text{ as}' =
\]
\[
\text{slice-edges S } [] \text{ asx'}
\]
\[
\text{and transfers (kinds as') } [(\text{cf}', \text{undefined})] \neq []
\]
\[
\text{and (-Entry) } \rightarrow_{\ast}^{\ast} (\text{-Low-})
\]
\[
\text{by (erule fundamental-property-of-static-slicing)}
\]
\[
\text{from } [(\text{cf}) \approx_{L} [(\text{cf}')] \vdash (\text{-High-}) \notin [\text{HRB-slice S}]_{\text{CFG}}
\]
\[
\text{have } \forall V \in \text{rv S } (\text{CFG-node } (-\text{Entry-}).) \text{ cf } V = \text{ cf' } V
\]
\[
\text{by (fastforce dest:lowEquivalence-relevant-nodes-Entry)}
\]
\[
\text{with } (-\text{Entry}) \rightarrow_{\ast}^{\ast} (-\text{-Low-}) \vdash (-\text{Entry}) \rightarrow_{\ast}^{\ast} (-\text{-Low-})
\]
\[
\vdash \text{preds (slice-kinds S asx') } [(\text{cf}', \text{undefined})]
\]
\[
\text{and slice-edges S } [] \text{ asx'}
\]
\[
\text{have } \forall V \in \text{Use } (-\text{Low-}).
\]
\[
\text{state-val transfers(slice-kinds S asx) } [(\text{cf, undefined})] \text{ } V =
\]
\[
\text{state-val transfers(slice-kinds S asx') } [(\text{cf, undefined})] \text{ } V
\]
\[
\text{by (-rule rv-Low-Use-Low, auto intro:get.proc-Entry)}
\]
\[
\text{with } \forall V \in \text{Use } (-\text{Low-}).
\]
\[
\text{state-val transfers(slice-kinds S asx) } [(\text{cf, undefined})] \text{ } V =
\]
\[
\text{state-val transfers(kinds as') } [(\text{cf, undefined})] \text{ } V
\]
\[
\text{and transfers (kinds as') } [(\text{cf, undefined})] \neq []
\]
\[
\text{show } ?\text{thesis by (fastforce simp:lowEquivalence-def UseLow neq-Nil-conv) qed}
\]

\text{theorem nonInterference-path:}
\text{assumes } [(\text{cf})] \approx_{L} [(\text{cf}')] \text{ and } (\text{-High-}) \notin [\text{HRB-slice S}]_{\text{CFG}}
\text{and CFG-node } (-\text{Low-}) \in S
\text{and (-Entry) } \rightarrow_{\ast}^{\ast} (-\text{-Exit-}) \text{ and preds (kinds as') } [(\text{cf}', \text{undefined})]
\text{and (-Entry) } \rightarrow_{\ast}^{\ast} (-\text{-Exit-}) \text{ and preds (kinds as') } [(\text{cf}', \text{undefined})]
\text{shows } \text{map } \text{fst } (\text{transfers (kinds as') } [(\text{cf, undefined})]) \approx_{L}
\text{map } \text{fst } (\text{transfers (kinds as') } [(\text{cf, undefined})])
\]
\text{proof –}
\text{from } (-\text{Entry}) \rightarrow_{\ast}^{\ast} (-\text{-Exit-}) \text{ obtain x xs where as } = x \# \text{xs}
\text{and (-Entry) } = \text{ sourcenode x and valid-edge x}
\text{and targetnode x - xs } \rightarrow_{\ast}^{\ast} (-\text{-Exit-})
\text{apply(cases as = [])}
\text{apply(clarsimp simp:up-def, drule empty-path-nodes, drule Entry-noteq-Exit, simp)}
by (fastforce elim: path-split-Cons simp: vp-def)
from (¬Entry) − as → √* (¬Exit) have valid-path as by (simp add: vp-def)
from (valid-edge x) have valid-node (targetnode x) by simp
hence inner-node (targetnode x)

proof (cases rule: valid-node-cases)
  case Entry
  with (valid-edge x) have False by (rule Entry-target)
  thus ?thesis by simp
  next
  case Exit
  with (targetnode x − x ∗ xs → ∗ (¬Exit)) have xs = [] by 
  from (¬Entry nodes ⟨Exit-source, auto⟩)
  from Entry-Exit-edge obtain z where valid-edge z
  and sourcenode z = (¬Entry) and targetnode z = (¬Exit)
  and kind z = (λs. False) _ by blast
  from (valid-edge x) (valid-edge z) (¬Entry) = sourcenode x
  (sourcenode z = (¬Entry)) Exit (targetnode z = (¬Exit))
  have x = z by (fastforce intro: edge-det)
  with (preds (kinds as) [(cf, undefined)] (as = x # xs) (xs = [])
  by simp add: kinds-def)
  have False by (simp add: kinds-def)
  thus ?thesis by simp

qed simp

with (targetnode x − x ∗ xs → ∗ (¬Exit)) obtain x' xs' where xs = xs' ∗ [x]
  and targetnode x − x ∗ xs' → ∗ (¬Low) and kind x' = (λs. True)
  by (fastforce elim: Entry-path-Low-path)

with (¬Entry) = sourcenode x (valid-edge x)
have (¬Entry) − x # xs → ∗ (¬Low) by (fastforce intro: Cons-path)
from (valid-path as) (as = x # xs) (xs = xs' ∗ [x])
have valid-path (x # xs') by (simp add: valid-path-def del: valid-path-aux.simps)
  (rule valid-path-aux-split simp)
with (¬Entry) − x # xs' → ∗ (¬Low) have (¬Entry) − x # xs' → ∗ (¬Low)
  by (simp add: vp-def)
from (as = x # xs) (xs = xs' ∗ [x]) have as = (x # xs') ∗ [x]
  by simp
with (preds (kinds as) [(cf, undefined)])
  have preds (kinds (x # xs')) [(cf, undefined)]
  by (simp add: kinds-def preds-split)
from (¬Entry) − as → ∗ (¬Exit) obtain y ys where as' = y # ys
  and (¬Entry) = sourcenode y and valid-edge y
  and targetnode y − ys → ∗ (¬Exit)
  apply (cases as' = [])
  apply (clarsimp simp: vp-def, drule empty-path-nodes, drule Entry-noteq-Exit, simp)
  by (fastforce elim: path-split-Cons simp: vp-def)
from (¬Entry) − as → ∗ (¬Exit) have valid-path as' by (simp add: vp-def)
from (valid-edge y) have valid-node (targetnode y) by simp
hence inner-node (targetnode y)

proof (cases rule: valid-node-cases)
  case Entry

27
with \((\text{valid-edge} \ y)\) have \(\text{False by (rule Entry-target)}\)

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

next

case \(\text{Exit}\)

with \(\langle \text{targetnode \ y} = \text{ys} \mapsto \cdot \text{(\text{-Entry\-})} \rangle \) have \(\text{ys } = \cdot \)

by \((-\text{drule path-Exit-source.auto})\)

from \((\text{Entry-Exit-edge})\) obtain \(\text{z where valid-edge z}\)

and \(\text{source-node z } = \cdot \text{(\text{-Entry\-})} \) and \(\text{target-node z } = \cdot \text{(\text{-Exit\-})}\)

and \(\text{kind \ z } = \langle \lambda s. \text{False} \rangle\)

by blast

from \((\text{valid-edge} \ y)\) \((\text{valid-edge} \ z)\) \((\text{-Entry\-}) = \text{source-node} \ y\)

\((\text{source-node} \ z = \cdot \text{(\text{-Entry\-})})\) \(\text{Exit} \cdot \text{(\text{-target-node} \ z = \cdot \text{(\text{-Exit\-})})}\)

have \(\text{y } = \text{z by (fastforce intro:edge-det)}\)

with \((\text{preds \ (kinds as')}\) \((\text{[(cf', undefined):]}\) \(\langle as' = y\#ys'\rangle \) \(\text{ys } = \cdot \)

\(\langle \text{kind} \ z = \langle \lambda s. \text{False} \rangle\)\)

have \(\text{False by (simp add: kinds-def):}\)

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

qed

simp

with \(\langle \text{targetnode} \ y = \text{ys} \mapsto \cdot \text{(\text{-Entry\-})} \rangle\) obtain \(\text{ys' \ ys'' where} \text{ys } = \text{ys}'@[[\text{y}']\)

and \(\text{target-node} \ y = \text{ys} \mapsto \cdot \text{(\text{-Low\-})} \) and \(\text{kind} \ y' = \langle \lambda s. \text{True} \rangle\)

by (fastforce elim: \text{Exit-path-Low-path})

with \((\cdot \text{Entry\-}) = \text{source-node} \ y\) \(\langle \text{valid-edge} \ y\)\)

have \(\langle \cdot \text{-Entry\-} \rangle - y\#y s' \mapsto \cdot \text{(\text{-Low\-})} \) by (fastforce intro: Cons-path)

from \((\text{valid-path as}') \langle as' = y\#ys' \rangle \) \(\langle \text{ys } = \text{ys}'@[[\text{y}']\)

have \(\langle \text{valid-path} \ (y\#ys')\)

by (simp add: \text{valid-path-def del: valid-path-aux,simps})

\(\langle \text{rule valid-path-aux-split,simp}\)

with \((\cdot \text{-Entry\-}) - y\#y s' \mapsto \cdot \text{(\text{-Low\-})}\)

have \(\langle \cdot \text{-Entry\-} \rangle - y\#y s' \mapsto \cdot \text{(\text{-Low\-})}\) by (simp add: \text{vp-def})

from \(\langle \text{as' } = y\#ys' \rangle \) \(\langle \text{ys } = \text{ys}'@[[\text{y}']\)

have \(\langle \text{as' } = \langle \text{y\#ys'} @[[\text{y}']\) by simp

with \(\langle \text{preds \ (kinds as')} \langle [\text{cf', undefined]}\)\)

have \(\langle \text{preds \ (kinds} (y\#ys')) \langle [\text{cf', undefined]}\)

by (simp add: \text{kinds-def preds-split})

from \(\langle [\text{cf}] \approx_L [\text{cf}'] \) \(\langle \cdot \text{High\-} \in [\text{HRB\-slice} \ S]\) \(\text{CFG} \cdot \langle \text{CFG\-node} \ (\text{-Low\-}) \in \ S\)

\(\langle \cdot \text{-Entry\-} \rangle - x\#x s' \mapsto \cdot \text{(\text{-Low\-})}\)

\(\langle \text{preds \ (kinds} (x\#x s') \langle [\text{cf, undefined]}\)\)

\(\langle \cdot \text{-Entry\-} \rangle - y\#y s' \mapsto \cdot \text{(\text{-Low\-})}\)

\(\langle \text{preds \ (kinds} (y\#y s') \langle [\text{cf', undefined]}\)\)

have \(\langle \text{map \ fst} \ (\text{transfers \ (kinds} (x\#x s') \langle [\text{cf, undefined]}\)\)

\(\langle \text{map \ fst} \ (\text{transfers \ (kinds} (y\#y s') \langle [\text{cf', undefined]}\)\)

by (rule nonInterference-path-to-Low)

with \(\langle \text{as } = x\#x s \rangle \) \(\langle \text{xs } = x@[[\text{x}']\)

\(\langle \text{kind} \ x' = \langle \lambda s. \text{True} \rangle\)

\(\langle \text{as' } = y\#y s' \rangle \) \(\langle \text{ys } = y@[[\text{y}']\)

\(\langle \text{kind} \ y' = \langle \lambda s. \text{True} \rangle\)

show \(\exists \text{thesis}\)

apply (cases transfers \(\langle \text{map \ kind} \ xs' \rangle \) \(\langle \text{transfer \ (kind} x) \langle [\text{cf, undefined]}\)\)

apply (auto simp add: kinds-def transfers-split)

by (cases transfers \(\langle \text{map \ kind} \ ys' \rangle \) \(\langle \text{transfer \ (kind} y) \langle [\text{cf', undefined]}\)\),

(auto simp add: kinds-def transfers-split))

qed

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

**locale** NonInterferenceInter =

NonInterferenceInterGraph sourcenode targetnode kind valid-edge Entry

get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses

\( H \) \ L \ High \ Low +

SemanticsProperty sourcenode targetnode kind valid-edge Entry get-proc

get-return-edges procs Main Exit Def Use ParamDefs ParamUses sem identifies

for sourcenode :: 'edge \Rightarrow \text{node} \text{ and targetnode :: 'edge \Rightarrow \text{node}}

and kind :: 'edge \Rightarrow \text{bool}

and Entry :: 'node ('('Entry')') \text{ and get-proc :: 'node \Rightarrow \text{pname}}

and get-return-edges :: 'edge \Rightarrow \text{edge set}

and procs :: ('pname \times \text{var list} \times \text{var list}) \text{ list and Main :: \text{pname}}

and Exit::'node ('('Exit')')

and Def :: 'node \Rightarrow \text{var set and Use :: 'node \Rightarrow \text{var set}}

and ParamDefs :: 'node \Rightarrow \text{var list and ParamUses :: 'node \Rightarrow \text{var set list}}

and sem :: 'com \Rightarrow ('var \Rightarrow \text{val}) \text{ list \Rightarrow \text{com} \Rightarrow ('var \Rightarrow \text{val}) \text{ list \Rightarrow bool}

((\{1,2\}) \Rightarrow (1,0,0,0) 81)

and identifies :: 'node \Rightarrow \text{com \Rightarrow bool (} \triangleq - [51,0] 80)

and \( H :: \text{var set and } L :: \text{var set}

and High :: 'node ('('High')') \text{ and Low :: 'node ('('Low')') +}

fixes final :: 'com \Rightarrow bool

assumes final-edge-Low: \[\text{\exists final c; n \triangleq c}\]

\[\Rightarrow \exists a. \text{valid-edge a \land sourcenode a = n \land targetnode a = (Low) \land kind a = \uparrow id}\]

**begin**

The following theorem needs the explicit edge from ('High') to n. An approach using a \textit{init} predicate for initial statements, being reachable from ('High') via a \((\lambda s. \text{True})\) 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 \([cf_1] \triangleq_L [cf_2] \text{ and ('High') } \notin [\text{HRB-slice} S] \text{ CFG}

and CFG-node (Low) \in S

and valid-edge a and sourcenode a = ('High') \text{ and targetnode a = n}

and kind a = (\lambda s. \text{True}), \text{ and } n \triangleq c \text{ and final c'}

and \((c,[cf_1]) \Rightarrow (c',s_1) \text{ and } (c,[cf_2]) \Rightarrow (c',s_2)

shows s_1 \triangleq_L s_2

proof –

from High-target-Entry-edge obtain ax where valid-edge ax

and sourcenode ax = (Entry-) \text{ and targetnode ax = ('High')}

and kind ax = (\lambda s. \text{True}) by blast
from \((n \triangleq c) \langle c, [cf_1] \rangle \Rightarrow \langle c', s_1 \rangle\)

obtain \(n_1 \triangleright as_1 \triangleright cfs_1\) where \(n \triangleright\triangleright as_1 \triangleright s_1 \triangleright n_1\) and \(n_1 \triangleq c'\)
    and \(\text{preds (kinds as_1) \{\{cf_1, \text{undefined}\}\}}\)
    and \(\text{transfers (kinds as_1) \{\{cf_1, \text{undefined}\}\}} = cfs_1\) and \(\text{map fst cfs}_1 = s_1\)
by (fastforce dest fundamental-property)

from \((n \triangleright as_1 \triangleright s_1 \triangleright n_1)\) (valid-edge \(a\) \& (source-node \(a = (-\text{High-})\)) \& (target-node \(a = n\))
    \& (kind \(a = (\lambda s. True)\))

have \((-\text{High-}) - a \# as_1 \rightarrow s_1 \triangleright n_1\) by (fastforce intro:Cons-path simp:vp-def valid-path-def)

from \((\text{final } c \triangleright n_1 \triangleq c)\)

obtain \(a_1\) where \(\text{valid-edge } a_1\) and \(\text{source-node } a_1 = n_1\)
    and \(\text{target-node } a_1 = (-\text{Low-})\) and \(\text{kind } a_1 = \uparrow \text{id}\)
by (fastforce dest:final-edge-Low)

hence \(n_1 \triangleright\triangleright [a_1] \Rightarrow (\text{Low-})\) by (fastforce intro:path-edge)

with \((-\text{High-}) - a \# as_1 \rightarrow s_1 \triangleright n_1\) have \((-\text{High-}) - (a \# as_1) \# ([a_1] \Rightarrow (\text{Low-})
    by (fastforce intro:path-Append simp:vp-def)

with \((\text{valid-edge } ax) \& (\text{source-node } ax = (-\text{Entry-}) \& (\text{target-node } ax = (-\text{High-}))

have \((-\text{Entry-}) - ax \# ((a \# as_1) \# ([a_1])) \Rightarrow (\text{Low-})\) by \((\text{rule Cons-path})\)

moreover

from \((-\text{High-}) - a \# as_1 \rightarrow s_1 \triangleright n_1\) have \((\text{valid-path-aux} []) \triangleright (a \# as_1)\)
    by (simp add: vp-def valid-path-def)

with \((\text{kind } a_1 = \uparrow \text{id})\) have \((\text{valid-path-aux} []) \triangleright ((a \# as_1) \# ([a_1]))\)
    by (fastforce intro:valid-path-aux-Append)

with \((\text{kind } ax = (\lambda s. True)\)) have \((\text{valid-path-aux} []) \triangleright ((ax \# ((a \# as_1) \# ([a_1])))\)
    by simp

ultimately have \((-\text{Entry-}) - ax \# ((a \# as_1) \# ([a_1])) \rightarrow s_1 \triangleright (\text{Low-})\)
    by (simp add: vp-def valid-path-def)

from \((\text{valid-edge } a) \triangleright \text{kind } a = (\lambda s. True)\) \& \(\text{source-node } a = (-\text{High-})\)
    \& \(\text{target-node } a = n\)

have \(\text{get-proc } n = \text{get-proc } (-\text{High-})\)
    by (fastforce dest: get-proc-intra simp: intra-kind-def)

with \(\text{get-proc-High}\) have \(\text{get-proc } n = \text{Main}\) by simp

from \((\text{valid-edge } a_1) \triangleright \text{source-node } a_1 = n_1\) \& \(\text{target-node } a_1 = (-\text{Low-})\) \& \(\text{kind } a_1 = \uparrow \text{id}\)

have \(\text{get-proc } n_1 = \text{get-proc } (-\text{Low-})\)
    by (fastforce dest: get-proc-intra simp: intra-kind-def)

with \(\text{get-proc-Low}\) have \(\text{get-proc } n_1 = \text{Main}\) by simp

from \((n \triangleright as_1 \rightarrow s_1 \triangleright n_1)\) have \(n \triangleright as_1 \rightarrow s_1 \triangleright n_1\)
    by (cases as_1)

    \(\langle \text{auto dest!}:\text{vpa-Main-slp intro!}:\text{get-proc } n_1 = \text{Main} \rangle \triangleright \langle \text{get-proc } n = \text{Main} \rangle\)
    simp: vp-def valid-path-def valid-call-list-def slp-def
    same-level-path-def simp del:valid-path-aux.sims)

    then obtain \(cfx r\) where \(cfx: \text{transfers (map kind } as_1) \{\{cf_1, \text{undefined}\}\} = \{\{cf_1, r\}\}
    \) by (fastforce elim: slp-callbackstack-length-equal simp: kinds-def)

from \((\text{kind } ax = (\lambda s. True)\) \& \(\text{kind } a = (\lambda s. True)\) \& \(\text{preds (kinds } as_1) \{\{cf_1, \text{undefined}\}\} \& \(\text{kind } a_1 = \uparrow \text{id}\) \& \(cfx\)

have \(\text{preds (kinds } ax \# ((a \# as_1) \# ([a_1]))) \{\{cf_1, \text{undefined}\}\}\)
    \(\langle \text{auto simp: kinds-def preds-split} \rangle\)

from \((n \triangleq c) \langle c, [cf_2] \rangle \Rightarrow \langle c', s_2 \rangle)\)
\begin{verbatim}
obtain \( n_2 \) \( as_2 \) \( cf_2 \) where \( n - as_2 \to \forall \ast \ n_2 \) and \( n_2 \triangleq c' \)
and \( \text{preds} (\text{kind} \ as_2) [\{\text{cf}_2, \text{undefined}\}] \)
and \( \text{transfers} (\text{kind} \ as_2) [\{\text{cf}_2, \text{undefined}\}] = \text{cf}_2 \) and \( \text{map \ fst} \ cf_2 = s_2 \)
by (fastforce dest: fundamental-property)

from \( n = as_2 \to \forall \ast \ n_2 \), \( \text{valid-edge} \ a \) \( \langle \text{sourcenode} \ a = (-\text{High-}) \rangle \) \( \langle \text{targetnode} \ a = n \rangle \)
(\( \langle \text{kind} \ a = (\lambda s. \text{True}) \rangle \)\( )

have \( (-\text{High-}) \to a \# as_2 \to \forall \ast \ n_2 \) by (fastforce intro: Cons-path simp: vp-def valid-path-def)
from \( \langle \text{final} \ c' \rangle \langle n_2 \triangleq c' \rangle \)
obtain \( a_2 \) where \( \text{valid-edge} \ a_2 \) and \( \text{sourcenode} \ a_2 = n_2 \)
and \( \langle \text{targetnode} \ a_2 = (-\text{Low-}) \rangle \) and \( \text{kind} \ a_2 = \uparrow \text{id} \) by (fastforce dest: final-edge-Low)

hence \( n_2 - [a_2] \to \ast \) \( (-\text{Low-}) \) by (fastforce intro: path-edge)

with \( \langle (-\text{High-}) \to a \# as_2 \to \forall \ast \ n_2 \rangle \) have \( \langle (-\text{High-}) \to (a \# as_2)@[a_2] \to \ast \rangle \) \( (-\text{Low-}) \) by (fastforce intro: path-Append simp: vp-def)

with \( \langle \text{valid-edge} \ ax \rangle \) \( \langle \text{sourcenode} \ ax = (-\text{Entry-}) \rangle \) \( \langle \text{targetnode} \ ax = (-\text{High-}) \rangle \)

have \( \langle (-\text{Entry-}) \to ax \# ([a \# as_2]@[a_2]) \to \ast \rangle \) \( (-\text{Low-}) \) by \( -(\text{rule Cons-path}) \)
moreover

from \( \langle (-\text{High-}) \to a \# as_2 \to \forall \ast \ n_2 \rangle \) have \( \text{valid-path-aux} \ [] \langle a \# as_2 \rangle \)
by (simp add: vp-def valid-path-def)

with \( \langle \text{kind} \ a_2 = \uparrow \text{id} \rangle \)
have \( \text{valid-path-aux} \ [] \langle ([a \# as_2]@[a_2]) \rangle \)
by (fastforce intro: valid-path-aux-Append)

with \( \langle \text{kind} \ ax = (\lambda s. \text{True}) \rangle \)
have \( \text{valid-path-aux} \ [] \langle (ax \# ([a \# as_2]@[a_2])) \rangle \)
by simp
ultimately have \( \langle (-\text{Entry-}) \to ax \# ([a \# as_2]@[a_2]) \to \forall \ast \rangle \) \( (-\text{Low-}) \)
by (simp add: vp-def valid-path-def)

from \( \langle \text{valid-edge} \ a \rangle \) \( \langle \text{kind} \ a = (\lambda s. \text{True}) \rangle \)
\( \langle \text{sourcenode} \ a = (-\text{High-}) \rangle \)
\( \langle \text{targetnode} \ a = n \rangle \)

have \( \text{get-proc} \ n = \text{get-proc} \ (-\text{High-}) \)
by (fastforce dest: get-proc-intra simp: intra-kind-def)

with \( \text{get-proc-High} \) have \( \text{get-proc} \ n = \text{Main} \) by simp

from \( \langle \text{valid-edge} \ a_2 \rangle \) \( \langle \text{sourcenode} \ a_2 = n_2 \rangle \)
\( \langle \text{targetnode} \ a_2 = (-\text{Low-}) \rangle \)
\( \langle \text{kind} \ a_2 = \uparrow \text{id} \rangle \)

have \( \text{get-proc} \ n_2 = \text{get-proc} \ (-\text{Low-}) \)
by (fastforce dest: get-proc-intra simp: intra-kind-def)

with \( \text{get-proc-Low} \) have \( \text{get-proc} \ n_2 = \text{Main} \) by simp

from \( \langle n - as_2 \to \forall \ast \ n_2 \rangle \) have \( n - as_2 \to \ast \ n_2 \)
by (cases as_2)

(auto dest: !pa-Main-slap intro: get-proc \( n_2 = \text{Main} \) \( \rightarrow \) get-proc \( n = \text{Main} \)
\[ \rightarrow \) simp: vp-def valid-path-def valid-call-list-def sli-def
same-level-path-def simp del: valid-path-aux.simps)

then obtain \( cfz', r' \)
where \( cfz' \) transfers \( \langle \text{map \ kind} \ as_2 \rangle \) \( \langle \{cfz, \text{undefined}\} \rangle \)
by (fastforce elim: sli-callstack-length-equal simp: kinds-def)

from \( \langle \text{kind} \ ax = (\lambda s. \text{True}) \rangle \)
\( \langle \text{kind} \ a = (\lambda s. \text{True}) \rangle \)
\( \langle \text{preds \ kind} \ as_2 \rangle \) \( \langle \text{kind} \ a_2 = \uparrow \text{id} \rangle \)
\( \text{cfz'} \)

have \( \text{preds \ kind} \ (ax \# ([a \# as_2]@[a_2])) \) \( \langle \text{cfz, undefined} \rangle \)
by (auto simp: kinds-def preds-split)

from \( \langle \text{cfz} \rangle \approx_L \langle \text{cfz} \rangle \) \( \rightarrow \) \( \langle \text{HNB-slice} \ S \rangle \)
\( \langle \text{CFG} \rangle \) \( \langle \text{CFG-node} \ (-\text{Low-}) \in S \rangle \)
\( \langle \text{(-Entry-)} \to ax \# ([a \# as_2]@[a_2]) \to \forall \ast \rangle \) \( (-\text{Low-}) \).
\end{verbatim}
have map fst (transfers (kinds (ax#((a#as1)@[a1]))) [[(cf1,undefined)]]) ≈L map fst (transfers (kinds (ax#((a#as2)@[a2]))) [[(cf2,undefined)]])

by (rule nonInterference-path-to-Low)
with \( \text{kind } ax = (\lambda s. \text{True}) \), \( \text{kind } a_1 = (\lambda s. \text{True}) \), \( \text{kind } a_2 = \uparrow \text{id} \), \( \text{kind } a_2 = \uparrow \text{id} \)

\( \text{transfers (kinds as1) } [[(cf1,\text{undefined})]] = cfs_1 \) \( \text{map fst } cfs_1 = s_1 \)
\( \text{transfers (kinds as2) } [[(cf2,\text{undefined})]] = cfs_2 \) \( \text{map fst } cfs_2 = s_2 \)

show \( \tilde{\text{thesis}} \) by (cases s_1) (cases s_2, (fastforce simp: kinds-def transfers-split) + ) +

qed

end

end

3 Framework Graph Lifting for Noninterference

theory LiftingInter
imports NonInterferenceInter
begin

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

3.1 Liftings

3.1.1 The datatypes

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

type-synonym ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge =
'node LDCFG-node × ((('var,'val,'ret,'pname) edge-kind) × 'node LDCFG-node

3.1.2 Lifting basic definitions using 'edge and 'node

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

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

| lve-Entry-edge: \[\text{ e } = (\text{ NewEntry}, (\lambda s. \text{ True}) \sqcup, \text{ Node } E)\]
  \[\Rightarrow \text{ lift-valid-edge valid-edge } \text{ src } \text{ trg } \text{ knd } E X e\]

| lve-Exit-edge: \[\text{ e } = (\text{ NewEntry}, (\lambda s. \text{ True}) \sqcup, \text{ NewExit})\]
  \[\Rightarrow \text{ lift-valid-edge valid-edge } \text{ src } \text{ trg } \text{ knd } E X e\]

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

lemma [simp]: \[\neg \text{ lift-valid-edge valid-edge } \text{ src } \text{ trg } \text{ knd } E X \text{ (Node E, et, Node X)}\]
by (auto elim: lift-valid-edge. cases)

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

inductive-set lift-get-return-edges :: ('edge ⇒ 'edge set) ⇒ ('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ ('var,'val,'ret,'pname) edge-kind)
  ⇒ ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge
⇒ ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge set
for get-return-edges :: 'edge ⇒ 'edge set and valid-edge :: 'edge ⇒ bool
and src::'edge ⇒ 'node and trg::'edge ⇒ 'node
and knd::'edge ⇒ ('var,'val,'ret,'pname) edge-kind
and e::('edge,'node,'var,'val,'ret,'pname) LDCFG-edge
where lift-get-return-edgesI:
  \[\text{ e } = (\text{ Node } (\text{ src } a), \text{ knd } a, \text{ Node } (\text{ trg } a)); \text{ valid-edge } a; a' \in \text{ get-return-edges } a; \]
  \[\text{ e' } = (\text{ Node } (\text{ src } a'), \text{ knd } a', \text{ Node } (\text{ trg } a'))\]
  \[\Rightarrow e' \in \text{ lift-get-return-edges get-return-edges valid-edge } \text{ src } \text{ trg } \text{ knd } e\]

3.1.3 Lifting the Def and Use sets

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

For Def::(\text{node} \Rightarrow \text{var set}) \text{ and } E::\text{node} \text{ and } X::\text{node} \\
and H::\text{var set} \text{ and } L::\text{var set}

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

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

Abbreviation lift-Def ::= (\text{node} \Rightarrow \text{var set}) \Rightarrow \text{node} \Rightarrow \text{node} \\
\text{var set} \Rightarrow \text{var set} \Rightarrow (\text{node LDCFG-node} \Rightarrow \text{var set})

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

Inductive-set lift-Use-set ::= (\text{node} \Rightarrow \text{var set}) \Rightarrow \text{node} \Rightarrow \text{node} \\
\text{var set} \Rightarrow \text{var set} \Rightarrow (\text{node LDCFG-node} \times \text{var}) \text{ set}

For Use::\text{node} \Rightarrow \text{var set} \text{ and } E::\text{node} \text{ and } X::\text{node} \\
and H::\text{var set} \text{ and } L::\text{var set}

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

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

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

Abbreviation lift-Use ::= (\text{node} \Rightarrow \text{var set}) \Rightarrow \text{node} \Rightarrow \text{node} \\
\text{var set} \Rightarrow \text{var set} \Rightarrow (\text{node LDCFG-node} \Rightarrow \text{var set})

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

Fun lift-ParamUses ::= (\text{node} \Rightarrow \text{var set list}) \Rightarrow \text{node LDCFG-node} \Rightarrow \text{var set list}

Where lift-ParamUses ParamUses (Node n) = ParamUses n
\[
| \text{lift-ParamUses ParamUses NewEntry} = []
| \text{lift-ParamUses ParamUses NewExit} = []
\]

Fun lift-ParamDefs ::= (\text{node} \Rightarrow \text{var list}) \Rightarrow \text{node LDCFG-node} \Rightarrow \text{var list}

Where Lift-ParamDefs ParamDefs (Node n) = ParamDefs n
\[
| \text{lift-ParamDefs ParamDefs NewEntry} = []
| \text{lift-ParamDefs ParamDefs NewExit} = []
\]
3.2 The lifting lemmas

3.2.1 Lifting the CFG locales

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

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

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

lemma lift-CFG:
  assumes wf: CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
  and pd: Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
  shows CFG src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-get-proc get-proc Main)
  (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
  procs Main
proof –
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(rule wf)
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
by(rule pd)
show ?thesis
proof
  fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and trg a = NewEntry
  thus False by(fastforce elim: lift-valid-edge_cases)
next
show lift-get-proc get-proc Main NewEntry = Main by simp
next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q:r↪p fs and src a = NewEntry
  thus False by(fastforce elim: lift-valid-edge_cases)
next
  fix a a'
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
  and src a = src a' and trg a = trg a'
  thus a = a'
proof (induct rule: lift-valid-edge.induct)
  case lve-edge thus False by (erule lift-valid-edge_cases, auto dest: edge-det)
qed (auto elim: lift-valid-edge_cases)

next
  fix a Q r f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and knd a = Q : r → Main
  thus False by (fastforce elim: lift-valid-edge_cases dest: Main-no-call-target)

next
  fix a Q’ f’
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and knd a = Q’ ← Main
  thus False by (fastforce elim: lift-valid-edge_cases dest: Main-no-return-source)

next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and knd a = Q : r → p fs
  thus ∃ ins outs. (p, ins, outs) ∈ set procs
      by (fastforce elim: lift-valid-edge_cases intro: callee-in-procs)

next
  fix a Q r p fs
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
      and knd a = Q : r → p fs
  then obtain ax where valid-edge ax and kind ax = Q : r → p fs
      and sourcenode ax ≠ Entry ∨ targetnode ax ≠ Exit
      and src a = Node (sourcenode ax) and trg a = Node (targetnode ax)
      by (fastforce elim: lift-valid-edge_cases)
  from valid-edge ax: (kind ax = Q : r → p fs); have all: ∀ a’. valid-edge a’ ∧ targetnode a’ = targetnode ax →
      (∃ Qx rz fsx. kind a’ = Qx : rz → p fsx)
      by (auto dest: call-edges-only)
{ fix a' 
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' 
  and trg a' = trg a 
  hence ∃ Qx rx fsx. knd a' = Qx:rx→pfx 
  proof (induct rule:lift-valid-edge.induct) 
  case (lve-edge ax' e) 
  note [simp] = ⟨e = (Node (sourcenode ax'), kind ax', Node (targetnode ax'))⟩ 
  from ⟨trg e = trg a⟩ ⟨trg a = Node (targetnode ax)⟩ 
  have targetnode ax' = targetnode ax by simp 
  with (valid-edge ax) all have ∃ Qx rx fsx. knd ax' = Qx:rx→pfx by blast 
  thus ?case by simp 
next 
  case (lve-Entry-edge e) 
  from ⟨e = (NewEntry, (λs. True) , Node Entry)⟩ ⟨trg e = trg a⟩ 
  ⟨trg a = Node (targetnode ax)⟩ 
  have targetnode ax = Entry by simp 
  with (valid-edge ax) have False by (rule Entry-target) 
  thus ?case by simp 
next 
  case (lve-Exit-edge e) 
  from ⟨e = (Node Exit, (λs. False) , NewExit)⟩ ⟨trg e = trg a⟩ 
  ⟨trg a = Node (targetnode ax)⟩ 
  have False by simp 
  thus ?case by simp 
next 
  case (lve-Entry-Exit-edge e) 
  from ⟨e = (NewEntry,(λs. False) ,NewExit)⟩ ⟨trg e = trg a⟩ 
  ⟨trg a = Node (targetnode ax)⟩ 
  have False by simp 
  thus ?case by simp 
qed } 
thus ∀ a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧ 
  trg a' = trg a → (∃ Qx rx fsx. knd a' = Qx:rx→pfx) by simp 
next 
fix a Q' p f' 
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a 
  and knd a = Q'←p f' 
then obtain ax where valid-edge ax and kind ax = Q'←p f' 
  and sourcenode ax ≠ Entry ∨ targetnode ax ≠ Exit 
and src a = Node (sourcenode ax) and trg a = Node (targetnode ax) 
by (fastforce elim:lift-valid-edge.cases) 
from (valid-edge ax) ⟨kind ax = Q'←p f'⟩ 
have all:∀ a'. valid-edge a' ∧ sourcenode a' = sourcenode ax → 
  (∃ Qx fx. kind a' = Qx:fx) by (auto dest:return-edges-only) 
{ fix a' 
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' 
  and src a' = src a 
  hence ∃ Qx fx. knd a' = Qx←fx 
  proof (induct rule:lift-valid-edge.induct) 
 37
case (lve-edge $ax' e$)

  note [simp] = \(e = (\text{Node } (\text{sourcenode } ax'), \text{kind } ax', \text{Node } (\text{targetnode } ax'))\)

  from \(\langle \text{src } e = \text{src } a \rangle \langle \text{src } a = \text{Node } (\text{sourcenode } ax) \rangle\)
  have \(\text{sourcenode } ax' = \text{sourcenode } ax\) by simp
  with \(\text{valid-edge } ax'\) all have \(\exists Qx fx. \text{kind } ax' = Qx \xrightarrow{p} fx\) by blast
  thus \(\text{?case by simp}\)

next

case (lve-Entry-edge $e$)

  from \(\langle e = (\text{NewEntry}, (\lambda s. \text{True}) \rightarrow, \text{Node Entry})\rangle \langle \text{src } e = \text{src } a \rangle \langle \text{src } a = \text{Node } (\text{sourcenode } ax) \rangle\)
  have \(\text{False}\) by simp
  thus \(\text{?case by simp}\)

next

case (lve-Exit-edge $e$)

  from \(\langle e = (\text{Node Exit}, (\lambda s. \text{True}) \rightarrow, \text{NewExit})\rangle \langle \text{src } e = \text{src } a \rangle \langle \text{src } a = \text{Node } (\text{sourcenode } ax) \rangle\)
  have \(\text{False}\) by \((\text{rule Exit-source})\)
  thus \(\text{?case by simp}\)

next

case (lve-Entry-Exit-edge $e$)

  from \(\langle e = (\text{NewEntry}, (\lambda s. \text{False}) \rightarrow, \text{NewExit})\rangle \langle \text{src } e = \text{src } a \rangle \langle \text{src } a = \text{Node } (\text{sourcenode } ax) \rangle\)
  have \(\text{False}\) by simp
  thus \(\text{?case by simp}\)

next

\[\forall a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \land \text{src } a' = \text{src } a \rightarrow (\exists Qx fx. \text{kind } a' = Qx \xrightarrow{p} fx)\] by simp

next

fix $a Q r p fs$

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

and \(\text{kind } a = Q: r \xrightarrow{p} fs\)

thus \(\text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind } a \neq \{\}\)

proof (induct rule: lift-valid-edge.induct)

case (lve-edge $ax e$)

  from \(\langle e = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax))\rangle \langle \text{kind } e = Q: r \xrightarrow{p} fs \rangle\)
  have \(\text{kind } ax = Q: r \xrightarrow{p} fs\) by simp
  with \(\text{valid-edge } ax\) have \(\text{get-return-edges } ax \neq \{\}\)
  by (rule get-return-edge-call)
  then obtain $ax'$ where $ax' \in \text{get-return-edges } ax$ by blast
  with \(\langle e = (\text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax))\rangle \langle \text{valid-edge } ax\rangle\)
  have \(\text{Node } (\text{sourcenode } ax'), \text{kind } ax', \text{Node } (\text{targetnode } ax')) \in \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind } e\)
  by (fastforce intro: Lift-get-return-edgesI)
  thus \(\text{?case by fastforce}\)

qed simp-all

next
fix a a′

assume a′ ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a

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

thus lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a′

proof (induct rule: lift-get-return-edges.induct)

case (lift-get-return-edges! ax a′ e′)

from ⟨valid-edge ax⟩ (a′ ∈ get-return-edges ax) have valid-edge a′

by (rule get-return-edges-valid)

from ⟨valid-edge ax⟩ (a′ ∈ get-return-edges ax) obtain Q r p fs

where kind ax = Q : r → p fs by (fastforce dest!: only-call-get-return-edges)

with ⟨valid-edge ax⟩ (a′ ∈ get-return-edges ax) obtain Q′ f′

where kind a′ = Q′ : f′ → p fs by (fastforce dest!: call-return-edges)

from ⟨valid-edge a′⟩ (kind a′ = Q′ : f′ → p fs) have get-proc(sourcenode a′) = p

by (rule get-proc-return)

have sourcenode a′ ≠ Entry

proof

assume sourcenode a′ = Entry

with get-proc-Entry ⟨get-proc(sourcenode a′) = p⟩ have p = Main by simp

with ⟨kind a′ = Q′ : f′ → p fs⟩ have kind a′ = Q′ : Main f′ by simp

with ⟨valid-edge a′⟩ show False by (rule Main-no-return-source)

qed

with ⟨e′ = (Node (sourcenode a′), kind a′, Node (targetnode a′))⟩

⟨valid-edge a′⟩

show ?case by (fastforce intro: lve-edge)

qed

next

fix a a′

assume a′ ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a

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

thus ∀ Q r p fs. kind a = Q : r → p fs

proof (induct rule: lift-get-return-edges.induct)

case (lift-get-return-edges! ax a′ e′)

from ⟨valid-edge ax⟩ (a′ ∈ get-return-edges ax) have ∀ Q r p fs. kind ax = Q : r → p fs

by (rule only-call-get-return-edges)

with ⟨a = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩

show ?case by simp

qed

next

fix a Q r p fs a′

assume a′ ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a and kind a = Q : r → p fs

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

thus ∀ Q′ f′. kind a′ = Q′ : f′ → p fs

proof (induct rule: lift-get-return-edges.induct)

case (lift-get-return-edges! ax a′ e′)

from ⟨a = (Node (sourcenode ax), kind ax, Node (targetnode ax))⟩


⟨knd a = Q;r→pfs⟩

have kind ax = Q;r→pfs by simp

with (valid-edge ax) (a' ∈ get-return-edges ax) have ∃ Q' f'. kind a' = Q'∈→p'f' by -(rule call-return-edges)

with (e' = (Node (sourcenode a'), kind a', Node (targetnode a')))

show ?case by simp

qed

next

fix a Q' p f'

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

and knd a = Q'∈→p'f'

thus ∃!a'. lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' ∧

(∃ Q r fs. knd a' = Q;r→pfs) ∧ a ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind a'

proof (induct rule:lift-valid-edge.induct)

case (lve-edge a e)

from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩

⟨knd e = Q'∈→p'f'⟩ have kind a = Q'∈→p'f' by simp

with (valid-edge a)

have ∃!a'. valid-edge a' ∧ (∃ Q r fs. kind a' = Q;r→pfs) ∧ a ∈ get-return-edges a'

by (rule return-needs-call)

then obtain a' Q r fs where valid-edge a' and kind a' = Q;r→pfs

and a ∈ get-return-edges a'

and imp:∀ x. valid-edge x ∧ (∃ Q r fs. kind x = Q;r→pfs) ∧ a ∈ get-return-edges x → x = a'

by (fastforce elim:ex1E)

let ?e' = (Node (sourcenode a'),kind a',Node (targetnode a'))

have sourcenode a' ≠ Entry

proof

assume sourcenode a' = Entry

with (valid-edge a') ⟨kind a' = Q;r→pfs⟩

show False by (rule Entry-no-call-source)

qed

with (valid-edge a')

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e'

by (fastforce intro:lift-valid-edge.lve-edge)

moreover

from ⟨kind a' = Q;r→pfs⟩ have knd ?e' = Q;r→pfs by simp

moreover

from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩

⟨valid-edge a', a ∈ get-return-edges a'⟩

have e ∈ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind ?e' by (fastforce intro:lift-get-return-edgesI)

moreover

{ fix x

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

and ∃ Q r fs. knd x = Q;r→pfs

and e ∈ lift-get-return-edges get-return-edges valid-edge }
source
node target
node kind x
d
from (lift-valid-edge valid-edge source
node target
node kind Entry Exit x)
∃ Q r fs. kind x = Q:r→pfs obtain y where valid-edge y
and x = (Node (source
node y), kind y, Node (target
node y))
by (fastforce elim:lift-valid-edge.cases)
with (e ∈ lift-get-return-edges get-return-edges valid-edge
source
node target
node kind x) (valid-edge a)
(e = (Node (source
node a), kind a, Node (target
node a)))

have x = ?e'
proof (induct rule: lift-get-return-edges.induct)
case (lift-get-return-edgesI ax ax' e)

from (valid-edge ax) (ax' ∈ get-return-edges ax) have valid-edge ax'
by (rule get-return-edges-valid)
from (e = (Node (source
node ax'), kind ax', Node (target
node ax')))
(e = (Node (source
node a), kind a, Node (target
node a)))

have source
node a = source
node ax' and target
node a = target
node ax'
by simp-all

with (valid-edge a) (valid-edge ax' have [simp]: a = ax' by (rule edge-det)
from ax = (Node (source
node ax), kind ax, Node (target
node ax)):
∃ Q r fs. kind x = Q:r→pfs have ∃ Q r fs. kind ax = Q:r→pfs by simp
with (valid-edge ax) (ax' ∈ get-return-edges ax) imp
have ax = a' by fastforce

with (ax = (Node (source
node ax), kind ax, Node (target
node ax)))

show ?thesis by simp

qed 
ultimately show ?case by (blast intro:exI1)

qed simp-all

next

fix a a'
assume a' ∈ lift-get-return-edges get-return-edges valid-edge source
node target
node kind a
and lift-valid-edge valid-edge source
node target
node kind Entry Exit a
thus ∃ a'', lift-valid-edge valid-edge source
node target
node kind Entry Exit a'' ∧
src a'' = trg a ∧ trg a'' = src a' ∧ kind a'' = (∀cf . False)
proof (induct rule: lift-get-return-edges.induct)
case (lift-get-return-edgesI ax a' e')
from (valid-edge ax) (a' ∈ get-return-edges ax)

obtain ax' where valid-edge ax' and source
node ax' = target
node ax
and target
node ax' = source
node a' and kind ax' = (∀cf . False) 
by (fastforce dest: intra-proc-additional-edge)
let ?ex = (Node (source
node ax'), kind ax', Node (target
node ax'))

have target
node ax ≠ Entry

proof

assume target
node ax = Entry

with (valid-edge ax) show False by (rule Entry-target)

qed

with (source
node ax' = target
node ax) have source
node ax' ≠ Entry by simp

with (valid-edge ax')

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

41
\[
\begin{align*}
&\text{by (fastforce intro: lve-edge)} \\
&\text{with } (e' = (\text{Node (sourcenode } a'), \text{kind } a', \text{Node (targetnode } a'))) \\
&\quad (a = (\text{Node (sourcenode } ax), \text{kind } ax, \text{Node (targetnode } ax))) \\
&\quad (e' = (\text{Node (sourcenode } a'), \text{kind } a', \text{Node (targetnode } a'))): \\
&\quad \text{show } ?\text{case by simp} \\
&\text{qed}
\end{align*}
\]

next

fix \( a \ a' \)

assume \( a' \in \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind } a \)

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

thus \( \exists a'''\). lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'''' \land \\
\quad \text{src } a'''' = \text{src } a \land \text{trg } a'''' = \text{trg } a' \land \text{kind } a'''' = (\lambda cf. \text{False}) \\
\text{proof (induct rule: lift-get-return-edges.induct)}

\text{case } (\text{lift-get-return-edgesI ax a } e')

from \( (\text{valid-edge ax} \quad (a' \in \text{get-return-edges ax}) \)

obtain ax' where valid-edge ax' and sourcenode ax' = sourcenode ax

and targetnode ax' = targetnode a' and kind ax' = (\lambda cf. \text{False}) \\
\text{by (fastforce dest: call-return-node-edge)}

let ?ex = (\text{Node (sourcenode ax')}, \text{kind ax'}, \text{Node (targetnode ax')})

from \( (\text{valid-edge ax} \quad (a' \in \text{get-return-edges ax}) \)

obtain Q r p fs where kind ax = Q:r\to p:fs

by (fastforce dest!: only-call-get-return-edges)

have sourcenode ax \neq \text{Entry}

proof

assume sourcenode ax = \text{Entry}

with \( (\text{valid-edge ax} \quad (\text{kind ax } = Q:r\to p:fs)) \text{ show False} \\
\text{by (rule Entry-no-call-source)} \\
\text{qed}

\text{with } (\text{sourcenode ax' } = \text{sourcenode ax}) \text{ have sourcenode ax' } \neq \text{Entry} \text{ by simp}

\text{with } (\text{valid-edge ax'})

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

by (fastforce intro: lve-edge)

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

\quad (a = (\text{Node (sourcenode } ax), \text{kind } ax, \text{Node (targetnode } ax))):

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

\quad \text{show } ?\text{case by simp} \\
\text{qed}

next

fix \( a \ Q r p fs \)

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

and \( \text{kind } a = Q:r\to p:fs \)

thus \( \exists a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \land \\
\quad \text{src } a' = \text{src } a \land \text{intra-kind } (\text{kind } a') \)

\text{proof (induct rule: lift-valid-edge.induct)}
case \((\text{lve-edge}\ a\ e)\)

from \((e = (\text{Node}\ (\text{sourcenode}\ a),\ \text{kind}\ a,\ \text{Node}\ (\text{targetnode}\ a)))\) \((\text{knd}\ e = Q:r\rightarrow pfs)\)

have kind\(a = Q:r\rightarrow pfs\) by simp

with \((\text{valid-edge}\ a)\) have \(\exists!a'. \text{valid-edge}\ a' \land \text{sourcenode}\ a' = \text{sourcenode}\ a\)

\(\land\) intra-kind\((\text{kind}\ a')\) by\((\text{rule}\ \text{call-only-one-intra-edge})\)

then obtain \(a'\) where \(\text{valid-edge}\ a'\land\text{sourcenode}\ a' = \text{sourcenode}\ a\)

and intra-kind\((\text{kind}\ a')\)

and imp:\(\forall x. \text{valid-edge}\ x \land \text{sourcenode}\ x = \text{sourcenode}\ a \land \text{intra-kind}\(\text{kind}\ x)\)

\(\rightarrow x = a'\) by\((\text{fastforce elim:ex1E})\)

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

have \(\text{sourcenode}\ a \neq \text{Entry}\) by simp

proof

assume \(\text{sourcenode}\ a = \text{Entry}\)

with \((\text{valid-edge}\ a)\) \((\text{kind}\ a = Q:r\rightarrow pfs)\) show False

by\((\text{rule Entry-no-call-source})\)

qed

with \((\text{sourcenode}\ a' = \text{sourcenode}\ a)\) have \(\text{sourcenode}\ a' \neq \text{Entry}\) by simp

with \((\text{valid-edge}\ a')\)

have \(\text{lift-valid-edge}\ \text{valid-edge}\ \text{sourcenode}\ \text{targetnode}\ \text{kind}\ \text{Entry}\ \text{Exit}\ ?e'\)

by\((\text{fastforce intro:lift-valid-edge.lve-edge})\)

moreover

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

\((\text{sourcenode}\ a' = \text{sourcenode}\ a)\)

have src\(?e' = \text{src}\ e\) by simp

moreover

from \((\text{intra-kind}(\text{kind}\ a'))\) have \(\text{intra-kind}\ (\text{kind}\ ?e')\) by simp

moreover

\{ fix \(x\)

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

and \(\text{src}\ x = \text{src}\ e\) and \(\text{intra-kind}\ (\text{kind}\ x)\)

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

have \(x = ?e'\)

proof\((\text{induct rule:lift-valid-edge.cases})\)

\text{case}\ (\text{lve-edge}\ ax\ ex)\)

from \((\text{intra-kind}\ (\text{kind}\ x)); (x = ex)\) \((\text{src}\ x = \text{src}\ e)\)

\((ex = (\text{Node}\ (\text{sourcenode}\ ax),\ \text{kind}\ ax,\ \text{Node}\ (\text{targetnode}\ ax)))\)

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

have \(\text{intra-kind}\ (\text{kind}\ ax)\) and \(\text{sourcenode}\ ax = \text{sourcenode}\ a\) by simp-all

with \((\text{valid-edge}\ ax)\) \(\text{imp}\ have\ ax = a'\) by fastforce

with \((x = ex)\) \((ex = (\text{Node}\ (\text{sourcenode}\ ax),\ \text{kind}\ ax,\ \text{Node}\ (\text{targetnode}\ ax)))\)

\show \(?case\ by\ simp\)

next

\text{case}\ (\text{lve-Entry-edge}\ ex)\)

with \((\text{src}\ x = \text{src}\ e)\)

\((e = (\text{Node}\ (\text{sourcenode}\ a),\ \text{kind}\ a,\ \text{Node}\ (\text{targetnode}\ a)))\)
have False by simp
thus ?case by simp
next
case (lve-Exit-edge ex)
  with \(\text{src} x = \text{src} e\):
  \((e = (\text{Node} (\text{sourcenode} a), \text{kind} a, \text{Node} (\text{targetnode} a)))\)
  have sourcenode a = Exit by simp
  with \((\text{valid-edge} a)\) have False by (rule Exit-source)
  thus ?case by simp
next
case (lve-Entry-Exit-edge ex)
  with \(\text{src} x = \text{src} e\):
  \((e = (\text{Node} (\text{sourcenode} a), \text{kind} a, \text{Node} (\text{targetnode} a)))\)
  have False by simp
  thus ?case by simp
qed
ultimately show ?case by (blast intro: ex1I)
qed simp-all
next
fix a Q' p f'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q' \leftarrow pf'
thus \(\exists !a'. \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a' \land try a' = try a \land intra-kind (knd a')\)}
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from \((e = (\text{Node} (\text{sourcenode} a), \text{kind} a, \text{Node} (\text{targetnode} a)))\) \(\langle \text{knd} e = Q' \leftarrow pf' \rangle\)
  have \(\text{knd} a = Q' \leftarrow pf'\) by simp
  with \((\text{valid-edge} a)\) have \(\exists !a'. \text{valid-edge a' \land targetnode a' = targetnode a \land intra-kind} (\text{knd a'})\) by (rule return-only-one-intra-edge)
then obtain a' where valid-edge a' and targetnode a' = targetnode a
  and intra-kind (kind a')
  and imp:\(\forall x. \text{valid-edge} x \land \text{targetnode} x = \text{targetnode} a \land \text{intra-kind} (\text{kind} x)\)
  \(\rightarrow x = a'\) by (fastforce elim: ex1E)
let \(\?e' = (\text{Node} (\text{sourcenode} a'), \text{kind} a', \text{Node} (\text{targetnode} a'))\)
have targetnode a \neq Exit
proof
  assume targetnode a = Exit
  with \((\text{valid-edge} a')\) \(\langle \text{knd} a = Q' \leftarrow pf' \rangle\) show False
  by (rule Exit-no-return-target)
qed
with \((\text{targetnode} a' = \text{targetnode} a)\) have targetnode a' \neq Exit by simp
with \((\text{valid-edge} a')\)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e'
  by (fastforce intro: lift-valid-edge.lve-edge)
moreover
from \((e = (\text{Node} (\text{sourcenode} a), \text{kind} a, \text{Node} (\text{targetnode} a)))\)
\[
\text{have } \text{trg } ?e' = \text{trg } e \text{ by simp}
\]

Moreover

\[
\text{from } \langle \text{intra-kind}(\text{kind } a') \rangle \text{ have intra-kind } (\text{kind } ?e') \text{ by simp}
\]

Moreover

\[
\{ \text{fix } x \cr \text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \cr \text{and } \text{trg } x = \text{trg } e \text{ and intra-kind } (\text{kind } x) \cr \text{from } \langle \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } x \rangle \cr \text{have } x = ?e' \cr \}\text{ proof } \cr (\text{induct rule: lift-valid-edge. cases}) \cr \text{case } (\text{lve-edge ax ex}) \cr \text{from } \langle \text{intra-kind } (\text{kind } x) \rangle \langle x = ex \rangle \langle \text{trg } x = \text{trg } e \rangle \cr \langle \text{e} = \langle \text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax) \rangle \rangle \cr \text{have intra-kind } (\text{kind } ax) \text{ and } \text{targetnode } ax = \text{targetnode } a \text{ by simp-all} \cr \text{with } \langle \text{valid-edge ax} \rangle \text{ imp have } ax = a' \text{ by fastforce} \cr \text{with } \langle x = ex \rangle \langle ex = \langle \text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax) \rangle \rangle \cr \text{show } ?\text{case by simp} \cr \text{next} \cr \text{case } (\text{lve-Entry-edge ex}) \cr \text{with } \langle \text{trg } x = \text{trg } e \rangle \cr \langle e = \langle \text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax) \rangle \rangle \cr \text{have targetnode } a = \text{Entry by simp} \cr \text{with } \langle \text{valid-edge a} \rangle \text{ have False by(rule Entry-target)} \cr \text{thus } ?\text{case by simp} \cr \text{next} \cr \text{case } (\text{lve-Exit-edge ex}) \cr \text{with } \langle \text{trg } x = \text{trg } e \rangle \cr \langle e = \langle \text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax) \rangle \rangle \cr \text{have False by simp} \cr \text{thus } ?\text{case by simp} \cr \text{next} \cr \text{case } (\text{lve-Entry-Exit-edge ex}) \cr \text{with } \langle \text{trg } x = \text{trg } e \rangle \cr \langle e = \langle \text{Node } (\text{sourcenode } ax), \text{kind } ax, \text{Node } (\text{targetnode } ax) \rangle \rangle \cr \text{have False by simp} \cr \text{thus } ?\text{case by simp} \cr \text{qed} \} \cr \text{ultimately show } ?\text{case by(blast intro:exII)} \cr \text{qed simp-all} \cr \text{next} \cr \text{fix } a a' Q_1 r_1 p s_1 Q_2 r_2 s_2 \cr \text{assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a \cr \text{and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit } a' \cr \text{and } \text{knd } a = Q_1 : r_1 \to p s_1 \text{ and } \text{knd } a' = Q_2 : r_2 \to p s_2 \cr \text{then obtain } x x' \text{ where valid-edge } x \cr \text{and } a:a = \langle \text{Node } (\text{sourcenode } x), \text{kind } x, \text{Node } (\text{targetnode } x) \rangle \text{ and valid-edge}
\[ x' \]
\[
\text{and } a' a' = (\text{Node} (\text{sourcenode } x'), \text{kind } x', \text{Node} (\text{targetnode } x'))
\]
\[
\text{by (auto elim! : lift-valid-edge \_cases)}
\]
\[
\text{with } \text{kind } a = Q_1 : r_1 \mapsto p f s_1 \text{ and } \text{kind } a' = Q_2 : r_2 \mapsto p f s_2
\]
\[
\text{by simp-all}
\]
\[
\text{have } \text{kind } x = Q_1 : r_1 \mapsto p f s_1 \text{ and } \text{kind } x' = Q_2 : r_2 \mapsto p f s_2
\]
\[
\text{by (rule same-proc-call-unique-target)}
\]
\[
\text{with } a a' \text{ show } \text{try } a = \text{try } a' \text{ by simp}
\]
\[
\text{next}
\]
\[
\text{from unique-callers show distinct-fst procs}.
\]
\[
\text{next}
\]
\[
\text{fix } p \text{ ins outs}
\]
\[
\text{assume } (p, \text{ins, outs}) \in \text{set procs}
\]
\[
\text{from distinct-formal-ins[OF this] show distinct ins}.
\]
\[
\text{next}
\]
\[
\text{fix } p \text{ ins outs}
\]
\[
\text{assume } (p, \text{ins, outs}) \in \text{set procs}
\]
\[
\text{from distinct-formal-outs[OF this] show distinct outs}.
\]
\[
\text{qed}
\]
\[
\text{qed}
\]

**lemma** lift-CFG-wf:

**assumes** wf: CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses

**and** pd: Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit

**shows** CFG-wf src trg knd
\[
(\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}) \text{ NewEntry}
\]
\[
(\text{lift-get-proc get-proc Main})
\]
\[
(\text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind})
\]
\[
\text{procs Main (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)}
\]
\[
(\text{lift-ParamDefs ParamDefs}) (\text{lift-ParamUses ParamUses})
\]

**proof** –

**interpret** CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses

**by (rule wf)**

**interpret** Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit

**by (rule pd)**

**interpret** CFG CFG src trg knd
\[
\text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry}
\]
\[
\text{lift-get-proc get-proc Main}
\]
\[
\text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind}
\]
\[
\text{procs Main}
\]

**by (fastforce intro: lift-CFG wf pd)**

**show** ?thesis

**proof**

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


\begin{verbatim}

lift-Use Entry Exit H L NewEntry = {}
by (fastforce elim: lift-Use-set.cases lift-Def-set.cases)

next
fix a Q r p fs ins outs
assume lift-valid-edge valid-edge source node target node kind Entry Exit a
and knd a = Q: \rightarrow p fs and (p, ins, outs) \in set procs
thus length (lift-ParamUses ParamUses (src a)) = length ins
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
from \langle e = (Node (source node a), kind a, Node (target node a)) \rangle \langle knd e = Q: \rightarrow p fs \rangle
have kind a = Q: \rightarrow p fs and src e = Node (source node a) by simp-all
with \langle valid-edge a \rangle \langle (p, ins, outs) \in set procs \rangle
have length (ParamUses (source node a)) = length ins
by -(rule ParamUses-call-source-length)
with \langle src e = Node (source node a) \rangle show \ ?case by simp
qed simp-all

next
fix a assume lift-valid-edge valid-edge source node target node kind Entry Exit a
thus distinct (lift-ParamDefs ParamDefs (try a))
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
from \langle valid-edge a \rangle have distinct (ParamDefs (target node a))
by (rule distinct-ParamDefs)
with \langle e = (Node (source node a), kind a, Node (target node a)) \rangle
show \ ?case by simp

case (lve-Entry-edge e)
have ParamDefs Entry = []
proof (rule ccontr)
assume ParamDefs Entry \neq []
then obtain V Vs where ParamDefs Entry = V \# Vs
by (cases ParamDefs Entry) auto
hence V \in set (ParamDefs Entry) by fastforce
hence V \in Def Entry by (fastforce intro: ParamDefs-in-Def)
with Entry-empty show False by simp
qed
with \langle e = (NewEntry, (\lambda s. True), Node Entry) \rangle show \ ?case by simp
qed simp-all

next
fix a Q’ p f’ ins outs
assume lift-valid-edge valid-edge source node target node kind Entry Exit a
and knd a = Q’ \rightarrow p f’ and (p, ins, outs) \in set procs
thus length (lift-ParamDefs ParamDefs (try a)) = length outs
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
from \langle e = (Node (source node a), kind a, Node (target node a)) \rangle
\langle knd e = Q’ \rightarrow p f’ \rangle
have kind a = Q’ \rightarrow p f’ and try e = Node (target node a) by simp-all
\end{verbatim}
with \( \langle \text{valid-edge } a \rangle \cdot \langle p, \text{ins}, \text{outs} \rangle \in \text{set procs} \)

have \( \text{length(ParamDefs targetnode } a) = \text{length outs} \)
   by \( - (\text{rule ParamDefs-return-target-length}) \)

with \( \langle \text{try } e = \text{Node targetnode } a \rangle \) show \?case by simp

qed simp-all

next

fix \( n \ V \)

assume \( C.F.G.CFG.\text{valid-node } \text{src} \ \text{try} \)

\( (\text{lift-valid-edge } \text{valid-edge } \text{source-node } \text{target-node } \text{kind } \text{Entry} \ \text{Exit} \ a) \ n \)

and \( V \in \text{set} \ (\text{lift-ParamDefs } \text{ParamDefs } n) \)

hence \( ((n = \text{NewEntry}) \lor n = \text{NewExit}) \lor (\exists m. \ n = \text{Node } m \land \text{valid-node } m) \)

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

thus \( V \in \text{lift-Def } \text{Def } \text{Entry} \ \text{Exit} \ \text{H} \ \text{L} \ n \ \text{apply} - \)

proof \( (\text{erule disjE})+ \)

 assume \( n = \text{NewEntry} \)

with \( \langle V \in \text{set} \ (\text{lift-ParamDefs } \text{ParamDefs } n) \rangle \) show \?thesis by simp

next

 assume \( n = \text{NewExit} \)

with \( \langle V \in \text{set} \ (\text{lift-ParamDefs } \text{ParamDefs } n) \rangle \) show \?thesis by simp

next

 assume \( \exists m. \ n = \text{Node } m \land \text{valid-node } m \)

then obtain \( m \) where \( n = \text{Node } m \) and \( \text{valid-node } m \) by blast

from \( \langle n = \text{Node } m. \ V \in \text{set} \ (\text{lift-ParamDefs } \text{ParamDefs } n) \rangle \)

have \( V \in \text{set} \ (\text{ParamDefs } m) \) by simp

with \( \langle \text{valid-node } m \rangle \) have \( V \in \text{Def } m \) by \( (\text{rule ParamDefs-in-Def}) \)

with \( \langle n = \text{Node } m \rangle \) show \?thesis by \( (\text{fastforce intro:lift-Def-node}) \)

qed simp-all

next

fix \( a \ Q \ r \ p \ fs \ \text{ins} \ \text{outs} \ V \)

assume \( \text{lift-valid-edge } \text{valid-edge } \text{source-node } \text{target-node } \text{kind } \text{Entry} \ \text{Exit} \ a \)

and \( \text{kind } a = Q: r \hookrightarrow p \ fs \) and \( \langle p, \text{ins}, \text{outs} \rangle \in \text{set procs} \) and \( V \in \text{set ins} \)

thus \( V \in \text{lift-Def } \text{Def } \text{Entry} \ \text{Exit} \ \text{H} \ \text{L} \ \text{trg } a \)

proof \( (\text{induct rule:lift-valid-edge}.\ \text{induct}) \)

case \( (\text{lve-edge } a \ e) \)

from \( \langle e = (\text{Node } \text{source-node } a. \ \text{kind } a. \ \text{Node } \text{target-node } a) \rangle \) \( \langle \text{kind } e = Q: r \hookrightarrow p \fs \rangle \)

have \( \text{kind } a = Q: r \hookrightarrow p \fs \) by simp

from \( \langle \text{valid-edge } a \rangle \) \( \langle \text{kind } a = Q: r \hookrightarrow p \fs \rangle \) \( \langle p, \text{ins}, \text{outs} \rangle \in \text{set procs} \) \( \langle V \in \text{set ins} \rangle \)

have \( V \in \text{Def } \text{target-node } a \) by \( (\text{rule ins-in-Def}) \)

from \( \langle e = (\text{Node } \text{source-node } a. \ \text{kind } a. \ \text{Node } \text{target-node } a) \rangle \)

have \( \text{try } e = \text{Node } \text{target-node } a \) by simp

with \( \langle V \in \text{Def } \text{target-node } a \rangle \) show \?case by \( (\text{fastforce intro:lift-Def-node}) \)

qed simp-all

next

fix \( a \ Q \ r \ p \ fs \)

assume \( \text{lift-valid-edge } \text{valid-edge } \text{source-node } \text{target-node } \text{kind } \text{Entry} \ \text{Exit} \ a \)

and \( \text{kind } a = Q: r \hookrightarrow p \fs \)

48
thus \( \text{lift-Def} \) \( \text{Def} \) \( \text{Entry} \) \( \text{Exit} \) \( H \) \( L \) \( (\text{src} \ a) = \{\} \)

proof (induct rule: \text{lift-valid-edge}.induct)
case (\text{loc-edge} \ a \ e)
  show \(?\text{case}\)
  proof
    (rule \text{ccontr})
    assume \( \text{lift-Def} \) \( \text{Def} \) \( \text{Entry} \) \( \text{Exit} \) \( H \) \( L \) \( (\text{src} \ e) \neq \{\} \)
    then obtain \( x \) where \( x \in \text{lift-Def} \) \( \text{Def} \) \( \text{Entry} \) \( \text{Exit} \) \( H \) \( L \) \( (\text{src} \ e) \) by blast
    from \( e = (\text{Node} \ (\text{source} a), \text{kind} a, \text{Node} \ (\text{target} a)) \) \( \text{kind} e = \)
    \( Q; r\rightarrow_p fs \)
    have \( \text{kind} a = Q; r\rightarrow_p fs \) by simp
    with \( (\text{valid-edge} a) \) have \( \text{Def} \) \( \text{source} a = \{\} \)
      by (rule \text{call-source-Def-empty})
    have \( \text{source} a \neq \text{Entry} \)
      proof
        assume \( \text{source} a = \text{Entry} \)
        with \( (\text{valid-edge} a) \) have \( \text{Def} \) \( \text{source} a = \{\} \)
          by (rule \text{Entry-no-call-source})
      qed
    qed
    from \( e = (\text{Node} \ (\text{source} a), \text{kind} a, \text{Node} \ (\text{target} a)) \)
    have \( \text{src} \ e \neq \text{Node} \ (\text{source} a) \)
      by simp
    with \( (\text{Def} \ (\text{source} a) = \{\}) \) \( (\exists x. x \in \text{lift-Def} \) \( \text{Def} \) \( \text{Entry} \) \( \text{Exit} \) \( H \) \( L \) \( (\text{src} \ e) \) \( (\text{source} a) \neq \text{Entry} \)
    show \( \text{False} \) by (fastforce elim: \text{lift-Def-set}.cases)
  qed
  qed simp-all
next
fix \( n \) \( V \)
assume \( \text{CFG} \).\( \text{CFG} \).\( \text{valid-node} \) \( \text{src} \) \( \text{try} \)
(\text{lift-valid-edge} \) \( \text{valid-edge} \) \( \text{source} \) \( \text{target} \) \( \text{kind} \) \( \text{Entry} \) \( \text{Exit} \) \( n \)
and \( V \in \bigcup \) \( \text{set} \) \( (\text{lift-ParamUses} \) \( \text{ParamUses} \) \( n \) \)
hence \((n = \text{NewEntry}) \lor n = \text{NewExit}) \lor (\exists m. n = \text{Node} m \land \text{valid-node} m)\)
by (auto elim: \text{lift-valid-edge}.cases simp: \text{CFG}\.\text{valid-node-def})
thus \( V \in \text{lift-Use} \) \( \text{Use} \) \( \text{Entry} \) \( \text{Exit} \) \( H \) \( L \) \( n \) \( \text{apply} = \)
proof (erule disjE)+
  assume \( n = \text{NewEntry} \)
  with \( (V \in \bigcup \text{set} \) \( \text{lift-ParamUses} \) \( \text{ParamUses} \) \( n \)) show \(?\text{thesis}\) by simp
next
assume \( n = \text{NewExit} \)
with \( (V \in \bigcup \text{set} \) \( \text{lift-ParamUses} \) \( \text{ParamUses} \) \( n \)) show \(?\text{thesis}\) by simp
next
assume \( \exists m. n = \text{Node} m \land \text{valid-node} m \)
then obtain \( m \) where \( n = \text{Node} m \) and \( \text{valid-node} m \) by blast
from \( (V \in \bigcup \text{set} \) \( \text{lift-ParamUses} \) \( \text{ParamUses} \) \( n \)) \( (n = \text{Node} m) \)
have \( V \in \bigcup \text{set} \) \( \text{ParamUses} \) \( m \) \( \text{by simp} \)
with \( \text{valid-node} m \) have \( V \in \text{Use} m \) by (rule \text{ParamUses-in-Use})
with \( (n = \text{Node} m) \) show \(?\text{thesis}\) by (fastforce intro: \text{lift-Use-node})
qed
next
fix a Q p f ins outs V
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q←p,f and (p, ins, outs) ∈ set procs and V ∈ set outs
thus V ∈ lift-Use Use Entry Exit H L (src a)
proof (induct rule:lift-valid-edge.induct)
case (lee-edge a e)
  from e = (Node (sourcenode a), kind a, Node (targetnode a))
  have knd e = Q←p,f by simp
  from (valid-edge a) ⟨knd a = Q←p,f⟩ (p, ins, outs) ∈ set procs ⟨V ∈ set outs⟩
  have V ∈ Use (sourcenode a) by (rule outs-in-Use)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  have src e = Node (sourcenode a) by simp
  with ⟨V ∈ Use (sourcenode a)⟩ show ?case by (fastforce intro:lift-Use-node)
qed simp-all
next
fix a V s
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and V /∈ lift-Def Def Entry Exit H L (src a) and intra-kind (knd a)
and pred (knd a) s
thus state-val (transfer (knd a) s) V = state-val s V
proof (induct rule:lift-valid-edge.induct)
case (lee-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨intra-kind (knd e)⟩ (pred (knd e) s)
  have intra-kind (knd a) and pred (knd a) s
  and knd e = kind a and src e = Node (sourcenode a) by simp-all
  from ⟨V /∈ lift-Def Def Entry Exit H L (src e)⟩ (src e = Node (sourcenode a))
  have V /∈ Def (sourcenode a) by (auto dest: lift-Def-node)
  from ⟨valid-edge a⟩ ⟨V /∈ Def (sourcenode a)⟩ ⟨intra-kind (knd a)⟩
  (pred (knd a) s)
  have state-val (transfer (knd a) s) V = state-val s V
  by (rule CFG-intra-edge-no-Def-equal)
  with ⟨knd e = kind a⟩ show ?case by simp
next
case (lee-Entry-edge e)
  from ⟨e = (NewEntry, (λs. True)安全保障, Node Entry)⟩ (pred (knd e) s)
  show ?case by (cases s) auto
next
case (lee-Exit-edge e)
  from ⟨e = (Node Exit, (λs. True)安全保障, NewExit)⟩ (pred (knd e) s)
  show ?case by (cases s) auto
next
case (lee-Entry-Exit-edge e)
  from ⟨e = (NewEntry, (λs. False)安全保障, NewExit)⟩ (pred (knd e) s)
  have False by (cases s) auto
  thus ?case by simp
qed
next
  fix a s s'
  assume assms: lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  \forall V \in lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
  intra-kind (knd a) pred (knd a) s pred (knd a) s'
  show \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
proof
  fix V assume V \in lift-Def Def Entry Exit H L (src a)
  with assms
  show state-val (transfer (knd a) s) V = state-val (transfer (knd a) s') V
proof (induct rule: lift-valid-edge.induct)
  case (lve-edge a e)
    from \exists (\langle Node (sourcenode a), \text{kind } a, Node (targetnode a) \rangle) \langle intra-kind (\text{kind } e) \rangle have intra-kind (\text{kind } a) by simp
  show ?thesis by simp
next
  case True
  hence sourcenode a = Entry by simp
  from Entry-Exit-edge obtain a' where valid-edge a'
    and sourcenode a' = Entry and targetnode a' = Exit
  and kind a' = (\lambda s. False). by blast
  have \exists Q. kind a = (Q) .
  proof (cases targetnode a = Exit)
    case True
    with \langle valid-edge a \rangle \langle valid-edge a' \rangle \langle sourcenode a = Entry \rangle
      \langle sourcenode a' = Entry \rangle \langle targetnode a' = Exit \rangle
    have a = a' by (fastforce dest: edge-det)
    with \langle \text{kind } a' = (\lambda s. \text{False}) \rangle show \langle \text{thesis } \rangle by simp
next
  case False
  with \langle valid-edge a \rangle \langle valid-edge a' \rangle \langle sourcenode a = Entry \rangle
    \langle sourcenode a' = Entry \rangle \langle targetnode a' = Exit \rangle
  \langle intra-kind (\text{kind } a) \rangle \langle \text{kind } a' = (\lambda s. \text{False}) \rangle .
  show \langle \text{thesis } \rangle by (auto dest: deterministic simp: intra-kind-def)
qed
from True : V \in lift-Def Def Entry Exit H L (src e). Entry-empty
  \langle e = (\langle Node (sourcenode a), \text{kind } a, Node (targetnode a) \rangle) \rangle
  have V \in H by (fastforce elim: lift-Def-set.cases)
from True : e = (\langle Node (sourcenode a), \text{kind } a, Node (targetnode a) \rangle)
  \langle sourcenode a \neq \text{Entry} \rangle \langle targetnode a \neq \text{Exit} \rangle
  have \forall V \in H. V \in lift-Use Use Entry Exit H L (src e)
  by (fastforce intro: lift-Use-High)
from \forall V \in lift-Use Use Entry Exit H L (src e).
  state-val s V = state-val s' V; \langle V \in H \rangle
  have state-val s V = state-val s' V by simp
with \langle e = (\langle Node (sourcenode a), \text{kind } a, Node (targetnode a) \rangle) \rangle
  \langle \exists Q. \text{kind } a = (Q) .\langle \text{add } (knd e) s \rangle \langle \text{add } (knd e) s' \rangle \rangle
51
\begin{verbatim}
show ?thesis by{cases s,auto,cases s',auto}
next
case False
  { fix V' assume V' ∈ Use (sourcenode a)
    with (e = (Node (sourcenode a), kind a, Node (targetnode a)))
    have V' ∈ lift-Use Entry Exit H L (src e)
      by (fastforce intro:lift-Use-node)
  }
with (∀ V∈lift-Use Use Entry Exit H L (src e).
  state-val s V = state-val s' V)
have ∀ V∈Use (sourcenode a). state-val s V = state-val s' V
by fastforce
from (valid-edge a) this (pred (kind e) s) (pred (kind e) s')
  ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨intra-kind (kind e)⟩
have ∀ V ∈ Def (sourcenode a). state-val (transfer (kind a) s) V =
  state-val (transfer (kind a) s') V
by -(erule CFG-intra-edge-transfer-uses-only-Use,auto)
from (V ∈ lift-Def Def Entry Exit H L (src e)) False
  ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
have V ∈ Def (sourcenode a) by (fastforce elim:lift-Def-set.cases)
with (∀ V ∈ Def (sourcenode a). state-val (transfer (kind a) s) V =
  state-val (transfer (kind a) s') V)
  ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
show ?thesis by simp
qed
next
case (lve-Entry-edge e)
from (V ∈ lift-Def Def Entry Exit H L (src e))
  ⟨e = (NewEntry, (λs. True), Node Entry)⟩
have False by (fastforce elim:lift-Def-set.cases)
thus ?case by simp
next
case (lve-Exit-edge e)
from (V ∈ lift-Def Def Entry Exit H L (src e))
  ⟨e = (Node Exit, (λs. True), NewExit)⟩
have False
by (fastforce elim:lift-Def-set.cases intro!:Entry-noteq-Exit simp:Exit-empty)
thus ?case by simp
next
case (lve-Entry-Exit-edge e)
thus ?case by(cases s) auto
qed
qed
next
fix a s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and pred (kind a) s and snd (hd s) = snd (hd s')
  and ∀ V∈lift-Use Use Entry Exit H L (src a). state-val s V = state-val s' V
\end{verbatim}
and length s = length s'
thus pred (knd a) s'
proof (induct rule: lift-valid-edge.induct)
case (lee-edge a e)
from: e = (Node (sourcenode a), kind a, Node (targetnode a)); (pred (knd e) s)
have pred (kind a) s and src e = Node (sourcenode a) by simp-all
from: src e = Node (sourcenode a)
\forall V \in lift-Use Use Entry Exit H L (src e). state-val s V = state-val s' V
have \forall V \in Use (sourcenode a). state-val s V = state-val s' V by (auto dest: lift-Use-node)
from: \forall V \in lift-Use Use Entry Exit H L (src e). state-val s V = state-val s' V
have \forall V \in Use (sourcenode a). state-val s V = state-val s' V by (auto dest: lift-Use-node)
this \langle length s = length s' \rangle have pred (kind a) s' by (rule CFG-edge-Uses-pred-equal)
with: e = (Node (sourcenode a), kind a, Node (targetnode a))
show ?case by simp
next
case (lee-Entry-edge e)
thus ?case by (cases s') auto
next
case (lee-Exit-edge e)
thus ?case by (cases s') auto
next
case (lee-Entry-Exit-edge e)
thus ?case by (cases s) auto
qed
next
fix a Q r p fs ins outs
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q : r \rightarrow p fs and (p, ins, outs) \in set procs
thus length fs = length ins
proof (induct rule: lift-valid-edge.induct)
case (lee-edge a e)
from: e = (Node (sourcenode a), kind a, Node (targetnode a)); (knd e = Q : r \rightarrow p fs)
have kind a = Q : r \rightarrow p fs by simp
from: \forall V \in lift-Use Use Entry Exit H L (src e). state-val s V = state-val s' V
have \forall V \in Use (sourcenode a). state-val s V = state-val s' V by (auto dest: lift-Use-node)
show ?case by (rule CFG-edge-Uses-pred-equal)
next
fix a Q r p fs a' Q' r' p' fs' s s'
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q : r \rightarrow p fs and knd a' = Q' : r' \rightarrow p' fs'
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
and src a = src a' and pred (knd a) s and pred (knd a') s
from: \forall V \in lift-Use Use Entry Exit H L (src e). state-val s V = state-val s' V
have pred (kind a) s and pred (kind a') s
obtain x where a:a = (Node (sourcenode x), kind x, Node (targetnode x))
and valid-edge x and src a = Node (sourcenode x)
and kind \(x = Q: r \rightarrow pfs\) and pred (kind \(x\)) \(s\)
by (fastforce elim: lift-valid-edge.cases)
from lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
\(\langle\text{knd} a' = Q: r \rightarrow pfs'\), \langle pred (knd a') \(s\)\rangle\)
obtain \(x'\) where \(a'\):\(a' = (\text{Node (sourcenode} x') , \text{kind} x' , \text{Node (targetnode} x')\))
and valid-edge \(x'\) and \(\langle\text{sourcenode} a' = \text{Node (sourcenode} x')\rangle\)
and kind \(x' = Q: r \rightarrow pfs'\) and pred (kind \(x'\)) \(s\)
by (fastforce elim: lift-valid-edge.cases)
from \(\langle\text{sourcenode} a = \text{Node (sourcenode} x)\rangle\), \(\langle\text{sourcenode} a' = \text{Node (sourcenode} x')\rangle\)
have sourcenode \(x = \text{sourcenode} x'\) by simp
donext fix \(a\) \(r\) \(p\) \(fs\) \(i\) ins outs \(s\) \(s'\)
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \(\langle\text{knd} a = Q: r \rightarrow pfs\rangle\) \(i < \text{length} ins\) and \(\langle p, \text{ins}, \text{outs}\rangle \in \text{set} \text{procs}\)
and pred (kind \(a\)) \(s\) and pred (kind \(a\)) \(s'\)
and \(\forall V \in \text{lift-ParamUses}\) ParamUses \(\langle\text{sourcenode} a\rangle\) \(\langle\text{state-val} s V = \text{state-val} s' V\rangle\)
thus params \(fs\) \(\langle\text{state-val} s\rangle\) \(\langle\text{state-val} s'\rangle\) \(\langle i\rangle\)
proof (induct rule: lift-valid-edge.induct)
case (lee-edge \(a\) \(e\))
from \(\langle e = (\text{Node (sourcenode} a), \text{kind} a, \text{Node (targetnode} a))\rangle\)
\(\langle\text{knd} e = Q: r \rightarrow pfs\rangle\)
\(\langle\text{pred} (\text{knd} e) s\rangle\) \(\langle\text{pred} (\text{knd} e) s'\rangle\)
have \(\langle\text{knd} a = Q: r \rightarrow pfs\rangle\) \(\langle\text{kind} a s\rangle\) and \(\langle\text{kind} a\rangle\) \(s'\)
and \(\langle\text{sourcenode} e = \text{Node (sourcenode} a)\rangle\)
by simp-all
from \(\langle\forall V \in \text{lift-ParamUses}\) ParamUses \(\langle\text{sourcenode} e\rangle\) \(\langle\text{state-val} s V = \text{state-val} s' V\rangle\)
\(\langle\text{sourcenode} e = \text{Node (sourcenode} a)\rangle\)
have \(\forall V \in (\text{ParamUses (sourcenode} a)) \langle i\rangle\). \(\text{state-val} s V = \text{state-val} s' V\) by simp
with \(\langle\text{valid-edge} a\rangle\)
\(\langle\text{knd} a = Q: r \rightarrow pfs\rangle\) \(\langle i < \text{length} ins\rangle\)
\(\langle\text{(p, ins, outs)} \in \text{set} \text{procs}\rangle\) \(\langle\text{pred} (\text{knd} a) s\rangle\) \(\langle\text{pred} (\text{knd} a) s'\rangle\)
show \(\forall\langle\text{case} by (\text{rule CFG-call-edge-params})\)\nqed simp-all
next
fix \(a\) \(Q' p f'\) ins outs \(cf\) \(cf'\)
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and \(\langle\text{knd} a = Q' \leftarrow pfs'\rangle\) and \(\langle\text{(p, ins, outs)} \in \text{set} \text{procs}\rangle\)
thus \(f' cf cf' = cf'(\text{lift-ParamDefs}\) ParamDefs \(\langle\text{try} a\rangle\) \(\langle\text{map} cf\) \(\text{outs}\)\)
proof (induct rule: lift-valid-edge.induct)
case (lee-edge \(a\) \(e\))
from \(\langle e = (\text{Node (sourcenode} a), \text{kind} a, \text{Node (targetnode} a))\rangle\)
\(\langle\text{knd} e = Q' \leftarrow pfs'\rangle\)
have kind $a = Q' ← p f'$ and $t r g \ e = \text{Node} \ (t a r g e t n o d e \ a)$ by simp-all
from \langle valid-edge \ a \rangle \ (\text{kind} \ a = Q' ← p f' \ (p, \ i n s, \ o u t s) \in \text{set} \ \pro c s) \\
have $f' \ c f \ c f' = c f' \ (\text{ParamDefs} \ (t a r g e t n o d e \ a) [\ :=] \ \text{map} \ c f \ \text{outs})$ \\
by (rule CFG-return-edge-fun) \\
with \ (t r g \ e = \text{Node} \ (t a r g e t n o d e \ a)) \ \text{show} \ ?\text{case} \ \text{by} \ \text{simp} \\
qed simp-all \\
next \\
fix $a \ a'$ \\
assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit $a$ \\
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit $a'$ \\
and src $a = src \ a'$ and $t r g \ a \neq t r g \ a'$ \\
and intra-kind (kind $a$) and intra-kind (kind $a'$) \\
thus \exists \ Q, Q'. \ \text{kind} \ a = (Q) \ (Q') \land \ (\forall s. (Q \ s \rightarrow \neg Q' \ s) \land (Q' \ s \rightarrow \neg Q \ s))$
proof (induct rule: lift-valid-edge.induct) \\
\case (lve-edge $a \ e$) \\
from \langle lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit $a'$ \rangle \\
\langle \text{valid-edge} \ a \ (e = (\text{Node} \ (s o u r c e n o d e \ a), \ \text{kind} \ a, \ \text{Node} \ (t a r g e t n o d e \ a))) \rangle \\
\langle src \ e = src \ a' \land t r g \ e \neq t r g \ a' \ (\text{intra-kind} \ (\text{kind} \ e)) \ (\text{intra-kind} \ (\text{kind} \ a')) \rangle \\
\show ?\text{case} \\
proof (induct rule: lift-valid-edge.induct) \\
\case lve-edge \ \text{thus} ?\text{case} \ \text{by} (\text{auto dest: deterministic}) \\
next \\
\case (lve-Exit-edge $e'$) \\
from (e = (\text{Node} \ (s o u r c e n o d e \ a), \ \text{kind} \ a, \ \text{Node} \ (t a r g e t n o d e \ a))) \\
(e' = (\text{Node} \ Exit, (\lambda s. \ True), \ \text{NewExit}) e = src e = src e') \\
\have sourcenode $a = \text{Exit} \ \text{by simp} \\
\with \ (\text{valid-edge} a) \ \text{have False by (rule Exit-source)} \\
\text{thus} ?\text{case by simp} \\
\qed auto \\
\qed (\text{fastforce elim: lift-valid-edge.cases}+) \\
\qed
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
by(rule pd)

interpret CFG:CFG src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main
by(fastforce intro:lift-CFG wf pd)

show ?thesis
proof
  fix a assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and src a = NewExit
  thus False by(fastforce elim:lift-valid-edge.cases)
next
  show lift-get-proc get-proc Main NewExit = Main by simp
next
  fix a Q p f
  assume lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  and knd a = Q←↩p,f and trg a = NewExit
  thus False by(fastforce elim:lift-valid-edge.cases)
next
  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 intro:lve-Entry-Exit-edge)
qed
qed

lemma lift-CFGExit-wf:
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
shows CFGExit-wf src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main NewExit (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
proof —
interpret CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
by(rule wf)
interpret Postdomination sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit
by(rule pd)
interpret CFG-wf:CFG-wf src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main lift-Def Def Entry Exit H L lift-Use Use Entry Exit H L
lift-ParamDefs ParamDefs lift-ParamUses ParamUses
by (fastforce intro: lift-CFG-wf wf pd)

**interpret** CFGExit: CFGExit src trg knd
lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
lift-get-proc get-proc Main
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
procs Main NewExit
by (fastforce intro: lift-CFGExit wf pd)

show ?thesis
proof
  show lift-Def Def Entry Exit H L NewExit = {} ∧
  lift-Use Use Entry Exit H L NewExit = {}
  by (fastforce elim: lift-Def-set cases lift-Use-set.cases)
qed
qed

### 3.2.2 Lifting the SDG

**lemma** lift-Postdomination:
  **assumes** wf: CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
  get-return-edges procs Main Exit Def Use ParamDefs ParamUses
  and pd: Postdomination sourcenode targetnode kind valid-edge Entry get-proc
  get-return-edges procs Main Exit
  and inner: CFGExit-inner-node sourcenode targetnode valid-edge Entry Exit nx
  shows Postdomination src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
  (lift-get-proc get-proc Main)
  (lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
  procs Main NewExit

proof –
  **interpret** CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc
  get-return-edges procs Main Exit Def Use ParamDefs ParamUses
  by (rule wf)
  **interpret** Postdomination sourcenode targetnode kind valid-edge Entry get-proc
  get-return-edges procs Main Exit
  by (rule pd)
  **interpret** CFGExit: CFGExit src trg knd
  lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit NewEntry
  lift-get-proc get-proc Main
  lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
  procs Main NewExit
  by (fastforce intro: lift-CFGExit wf pd)

  { fix m assume valid-node m
    then obtain a where valid-edge a and m = sourcenode a ∨ m = targetnode
    a
    by (auto simp: valid-node-def) }
from \( (m = \text{source}node \ a \lor m = \text{target}node \ a) \)

have \( \text{CFG}.\text{CFG}.\text{valid-node} \ src \ trg \)

\((\text{lift-valid-edge} \ \text{valid-edge} \ \text{source}node \ \text{target}node \ \text{kind} \ \text{Entry} \ \text{Exit}) \ (\text{Node} \ m)\)

proof

assume \( m = \text{source}node \ a \)

show \( {?thesis} \)

proof (cases \( m = \text{Entry} \))


case True

have \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source}node \ \text{target}node \ \text{kind} \ \text{Entry} \ \text{Exit} \)

\((\text{NewEntry},(\lambda s. \ True)\_\text{\_},\text{Node Entry}) \ \text{by}(\text{fastforce intro: lve-Entry-edge})\)

with \( m = \text{Entry} \): show \( {?thesis} \) by (fastforce simp: CFGExit.valid-node-def)

next

case False

with \( m = \text{source}node \ a \) ⟨ valid-edge a ⟩

have \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source}node \ \text{target}node \ \text{kind} \ \text{Entry} \ \text{Exit} \)

\((\text{Node} \ (\text{source}node \ a), \text{kind} \ a, \text{Node} \ (\text{target}node \ a))\)

by (fastforce intro: lve-edge)

with \( m = \text{source}node \ a \): show \( {?thesis} \) by (fastforce simp: CFGExit.valid-node-def)

qed

next

assume \( m = \text{target}node \ a \)

show \( {?thesis} \)

proof (cases \( m = \text{Exit} \))


case True

have \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source}node \ \text{target}node \ \text{kind} \ \text{Entry} \ \text{Exit} \)

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

with \( m = \text{Exit} \): show \( {?thesis} \) by (fastforce simp: CFGExit.valid-node-def)

next

case False

with \( m = \text{target}node \ a \) ⟨ valid-edge a ⟩

have \( \text{lift-valid-edge} \ \text{valid-edge} \ \text{source}node \ \text{target}node \ \text{kind} \ \text{Entry} \ \text{Exit} \)

\((\text{Node} \ (\text{source}node \ a), \text{kind} \ a, \text{Node} \ (\text{target}node \ a))\)

by (fastforce intro: lve-edge)

with \( m = \text{target}node \ a \): show \( {?thesis} \) by (fastforce simp: CFGExit.valid-node-def)

qed

qed

note \( \text{lift-valid-node} = \text{this} \)

\{ fix \( n \ \text{as} \ n' \ \text{cs} \ m \ m' \)

assume \( \text{valid-path-aux} \ \text{cs} \ \text{as} \ \text{and} \ m \rightarrow^{*} m' \ \text{and} \ \forall c \in \text{set cs}. \ \text{valid-edge} \ c \)

\text{and} \( m \neq \text{Entry} \lor m' \neq \text{Exit} \)

hence \( \exists \text{cs'} \ \text{es}. \ \text{CFG}.\text{CFG}.\text{valid-path-aux} \ \text{kind} \)

\((\text{lift-get-return-edges} \ \text{get-return-edges} \ \text{valid-edge} \ \text{source}node \ \text{target}node \ \text{kind}) \)

\(\text{cs'} \ \text{es} \ \land \)

\(\text{list-all2} \ (\lambda c. \ c' = (\text{Node} \ (\text{source}node \ c), \text{kind} \ c, \text{Node} \ (\text{target}node \ c))) \ \text{cs} \ \text{cs'} \)

\(\land \ \text{CFG}.\text{CFG}.\text{path} \ \text{src} \ \text{try} \)

\((\text{lift-valid-edge} \ \text{valid-edge} \ \text{source}node \ \text{target}node \ \text{kind} \ \text{Entry} \ \text{Exit}) \)

\((\text{Node} \ m) \ \text{es} \ (\text{Node} \ m')\)

proof (induct arbitrary: \text{m rule: vpa-induct})

case \( \text{vpa-empty} \ \text{cs} \)

58
from \( m \rightarrow [\cdot] \rightarrow ^* m' \) have [simp]: \( m = m' \) by fastforce

from \( m \rightarrow [\cdot] \rightarrow ^* m' \) have valid-node \( m \) by (rule path-valid-node)

obtain \( cs' \) where \( cs' = \) map \( (\lambda c. \left( \text{Node (sourcenode c), kind c, Node (targetnode c)} \right)) \) \( cs \) by simp

hence list-all2

(\( \lambda c'. \, c' = \left( \text{Node (sourcenode c), kind c, Node (targetnode c)} \right) \) \( cs \) \( cs' \) by (simp add: list-all2-conv-all-nth)

with \( \langle \text{valid-node } m \rangle \) show ?case

apply (rule-tac \( x = cs' \) in exI)

apply (rule-tac \( x = [] \) in exI)

by (fastforce intro: CFGExit.empty-path lift-valid-node)

next

case (vpa-intra \( cs \) \( a \) \( as \))

note \( IH = \{ \langle m \rightarrow as \rightarrow ^* m' \rangle; \forall c \in \text{set } cs. \text{ valid-edge } c; m \neq \text{Entry} \lor m' \neq \text{Exit} \} \) =>

\( \exists cs' \) es. CFG.valid-path-aux knd

(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) \( cs' \) es \&

list-all2 (\( \lambda c', \, c' = \left( \text{Node (sourcenode c), kind c, Node (targetnode c)} \right) \) \( cs \) \( cs' \) \& CFG.path src trg

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

\( (\text{Node } m) \) es (\( \text{Node } m' \))

from \( \langle m \rightarrow a \# as \rightarrow ^* m' \rangle \) have \( m = \text{sourcenode } a \) and valid-edge \( a \)

and targetnode \( a \rightarrow as \rightarrow ^* m' \) by (auto elim: path-split-cons)

show ?case

proof (cases sourcenode \( a = \text{Entry} \land \text{targetnode } a = \text{Exit})

case True

with \( \langle m = \text{sourcenode } a \rangle; \langle m \neq \text{Entry} \lor m' \neq \text{Exit} \rangle \)

have \( m' \neq \text{Exit} \) by simp

from True have targetnode \( a = \text{Exit} \) by simp

with \( \langle \text{targetnode } a \rightarrow as \rightarrow ^* m' \rangle \) have \( m' = \text{Exit} \)

by \( \sim (\text{drule path-Exit-source, auto}) \)

with \( \langle m' \neq \text{Exit} \rangle \) have False by simp

thus ?thesis by simp

next

case False

let \( ?e = \left( \text{Node (sourcenode } a), \text{kind } a, \text{Node (targetnode } a) \right) \)

from False (valid-edge \( a \))

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit \( ?e \)

by (fastforce intro: lve-edge)

have targetnode \( a \neq \text{Entry} \)

proof

assume targetnode \( a = \text{Entry} \)

with (valid-edge \( a \)) show False by (rule Entry-target)

qed

hence targetnode \( a \neq \text{Entry} \lor m' \neq \text{Exit} \) by simp

from \( IH [OF \langle \text{targetnode } a \rightarrow as \rightarrow ^* m' \rangle; \forall c \in \text{set } cs. \text{ valid-edge } c \} \) this]

obtain \( cs' \) es

where valid-path: CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs' es

and list:list-all2
(\lambda c ', c' = \langle \text{Node (sourcenode c)}, \text{kind c}, \text{Node (targetnode c)} \rangle) cs cs'

and path:CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(\langle \text{Node (targetnode a)} \rangle) es (\langle \text{Node m'} \rangle) by blast

from (\langle \text{intra-kind} (\text{kind a}) \rangle) valid-path have CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode

targetnode kind) cs' (?e#es) by (fastforce simp:intra-kind-def)

moreover
from path (m = sourcenode a)
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit ?e)
have CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(\langle \text{Node m} \rangle) (?e#es) (\langle \text{Node m'} \rangle) by (fastforce intro:CFGExit.Cons-path)

ultimately show ?thesis using list by blast

qed

next

case (vpa-Call cs a as Q r p fs)

note IH = \langle \bigwedge m. [[m \# as\rightarrow m; \forall c \in \text{set} (a \# cs), valid-edge c; m \neq Entry \lor m' \neq Exit]] \implies
\exists cs' es. CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) cs' es \land
list-all2 (\langle \lambda c', c' = \langle \text{Node (sourcenode c)}, \text{kind c}, \text{Node (targetnode c)} \rangle \rangle
(a#cs) cs' \land CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(\langle \text{Node m} \rangle) es (\langle \text{Node m'} \rangle);

from (m = sourcenode a) have m = sourcenode a and valid-edge a

and targetnode a \# as\rightarrow m' by (auto elim: path-split-Cons)

from \forall c \in \text{set} cs. valid-edge c (\langle \text{valid-edge a} \rangle
have \forall c \in \text{set} (a \# cs). valid-edge c by simp

let ?e = \langle \text{Node (sourcenode a)}, \text{kind a}, \text{Node (targetnode a)} \rangle

have sourcenode a \neq Entry

proof

assume sourcenode a = Entry

with (\langle valid-edge a \rangle) (\text{kind a} = Q: r\rightarrow p fs)

show False by (rule Entry-no-call-source)

qed

with (\langle valid-edge a \rangle)

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

by (fastforce intro: lve-edge)

have targetnode a \neq Entry

proof

assume targetnode a = Entry

with (\langle valid-edge a \rangle) show False by (rule Entry-target)

qed

hence targetnode a \neq Entry \lor m' \neq Exit by simp

60
from IH \([OF \langle \text{targetnode} \ a \text{-} \text{as} \to \text{m} \rangle \forall c \in \text{set} \ (a \ # \ cs)\text{, valid-edge} \ c \to \text{this}]\) 

obtain \(cs' \ es\)

where valid-path:CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) cs' es

and list: list-all2
(\(\lambda c. \ c' = (\text{Node} \ (\text{sourcecode} \ c), \ \text{kind} \ c, \ \text{Node} \ (\text{targetnode} \ c))\)) (a\#cs) cs'

and path:CFG.path src try
(lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit)
(Node (targetnode a)) es (Node m') by blast

from list obtain \(cxs \ where \ cs' = cx#cxs\)

and cx:cx = (Node (sourcecode a), kind a, Node (targetnode a))

and list':list-all2
(\(\lambda c. \ c' = (\text{Node} \ (\text{sourcecode} \ c), \ \text{kind} \ c, \ \text{Node} \ (\text{targetnode} \ c))\)) cs csx
by (fastforce simp: list-all2-ConsI)

from valid-path cx \(\langle cs' = cx#cxs \rangle \ (\text{kind} \ a = Q; r \to p; fs)\)

have CFG.valid-path-aux knd
(lift-get-return-edges get-return-edges valid-edge sourcecode targetnode kind) csx (?e#es) by simp

moreover

from path \(\langle m = \text{sourcecode} \ a \rangle\)
(lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit ?e)

have CFG.path src try
(lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit)
(Node m') \((?e#es) \ (\text{Node} \ m')\) by (fastforce intro:CFGExit.Cons-path)

ultimately show \(?case \ using \ list' \ by \ blast\)

next

case \(\langle \text{vpa-ReturnEmpty} \ cs \ a \ as \ Q \ p \ f \rangle\)

note IH \(= \langle \forall m. \ [m \text{-} \text{as} \to \text{m}'; \forall c \in \text{set} \ []. \ \text{valid-edge} \ c; \ m \neq \text{Entry} \lor \ m' \neq \text{Exit} \rangle \Rightarrow \exists cs' \ es. \ CFG.\text{valid-path-aux} \ knd\)

(lift-get-return-edges get-return-edges valid-edge sourcecode targetnode kind) cs' es ∧

list-all2 (\(\lambda c. \ c' = (\text{Node} \ (\text{sourcecode} \ c), \ \text{kind} \ c, \ \text{Node} \ (\text{targetnode} \ c))\))
[\(\ [\] \ cs' \ ∧ \ CFG.\text{path} src try\)
(lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit)
(Node m) es (Node m')

from \(m \text{-} a \ # \ as \to \text{m}'\) have \(m \ = \text{sourcecode} \ a \ and \ \text{valid-edge} \ a\)

and targetnode a \(\text{-} \text{as} \to \text{m}'\) by (auto elim: path-split-Cons)

let \(\forall c = (\text{Node} \ (\text{sourcecode} \ a), \ \text{kind} a, \ \text{Node} \ (\text{targetnode} \ a))\)

have targetnode a \(\neq \text{Exit}\)

proof

assume targetnode a = Exit

with \(\langle \text{valid-edge} \ a \rangle \ (\text{kind} \ a = Q; r \to p; fs)\) show False by (rule Exit-no-return-target)

qed

with \(\langle \text{valid-edge} \ a \rangle\)

have lift-valid-edge valid-edge sourcecode targetnode kind Entry Exit ?e

by (fastforce intro: live-edge)

have targetnode a \(\neq \text{Entry}\)
proof
  assume \( \text{targetnode \ a = Entry} \)
  with \( \langle \text{valid-edge \ a} \rangle \)
  show \( \text{False} \) by (rule \( \text{Entry-target} \))
  qed
hence \( \text{targetnode \ a \neq Entry \lor \ m' \neq Exit} \) by simp
from IH[\( \text{OF \langle \text{targetnode \ a = as \rightarrow \rightarrow \ m'} \rangle \ - \ this} \)] obtain \( e \)
where valid-path:CFG.valid-path-aux knd
  (lift-get-return-edges get-return-edges valid-edge sourcenode \( \text{targetnode \ kind} \) \( \square \) \( e \))
and path:CFG.path src try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  \( \langle \text{Node \ (targetnode \ a)} \rangle \) \( e \) \( \langle \text{Node \ m'} \rangle \) by auto
from valid-path \( \langle \text{kind \ a = Q \leftarrow p f} \rangle \)
have \( \text{CFG.valid-path-aux knd} \)
  (lift-get-return-edges get-return-edges valid-edge sourcenode \( \text{targetnode \ kind} \) \( \square \) \( e \) \# \( e \))
  by simp
moreover from path \( \langle \text{m = sourcenode \ a} \rangle \)
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  \( \langle \text{Node \ (targetnode \ a)} \rangle \) \( e \) \( \langle \text{Node \ m'} \rangle \) by (fastforce intro:CFGExit.Cons-path)
ultimately show \( \text{?case using} \ (cs = []) \) by blast
next
case \( \langle \text{vpa-ReturnCons \ cs \ a = Q \leftarrow p f \ c' \ cs'} \rangle \)
  note IH = \( \langle \bigwedge m. \ [m - a \# as \rightarrow \rightarrow m'; \forall c \in \text{set} \ cs'. \text{valid-edge} \ c; \ m \neq \text{Entry} \lor \ m' \neq \text{Exit} \rangle \) \( \Rightarrow \)

  \( \exists \ csx es. \text{CFG.valid-path-aux knd} \)
  (lift-get-return-edges get-return-edges valid-edge sourcenode \( \text{targetnode \ kind} \) \( \square \) \( csx \) \( es \) \( \land \) \( \text{list-all2} \ (\lambda c' c'. \ c' = \langle \text{Node \ (sourcenode \ c)}, \text{kind} \ c, \text{Node \ (targetnode \ c)} \rangle) \)
  \( cs' \) \( csx \land \text{CFG.path src try} \)
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  \( \langle \text{Node \ m} \rangle \) \( e \) \( \langle \text{Node \ m'} \rangle \)
from \( \langle m - a \# as \rightarrow \rightarrow m'; \forall c \in \text{set} \ cs'. \text{valid-edge} \ c; \ m \neq \text{Entry} \lor \ m' \neq \text{Exit} \rangle \) \( \Rightarrow \)

  \( \exists \ csx \ es. \text{CFG.valid-path-aux knd} \)
  (lift-get-return-edges get-return-edges valid-edge sourcenode \( \text{targetnode \ kind} \) \( \square \) \( csx \) \( es \) \( \land \) \( \text{list-all2} \ (\lambda c' c'. \ c' = \langle \text{Node \ (sourcenode \ c)}, \text{kind} \ c, \text{Node \ (targetnode \ c)} \rangle) \)
  \( cs' \) \( csx \land \text{CFG.path src try} \)
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  \( \langle \text{Node \ m} \rangle \) \( es \) \( \langle \text{Node \ m'} \rangle \)
from \( \langle m - a \# as \rightarrow \rightarrow m'; \forall c \in \text{set} \ cs'. \text{valid-edge} \ c; \ m \neq \text{Entry} \lor \ m' \neq \text{Exit} \rangle \) \( \Rightarrow \)

  \( \exists \ csx \ es. \text{CFG.valid-path-aux knd} \)
  (lift-get-return-edges get-return-edges valid-edge sourcenode \( \text{targetnode \ kind} \) \( \square \) \( csx \) \( es \) \( \land \) \( \text{list-all2} \ (\lambda c' c'. \ c' = \langle \text{Node \ (sourcenode \ c)}, \text{kind} \ c, \text{Node \ (targetnode \ c)} \rangle) \)
  \( cs' \) \( csx \land \text{CFG.path src try} \)
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  \( \langle \text{Node \ m} \rangle \) \( es \) \( \langle \text{Node \ m'} \rangle \)
from \( \forall c \in \text{set} \ cs. \text{valid-edge} \ c; \ cs = c' \neq cs' \)
have valid-edge c' \( \land \forall c \in \text{set} \ cs'. \text{valid-edge} \ c \) by simp-all
let \( ?e = \langle \text{Node \ (sourcenode \ a)}, \text{kind} \ a, \text{Node \ (targetnode \ a)} \rangle \)
have targetnode \ a \neq Exit
proof
  assume targetnode \ a = Exit
  with \( \langle \text{valid-edge \ a} \rangle \) \( \langle \text{kind} \ a = Q \leftarrow p f \rangle \)
  show \( \text{False} \) by (rule \( \text{Exit-no-return-target} \))
  qed
with \( \langle \text{valid-edge \ a} \rangle \)
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit \( ?e \)
  by (fastforce intro:live-edge)
have targetnode \ a \neq Entry
proof
assume targetnode \( a = \text{Entry} \)

with \( \langle \text{valid-edge } a \rangle \) show False by (rule Entry-target)

qed

hence targetnode \( a \neq \text{Entry} \lor m' \neq \text{Exit} \) by simp

from \( \text{IH}[OF \langle \text{targetnode } a \rightarrow \ast \rangle \land \forall c \in \text{set } cs', \text{valid-edge } c \rangle \) this

obtain \( csx \) es

where valid-path:CFG.valid-path-aux knd

(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind) csx es

and list: list-all2

\( \lambda c. c' = (\text{Node (sourcenode } c), \text{kind } c, \text{Node (targetnode } c)) \) es' csx

and path:CFG.path src trg

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

(\( \text{Node (targetnode } a) \) ) es (Node \( m' \)) by blast

from \( \langle \text{valid-edge } c' \rangle \land (a \in \text{get-return-edges } c' \rangle \)

have \( ?c \in \text{lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind} \)

(Node \( \text{sourcenode } c', \text{kind } c', \text{Node (targetnode } c') \) )

by (fastforce intro: lift-get-return-edgesI)

with valid-path \( \langle \text{kind } a = Q' \rightarrow \rangle \) \( \langle \text{Node (sourcenode } c', \text{kind } c', \text{Node (targetnode } c') \rangle \# csx \) ) \( (?e \# es) \)

by simp

moreover

from \( \langle \text{path } \langle \text{m = sourcenode } a \rangle \rangle \)

(lift-valid-edge valid-edge targetnode kind Entry Exit ?e)

have CFG.path src trg

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

(Node \( m \) ) \( (?e \# es) \) (Node \( m' \) ) by (fastforce intro:CFGExit.Cons-path)

ultimately show \( ?\) case using \( \langle \text{kind } a = Q' \rightarrow \rangle \) by blast

qed

hence lift-valid-path: \( \forall m \in m', [m \rightarrow \ast \rightarrow m' \land m \neq \text{Entry} \lor m' \neq \text{Exit}] \)

\( \Rightarrow \exists es, \text{CFG.CFG.valid-path'} src trg knd \)

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

(lift-get-return-edges get-return-edges valid-edge targetnode kind)

(Node \( m \) ) es (Node \( m' \) )

by (fastforce simp:vp-def valid-path-def CFGExit.vp-def CFGExit.valid-path-def)

show ?thesis

proof

fix \( n \) assume CFG.CFG.valid-node src trg

(lift-valid-edge valid-edge targetnode kind Entry Exit) \( n \)

hence \( (n = \text{NewEntry}) \lor n = \text{NewExit} \lor (\exists m. n = \text{Node } m \land \text{valid-node } m) \)

63
by (auto elim: lift-valid-edge.cases simp: CFGExit.valid-node-def)

thus \exists \cdot CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
NewEntry as n apply —

proof (erule disjE)+

assume n = NewEntry
henceCFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
NewEntry [] n
by (fastforce intro: CFGExit.empty-path
simp: CFGExit.vp-def CFGExit.valid-path-def)

thus ?thesis by blast

next

assume n = NewExit

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

hence CFG.CFG.path src trg
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
NewEntry [(NewEntry,(\lambda s. False) \_, NewExit)] NewExit
by (fastforce dest: CFGExit.path-edge)

with (n = NewExit) have CFG.CFG.valid-path’ src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
NewEntry [(NewEntry,(\lambda s. False) \_, NewExit)] n
by (fastforce simp: CFGExit.vp-def CFGExit.valid-path-def)

thus ?thesis by blast

next

assume \exists \cdot m. n = Node m \land valid-node m
then obtain m where n = Node m and valid-node m by blast
from (valid-node m)

show ?thesis

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

  case Entry

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

  with (m = Entry) (n = Node m) have CFG.CFG.path src trg
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  NewEntry [(NewEntry,(\lambda s. True) \_, Node Entry)] n
  by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
      simp: CFGExit.valid-node-def)

  thus ?thesis by (fastforce simp: CFGExit.vp-def CFGExit.valid-path-def)

next

  case Exit

  from inner obtain ax where valid-edge ax and intra-kind (kind ax)
  and inner-node (sourcenode ax)
  and targetnode ax = Exit by (erule inner-node-Exit-edge)

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

hence CFG.path src try
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(Node (source-node ax)) [(Node (source-node ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
  simp: CFGExit.valid-node-def)

with (intra-kind (kind ax))

have slp-edge: CFG_CFG.same-level-path src try knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind)
(Node (source-node ax)) [(Node (source-node ax), kind ax, Node Exit)]
(Node Exit)
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def
  intra-kind-def)

have source-node ax ≠ Exit
proof

assume source-node ax = Exit
  with ⟨valid-edge ax⟩ show False by (rule Exit-source)
qed

have
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(Node Entry) by (fastforce intro: lve-Entry-edge)

hence CFG.path src try
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(Node Entry) [(Node Entry, λs. True), Node Entry)] (Node Entry)
by (fastforce intro: CFGExit.Cons-path CFGExit.empty-path
  simp: CFGExit.valid-node-def)

hence slp-edge': CFG_CFG.same-level-path src try knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind)
(Node Entry) [(Node Entry, λs. True), Node Entry)] (Node Entry)
by (fastforce simp: CFGExit.slp-def CFGExit.same-level-path-def)

from (inner-node (source-node ax)) have valid-node (source-node ax)
by (rule inner-is-valid)
then obtain asx where Entry − asx → * source-node ax
by (fastforce dest: Entry-path)
with (source-node ax ≠ Exit)

have ∃ es. CFG_CFG.valid-path src try knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind) (Node Entry) es (Node (source-node ax))
by (fastforce intro: lift-valid-path)
then obtain es where CFG_CFG.valid-path' src try knd
(lift-valid-edge valid-edge source-node target-node kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge source-node
target-node kind) (Node Entry) es (Node (source-node ax)) by blast
with slp-edge have CFG.CFG.valid-path' src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind)
(Node Entry) (es@[[Node (sourcenode ax),kind ax,Node Exit]]) (Node Exit)
by -(rule CFGExit.ep-slp-Append)
with slp-edge' have CFG.CFG.valid-path' src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) NewEntry
  (es@[(Node (sourcenode ax),kind ax,Node Exit)]) (Node Exit)
by (rule CFGExit.slp-vp-Append)
with \(m = \text{Exit}\) \(n = \text{Node m}\) show \(?\text{thesis}\) by simp blast
next
case inner
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  (NewEntry,(\(\lambda s \cdot \text{True}\)) \(\sqrt{\ast}\),Node Entry) by (fastforce intro:ive-Entry-edge)
hence CFG.path src try
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (NewEntry) [(NewEntry,(\(\lambda s \cdot \text{True}\)) \(\sqrt{\ast}\),Node Entry)] (Node Entry)
by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path
  simp:CFGExit.valid-node-def)
hence slp-edge:CFG.CFG.same-level-path' src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind)
  (NewEntry) [(NewEntry,(\(\lambda s \cdot \text{True}\)) \(\sqrt{\ast}\),Node Entry)] (Node Entry)
by (fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def)
from \(\text{valid-node m}\) obtain as where Entry \(\rightarrow \ast\) \(\ast\) m
by (fastforce dest:Entry-path)
with \(\text{inner-node m}\)
have \(\exists es. \ CFG.CFG.valid-path' src trg knd\)
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node Entry) es (Node m)
by (fastforce intro:lift-valid-path simp:inner-node-def)
then obtain es where \(\ CFG.CFG.valid-path' src trg knd\)
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node Entry) es (Node m) by blast
with slp-edge have CFG.CFG.valid-path' src trg knd
  (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
  (lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) NewEntry [(NewEntry,(\(\lambda s \cdot \text{True}\)) \(\sqrt{\ast}\),Node Entry)]@es
(Node m)
by (rule CFGExit.slp-vp-Append)
with \(n = \text{Node m}\) show \(?\text{thesis}\) by simp blast
qed
qed

next

**fix** $n$ **assume** $\text{CFG.CFG.valid-node src try}$

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

**hence** ($(n = \text{NewEntry}) \lor n = \text{NewExit}) \lor (\exists m. n = \text{Node m} \land \text{valid-node m})$

by(auto elim:lift-valid-edge.cases simp:CFGExit.valid-node-def)

thus $3 \text{as. CFG.CFG.valid-path'} src try knd$

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

(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)

$n$ as $\text{NewExit}$ **apply**

proof(erule disjE)+

**assume** $n = \text{NewEntry}$

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

$(\text{NewEntry},(\lambda s. \text{False}),\text{NewExit})$ by(fastforce intro:lve-Entry-Exit-edge)

**hence** $\text{CFG.CFG.path src try}$

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

$\text{NewEntry} [(\text{NewEntry},(\lambda s. \text{False}),\text{NewExit} ) ] \text{NewExit}$

by(fastforce dest:CFGExit.path-edge)

with $(n = \text{NewEntry})$ **have** $\text{CFG.CFG.valid-path' src try knd}$

(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind Entry Exit)

(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind Entry Exit)

$n [(\text{NewEntry},(\lambda s. \text{False}),\text{NewExit} ) ] \text{NewExit}$

by(fastforce simp:CFGExit.vp-def CFGExit.valid-path-def)

thus $?\text{thesis}$ by blast

next

**assume** $n = \text{NewExit}$

**hence** $\text{CFG.CFG.valid-path'} src try knd$

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

(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind Entry Exit)

$n [ ] \text{NewExit}$

by(fastforce intro:CFGExit.empty-path

simp:CFGExit.vp-def CFGExit.valid-path-def)

thus $?\text{thesis}$ by blast

next

**assume** $\exists m. n = \text{Node m} \land \text{valid-node m}$

then obtain $m$ **where** $n = \text{Node m}$ and valid-node $m$ by blast

from (valid-node m)

show $?\text{thesis}$

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

case Entry

from inner obtain $ax$ where valid-edge $ax$ and intra-kind (kind $ax$)

and inner-node (targetnode $ax$) and sourcenode $ax = \text{Entry}$

by(erule inner-node-Entry-edge)

**hence** lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit

$(\text{Node Entry},\text{kind ax},\text{Node} (\text{targetnode ax}) )$

by(auto intro:lift-valid-edge.lve-edge simp:inner-node-def)

**hence** $\text{CFG.path src try}$

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

67
\[(\text{Node Entry})\ [(\text{Node Entry}.\text{kind ax},\text{Node } (\text{targetnode ax}))]\]
\[(\text{Node } (\text{targetnode ax}))\]
by (fastforce intro : \text{CFGExit} . \text{Cons-path} \text{CFGExit} . \text{empty-path}
\text{simp:CFGExit} . \text{valid-node-def})
\text{with (intra-kind (kind ax))}
\text{have slp-edge:CFG.CFG\_same-level-path'} src trg knd
\text{(lift-valid-edge valid-edge source-node target-node kind Entry Exit)}
\text{(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)}
\[(\text{Node Entry})\ [(\text{Node Entry}.\text{kind ax},\text{Node } (\text{targetnode ax}))]\]
\[(\text{Node } (\text{targetnode ax}))\]
by (fastforce simp: \text{CFGExit} . \text{slp-def} \text{CFGExit} . \text{same-level-path-def}
\text{intra-kind-def})
\text{have targetnode ax \neq Entry}
\text{proof}
\text{assume targetnode ax = Entry}
\text{with (valid-edge ax) show False by (rule Entry-target)}
\text{qed}
\text{have slp-edge:CFG.CFG\_same-level-path' src trg knd}
\text{(lift-valid-edge valid-edge source-node target-node kind Entry Exit)}
\text{(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)}
\[(\text{Node Entry})\ [(\text{Node Entry}.\text{kind ax},\text{Node } (\text{targetnode ax}))]\]
\[(\text{Node } (\text{targetnode ax}))\]
by (fastforce intro : \text{CFGExit} . \text{Cons-path} \text{CFGExit} . \text{empty-path}
\text{simp:CFGExit} . \text{valid-node-def})
\text{hence slp-edge' :CFG.CFG\_same-level-path' src trg knd}
\text{(lift-valid-edge valid-edge source-node target-node kind Entry Exit)}
\text{(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)}
\[(\text{Node Exit})\ [(\text{Node Exit}.\text{entry},\text{NewExit})] \text{NewExit}
\text{by (fastforce intro : \text{CFGExit} . \text{Cons-path} \text{CFGExit} . \text{empty-path}
\text{simp:CFGExit} . \text{valid-node-def})}
\text{hence slp-edge' :CFG.CFG\_same-level-path' src trg knd}
\text{(lift-valid-edge valid-edge source-node target-node kind Entry Exit)}
\text{(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)}
\[(\text{Node Exit})\ [(\text{Node Exit}.\text{entry},\text{NewExit})] \text{NewExit}
\text{by (fastforce intro : \text{CFGExit} . \text{Cons-path} \text{CFGExit} . \text{empty-path}
\text{simp:CFGExit} . \text{valid-node-def})}
\text{from (inner-node (targetnode ax)) have valid-node (targetnode ax)}
\text{by (rule inner-is-valid)}
\text{then obtain asx where targetnode ax \rightarrow asx \star Exit}
\text{by (fastforce dest : \text{Exit-path})}
\text{with (targetnode ax \neq Entry)}
\text{have \exists es. CFG.CFG\_valid-path' src trg knd}
\text{(lift-valid-edge valid-edge source-node target-node kind Entry Exit)}
\text{(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)}
\text{(Node (targetnode ax)) es (Node Exit)}
\text{by (fastforce intro : \text{lift-valid-path})}
\text{then obtain es where CFG.CFG\_valid-path' src trg knd}
\text{(lift-valid-edge valid-edge source-node target-node kind Entry Exit)}
\text{(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)}
\text{(Node (targetnode ax)) es (Node Exit) by blast}
\text{with slp-edge have CFG.CFG\_valid-path' src trg knd}
\text{(lift-valid-edge valid-edge source-node target-node kind Entry Exit)}
\text{(lift-get-return-edges get-return-edges valid-edge source-node target-node kind)}
(Node Entry) (((Node Entry,kind ax,Node (targetnode ax)])@es) (Node Exit)

by (rule CFGExit.slp-vp-Append)

with slp-edge' have CFG.CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node Entry)
(((Node Entry,kind ax,Node (targetnode ax)])@es)@
[(Node Exit,(λs. True),NewExit)]) NewExit
by − (rule CFGExit.vp-slp-Append)

with (m = Entry) (n = Node m) show ?thesis by simp blast

next
case Exit

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(Node Exit,(λs. True),NewExit) by (fastforce intro:ltv-Exit-edge)

with (m = Exit) (n = Node m) have CFG.CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
n (Node Exit,(λs. True),NewExit) NewExit

by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path
simp:CFGExit.valid-node-def)

thus ?thesis by (fastforce simp:CFGExit.vp-def CFGExit.valid-path-def)

next
case inner

have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
(Node Exit,(λs. True),NewExit) by (fastforce intro:ltv-Exit-edge)

hence CFG.path src try
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(Node Exit) (Node Exit,(λs. True),NewExit) NewExit

by (fastforce intro:CFGExit.Cons-path CFGExit.empty-path
simp:CFGExit.valid-node-def)

hence slp-edge:CFG.CFG.same-level-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind)
(Node Exit) (Node Exit,(λs. True),NewExit) NewExit

by (fastforce simp:CFGExit.slp-def CFGExit.same-level-path-def)

from (valid-node m) obtain as where m −→ √∗ Exit
by (fastforce dest:Exit-path)

with (inner-node m)

have ∃ es, CFG.CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node m) es (Node Exit)

by (fastforce intro:ltv-valid-path simp:inner-node-def)

then obtain es where CFG.CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node m) es (Node Exit) by blast

with slp-edge have CFG.CFG.valid-path' src try knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit)
(lift-get-return-edges get-return-edges valid-edge sourcenode
targetnode kind) (Node m) (es@[[(Node Exit,(λs. True) \_\_\_,NewExit)]) NewExit
by (−(rule CFGExit.op-slp-Append)
with \(n = Node m\) show \(\text{thesis by simp blast}\)
qed
qed
next
fix \(n n'\)
assume method-exit1::CFGExit.CFGExit.method-exit src knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n
and method-exit2::CFGExit.CFGExit.method-exit src knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit n'
and lift-eq::lift-get-proc get-proc Main n = lift-get-proc get-proc Main n'
from method-exit1 show \(n = n'\)
proof (rule CFGExit.method-exit-cases)
assume \(n = NewExit\)
from method-exit2 show \(\text{thesis}\)
proof (rule CFGExit.method-exit-cases)
assume \(n' = NewExit\)
with \(n = NewExit\) show \(\text{thesis by simp}\)
next
fix \(a Q f p\)
assume \(n' = src a\)
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q \_ \_ \_<\_pf
hence lift-get-proc get-proc Main (src a) = p
by (−(rule CFGExit.get-proc-return)
with CFGExit.get-proc-Exit lift-eq (n' = src a) (n = NewExit):
have \(p = Main\) by simp
with \(\text{knd a = Q \_ \_ \_<\_pf}\)
have \(\text{knd a = Q \_ \_ \_<\_pf}\) by simp
with \(\text{lifl-valid-edge valid-edge sourcenode targetnode kind Entry Exit a}\)
have False by (rule CFGExit.Main-no-return-source)
thus \(\text{thesis by simp}\)
qed
next
fix \(a Q f p\)
assume \(n = src a\)
and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
and knd a = Q \_ \_ \_<\_pf
then obtain \(x\) where valid-edge \(x\) and \(src a = \text{Node (sourcenode x)}\)
and \(\text{knd x = Q \_ \_ \_<\_pf}\)
by (fastforce elim::lift-valid-edge.cases)
hence method-exit (sourcenode x) by (fastforce simp::method-exit-def)
from method-exit2 show \(\text{thesis}\)
proof (rule CFGExit.method-exit-cases)
assume \(n' = NewExit\)
from \(\text{lifl-valid-edge valid-edge sourcenode targetnode kind Entry Exit a}\)
\(\langle\text{knd a = Q \_ \_ \_<\_pf}\rangle\)
have lift-get-proc get-proc Main (src a) = p
  by -(rule CFGExit.get-proc-return)
with CFGExit.get-proc-Exit lift-eq (n = src a) (n' = NewExit)
have p = Main by simp
with (knd a = Q←p'f') have knd a = Q←Main' by simp
with (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a)
have False by (rule CFGExit.Main-no-return-source)
thus ?thesis by simp
next
fix a' Q' f' p'
assume n' = src a'
  and lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a'
  and knd a' = Q'←p'f'
then obtain x' where valid-edge x' and src a' = Node (sourcenode x')
  and kind x' = Q'←p'f'
  by (fastforce elim:lift-valid-edge.cases)
hence method-exit (sourcenode x') by (fastforce simp:method-exit-def)
with (method-exit (sourcenode x)) lift-eq (n = src a) (n' = src a')
  (src a = Node (sourcenode x)) (src a' = Node (sourcenode x'))
have sourcenode x = sourcenode x' by (fastforce intro:method-exit-unique)
with (src a = Node (sourcenode x)) (src a' = Node (sourcenode x'))
  (n = src a) (n' = src a')
show ?thesis by simp
qed
qed
qed

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

3.2.3 Low-deterministic security via the lifted graph

lemma Lift-NonInterferenceGraph:
fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
and get-proc and get-return-edges and procs and Main
and Def and Use and ParamDefs and ParamUses and H and L
defines \text{lve}::\text{lve} \equiv \text{lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit}
and \text{lget-proc}::\text{lget-proc} \equiv \text{lift-get-proc get-proc Main}
and \text{lget-return-edges}::\text{lget-return-edges} \equiv
\text{lift-get-return-edges valid-edge sourcenode targetnode kind}
and \text{lDef}::\text{lDef} \equiv \text{lift-Def Def Entry Exit H L}
and \text{lUse}::\text{lUse} \equiv \text{lift-Use Use Entry Exit H L}
and \text{lParamDefs}::\text{lParamDefs} \equiv \text{lift-ParamDefs ParamDefs}
and \text{lParamUses}::\text{lParamUses} \equiv \text{lift-ParamUses ParamUses}
assumes SDG:SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main NewExit Def Use ParamDefs ParamUses
and inner::CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit \forall
and H \cap L = \{\} and H \cup L = \text{UNIV}
shows NonInterferenceInterGraph src trg knd lve NewEntry lget-proc
lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L
(Node Entry) (Node Exit)
proof –
interpret SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main NewExit Def Use ParamDefs ParamUses
by\text{\textbf{(rule SDG)}}
interpret SDG':SDG src trg knd lve NewEntry lget-proc lget-return-edges
procs Main NewExit lDef lUse lParamDefs lParamUses
by\text{\textbf{(fastforce intro:lift-SDG SDG inner simp:lve lget-proc lget-return-edges lDef lUse lParamDefs lParamUses)}}
show ?thesis
proof
\text{fix a assume lve a and src a = NewEntry}
\text{thus try a = NewExit \lor try a = Node Entry}
by (fastforce elim: lift-valid-edge.cases simp:lve)
next
show ∃ a. lve a ∧ src a = NewEntry ∧ trg a = Node Entry ∧ knd a = (λs. True)
  by (fastforce intro: lve-Entry-edge simp:lve)
next
fix a assume lve a and trg a = Node Entry
from ⟨lve a⟩
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  by (simp add:lve)
from this (trg a = Node Entry)
show src a = NewEntry
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨trg e = Node Entry⟩
have targetnode a = Entry by simp
  with ⟨valid-edge a⟩ have False by (rule Entry-target)
  thus ?case by simp
qed simp-all
next
fix a assume lve a and trg a = NewExit
thus src a = NewEntry ∨ src a = Node Exit
  by (fastforce elim: lift-valid-edge.cases simp:lve)
next
show ∃ a. lve a ∧ src a = Node Exit ∧ trg a = NewExit ∧ knd a = (λs. True)
  by (fastforce intro: lve-Exit-edge simp:lve)
next
fix a assume lve a and src a = Node Exit
from ⟨lve a⟩
have lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit a
  by (simp add:lve)
from this ⟨src a = Node Exit⟩
show trg a = NewExit
proof (induct rule: lift-valid-edge.induct)
case (lve-edge a e)
  from ⟨e = (Node (sourcenode a), kind a, Node (targetnode a))⟩
  ⟨src e = Node Exit⟩
have sourcenode a = Exit by simp
  with ⟨valid-edge a⟩ have False by (rule Exit-source)
  thus ?case by simp
qed simp-all
next
from lDef show lDef (Node Entry) = H
  by (fastforce elim: lift-Def-set.cases intro: lift-Def-High)
next
from Entry-noteq-Exit lUse show lUse (Node Entry) = H
  by (fastforce elim: lift-Use-set.cases intro: lift-Use-High)
next
from Entry-noteq-Exit lUse show lUse (Node Exit) = L
by (fastforce elim:lift-Usecases intro:lift-Use-Low)
next
from :H ∩ L = {} show H ∩ L = {} .
next
from :H ∪ L = UNIV show H ∪ L = UNIV .
qed
qed
end

References


