Towards Certified Slicing

Daniel Wasserrab

March 12, 2013

Abstract

Slicing is a widely-used technique with applications in e.g. compiler technology and software security. Thus verification of algorithms in these areas is often based on the correctness of slicing, which should ideally be proven independent of concrete programming languages and with the help of well-known verifying techniques such as proof assistants. As a first step in this direction, this contribution presents a framework for dynamic [2] and static intraprocedural slicing [1] based on control flow and program dependence graphs. Abstracting from concrete syntax we base the framework on a graph representation of the program fulfilling certain structural and well-formedness properties.

We provide two instantiations to show the validity of the framework: a simple While language and the sophisticated object-oriented byte code language from Jinja [3].

0.1 Auxiliary lemmas

theory *AuxLemmas* **imports** *Main* **begin**

abbreviation *arbitrary* == *undefined*

Lemmas about left- and rightmost elements in lists

lemma *leftmost-element-property*:
assumes $\exists x \in set\ xs.\ P\ x$
obtains $zs\ x'\ ys$ **where** $xs = zs@x'\#ys$ **and** $P\ x'$ **and** $\forall z \in set\ zs.\ \neg\ P\ z$
⟨*proof*⟩

lemma *rightmost-element-property*:
assumes $\exists x \in set\ xs.\ P\ x$
obtains $ys\ x'\ zs$ **where** $xs = ys@x'\#zs$ **and** $P\ x'$ **and** $\forall z \in set\ zs.\ \neg\ P\ z$
⟨*proof*⟩

Lemma concerning maps and @

lemma *map-append-append-maps*:
  assumes *map*:*map f xs* = *ys*@*zs*
  obtains *xs′ xs″* **where** *map f xs′* = *ys* **and** *map f xs″* = *zs* **and** *xs*=*xs′*@*xs″*
⟨*proof*⟩

Lemma concerning splitting of *list*s

lemma *path-split-general*:
assumes *all*:∀ *zs*. *xs* ≠ *ys*@*zs*
obtains *j zs* **where** *xs* = (*take j ys*)@*zs* **and** *j* < *length ys*
  and ∀ *k* > *j*. ∀ *zs′*. *xs* ≠ (*take k ys*)@*zs′*
⟨*proof*⟩

end

Chapter 1

The Framework

As slicing is a program analysis that can be completely based on the information given in the CFG, we want to provide a framework which allows us to formalize and prove properties of slicing regardless of the actual programming language. So the starting point for the formalization is the definition of an abstract CFG, i.e. without considering features specific for certain languages. By doing so we ensure that our framework is as generic as possible since all proofs hold for every language whose CFG conforms to this abstract CFG. This abstract CFG can be used as a basis for static intraprocedural slicing as well as for dynamic slicing, if in the dynamic case all method calls are inlined (i.e., abstract CFG paths conform to traces).

1.1 Basic Definitions

theory *BasicDefs* **imports** *AuxLemmas* **begin**

1.1.1 Edge kinds

datatype $'state\ edge\text{-}kind = Update\ 'state \Rightarrow 'state$ $(\Uparrow\text{-})$
$\quad | \ Predicate\ 'state \Rightarrow bool$ $('(\text{-}')_{\surd})$

1.1.2 Transfer and predicate functions

fun $transfer :: 'state\ edge\text{-}kind \Rightarrow 'state \Rightarrow 'state$
where $transfer\ (\Uparrow f)\ s = f\ s$
$\quad | \ transfer\ (P)_{\surd}\ s \quad = s$

fun $transfers :: 'state\ edge\text{-}kind\ list \Rightarrow 'state \Rightarrow 'state$
where $transfers\ []\ s \quad = s$
$\quad | \ transfers\ (e\#es)\ s = transfers\ es\ (transfer\ e\ s)$

fun $pred :: 'state\ edge\text{-}kind \Rightarrow 'state \Rightarrow bool$
where $pred\ (\Uparrow f)\ s = True$
$\quad | \ pred\ (P)_{\surd}\ s \quad = (P\ s)$

fun $preds :: 'state\ edge\text{-}kind\ list \Rightarrow 'state \Rightarrow bool$

where *preds* [] *s* = *True*
| *preds* (*e*#*es*) *s* = (*pred e s* ∧ *preds es* (*transfer e s*))

lemma *transfers-split*:
(*transfers* (*ets*@*ets'*) *s*) = (*transfers ets'* (*transfers ets s*))
⟨*proof*⟩

lemma *preds-split*:
(*preds* (*ets*@*ets'*) *s*) = (*preds ets s* ∧ *preds ets'* (*transfers ets s*))
⟨*proof*⟩

lemma *transfers-id-no-influence*:
transfers [*et* ← *ets*. *et* ≠ ↑*id*] *s* = *transfers ets s*
⟨*proof*⟩

lemma *preds-True-no-influence*:
preds [*et* ← *ets*. *et* ≠ (λ*s*. *True*)$_{\surd}$] *s* = *preds ets s*
⟨*proof*⟩

end

1.2 CFG

theory *CFG* **imports** *BasicDefs* **begin**

1.2.1 The abstract CFG

locale *CFG* =
fixes *sourcenode* :: *'edge* ⇒ *'node*
fixes *targetnode* :: *'edge* ⇒ *'node*
fixes *kind* :: *'edge* ⇒ *'state edge-kind*
fixes *valid-edge* :: *'edge* ⇒ *bool*
fixes *Entry*::*'node* (*'*(*'-Entry'-*)*)
assumes *Entry-target* [*dest*]: ⟦*valid-edge a*; *targetnode a* = (*-Entry-*)⟧ ⟹ *False*
and *edge-det*:
⟦*valid-edge a*; *valid-edge a'*; *sourcenode a* = *sourcenode a'*;
targetnode a = *targetnode a'*⟧ ⟹ *a* = *a'*

begin

definition *valid-node* :: *'node* ⇒ *bool*
where *valid-node n* ≡
(∃ *a*. *valid-edge a* ∧ (*n* = *sourcenode a* ∨ *n* = *targetnode a*))

lemma [*simp*]: *valid-edge a* ⟹ *valid-node* (*sourcenode a*)

⟨*proof*⟩

lemma [*simp*]: *valid-edge a* ⟹ *valid-node* (*targetnode a*)
⟨*proof*⟩

1.2.2 CFG paths and lemmas

inductive *path* :: *'node* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(- −-→∗ - [*51*,*0*,*0*] *80*)
where
empty-path:*valid-node n* ⟹ *n* −[]→∗ *n*

| *Cons-path*:
⟦*n''* −*as*→∗ *n'*; *valid-edge a*; *sourcenode a* = *n*; *targetnode a* = *n''*⟧
⟹ *n* −*a*#*as*→∗ *n'*

lemma *path-valid-node*:
assumes *n* −*as*→∗ *n'* **shows** *valid-node n* **and** *valid-node n'*
⟨*proof*⟩

lemma *empty-path-nodes* [*dest*]:*n* −[]→∗ *n'* ⟹ *n* = *n'*
⟨*proof*⟩

lemma *path-valid-edges*:*n* −*as*→∗ *n'* ⟹ ∀ *a* ∈ *set as*. *valid-edge a*
⟨*proof*⟩

lemma *path-edge*:*valid-edge a* ⟹ *sourcenode a* −[*a*]→∗ *targetnode a*
⟨*proof*⟩

lemma *path-Entry-target* [*dest*]:
assumes *n* −*as*→∗ (-*Entry*-)
shows *n* = (-*Entry*-) **and** *as* = []
⟨*proof*⟩

lemma *path-Append*:⟦*n* −*as*→∗ *n''*; *n''* −*as'*→∗ *n'*⟧
⟹ *n* −*as*@*as'*→∗ *n'*
⟨*proof*⟩

lemma *path-split*:
assumes *n* −*as*@*a*#*as'*→∗ *n'*
shows *n* −*as*→∗ *sourcenode a* **and** *valid-edge a* **and** *targetnode a* −*as'*→∗ *n'*
⟨*proof*⟩

lemma *path-split-Cons*:
assumes $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
obtains $a'\ as'$ **where** $as = a'\#as'$ **and** $n = sourcenode\ a'$
and *valid-edge* a' **and** $targetnode\ a'\ -as'\rightarrow*\ n'$
⟨*proof*⟩

lemma *path-split-snoc*:
assumes $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
obtains $a'\ as'$ **where** $as = as'@[a']$ **and** $n\ -as'\rightarrow*\ sourcenode\ a'$
and *valid-edge* a' **and** $n' = targetnode\ a'$
⟨*proof*⟩

lemma *path-split-second*:
assumes $n\ -as@a\#as'\rightarrow*\ n'$ **shows** $sourcenode\ a\ -a\#as'\rightarrow*\ n'$
⟨*proof*⟩

lemma *path-Entry-Cons*:
assumes (*-Entry-*) $-as\rightarrow*\ n'$ **and** $n' \neq$ (*-Entry-*)
obtains $n\ a$ **where** $sourcenode\ a =$ (*-Entry-*) **and** $targetnode\ a = n$
and $n\ -tl\ as\rightarrow*\ n'$ **and** *valid-edge* a **and** $a = hd\ as$
⟨*proof*⟩

lemma *path-det*:
$[\![n\ -as\rightarrow*\ n'; n\ -as\rightarrow*\ n'']\!] \Longrightarrow n' = n''$
⟨*proof*⟩

definition
$sourcenodes :: 'edge\ list \Rightarrow 'node\ list$
where $sourcenodes\ xs \equiv map\ sourcenode\ xs$

definition
$kinds :: 'edge\ list \Rightarrow 'state\ edge\text{-}kind\ list$
where $kinds\ xs \equiv map\ kind\ xs$

definition
$targetnodes :: 'edge\ list \Rightarrow 'node\ list$
where $targetnodes\ xs \equiv map\ targetnode\ xs$

lemma *path-sourcenode*:
$[\![n\ -as\rightarrow*\ n'; as \neq []]\!] \Longrightarrow hd\ (sourcenodes\ as) = n$
⟨*proof*⟩

lemma *path-targetnode*:
$[\![n\ -as\rightarrow*\ n';\ as \neq []]\!] \Longrightarrow last\ (targetnodes\ as) = n'$
⟨*proof*⟩

lemma *sourcenodes-is-n-Cons-butlast-targetnodes*:
$[\![n\ -as\rightarrow*\ n';\ as \neq []]\!] \Longrightarrow$
$sourcenodes\ as = n\#(butlast\ (targetnodes\ as))$
⟨*proof*⟩

lemma *targetnodes-is-tl-sourcenodes-App-n'*:
$[\![n\ -as\rightarrow*\ n';\ as \neq []]\!] \Longrightarrow$
$targetnodes\ as = (tl\ (sourcenodes\ as))@[n']$
⟨*proof*⟩

lemma *Entry-sourcenode-hd*:
assumes $n\ -as\rightarrow*\ n'$ **and** $(\text{-}Entry\text{-}) \in set\ (sourcenodes\ as)$
shows $n = (\text{-}Entry\text{-})$ **and** $(\text{-}Entry\text{-}) \notin set\ (sourcenodes\ (tl\ as))$
⟨*proof*⟩

end

end

theory *CFGExit* **imports** *CFG* **begin**

1.2.3 Adds an exit node to the abstract CFG

locale *CFGExit* = *CFG sourcenode targetnode kind valid-edge Entry*
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'*(*'-Entry'-*')) +
fixes *Exit*::*'node* (*'*(*'-Exit'-*'))
assumes *Exit-source* [*dest*]: $[\![valid\text{-}edge\ a;\ sourcenode\ a = (\text{-}Exit\text{-})]\!] \Longrightarrow False$
and *Entry-Exit-edge*: $\exists a.\ valid\text{-}edge\ a \wedge sourcenode\ a = (\text{-}Entry\text{-}) \wedge$
$targetnode\ a = (\text{-}Exit\text{-}) \wedge kind\ a = (\lambda s.\ False)_{\surd}$

begin

lemma *Entry-noteq-Exit* [*dest*]:
assumes *eq*:$(\text{-}Entry\text{-}) = (\text{-}Exit\text{-})$ **shows** *False*
⟨*proof*⟩

lemma *Exit-noteq-Entry* [*dest*]:$(\text{-}Exit\text{-}) = (\text{-}Entry\text{-}) \Longrightarrow False$

⟨*proof*⟩

lemma [*simp*]: *valid-node* (*-Entry-*)
⟨*proof*⟩

lemma [*simp*]: *valid-node* (*-Exit-*)
⟨*proof*⟩

definition *inner-node* :: *'node* ⇒ *bool*
  where *inner-node-def*:
  inner-node n ≡ *valid-node n* ∧ *n* ≠ (*-Entry-*) ∧ *n* ≠ (*-Exit-*)

lemma *inner-is-valid*:
  inner-node n ⟹ *valid-node n*
⟨*proof*⟩

lemma [*dest*]:
  inner-node (*-Entry-*) ⟹ *False*
⟨*proof*⟩

lemma [*dest*]:
  inner-node (*-Exit-*) ⟹ *False*
⟨*proof*⟩

lemma [*simp*]:⟦*valid-edge a*; *targetnode a* ≠ (*-Exit-*)⟧
  ⟹ *inner-node* (*targetnode a*)
  ⟨*proof*⟩

lemma [*simp*]:⟦*valid-edge a*; *sourcenode a* ≠ (*-Entry-*)⟧
  ⟹ *inner-node* (*sourcenode a*)
  ⟨*proof*⟩

lemma *valid-node-cases* [*consumes 1*, *case-names Entry Exit inner*]:
  ⟦*valid-node n*; *n* = (*-Entry-*) ⟹ *Q*; *n* = (*-Exit-*) ⟹ *Q*;
    inner-node n ⟹ *Q*⟧ ⟹ *Q*
⟨*proof*⟩

lemma *path-Exit-source* [*dest*]:
  assumes (*-Exit-*) −*as*→∗ *n'* **shows** *n'* = (*-Exit-*) **and** *as* = []
  ⟨*proof*⟩

lemma *Exit-no-sourcenode*[*dest*]:
  assumes *isin*:(*-Exit-*) ∈ *set* (*sourcenodes as*) **and** *path*:*n* −*as*→∗ *n'*
  shows *False*
⟨*proof*⟩

end

end

1.3 Postdomination

theory *Postdomination* **imports** *CFGExit* **begin**

1.3.1 Standard Postdomination

locale *Postdomination* = *CFGExit sourcenode targetnode kind valid-edge Entry Exit*
  for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
  and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
  and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Exit* :: *'node* (*'('-Exit'-')*) +
  assumes *Entry-path*:*valid-node n* ⟹ ∃ *as*. (*-Entry-*) −*as*→∗ *n*
  and *Exit-path*:*valid-node n* ⟹ ∃ *as*. *n* −*as*→∗ (*-Exit-*)

begin

definition *postdominate* :: *'node* ⇒ *'node* ⇒ *bool* (*- postdominates -* [*51*,*0*])
where *postdominate-def*:*n' postdominates n* ≡
    (*valid-node n* ∧ *valid-node n'* ∧
    ((∀ *as*. *n* −*as*→∗ (*-Exit-*) ⟶ *n'* ∈ *set* (*sourcenodes as*))))

lemma *postdominate-implies-path*:
  assumes *n' postdominates n* **obtains** *as* **where** *n* −*as*→∗ *n'*
⟨*proof*⟩

lemma *postdominate-refl*:
  assumes *valid*:*valid-node n* **and** *notExit*:*n* ≠ (*-Exit-*)
  shows *n postdominates n*
⟨*proof*⟩

lemma *postdominate-trans*:
  assumes *pd1*:*n''* *postdominates n* **and** *pd2*:*n'* *postdominates n''*
  shows *n'* *postdominates n*
⟨*proof*⟩

lemma *postdominate-antisym*:
  assumes *pd1*:*n'* *postdominates n* **and** *pd2*:*n postdominates n'*
  shows *n* = *n'*

⟨*proof*⟩

lemma *postdominate-path-branch*:
assumes $n\ -as\rightarrow*\ n''$ **and** n' *postdominates* n'' **and** $\neg\ n'$ *postdominates* n
obtains $a\ as'\ as''$ **where** $as = as'@a\#as''$ **and** *valid-edge* a
and $\neg\ n'$ *postdominates* (*sourcenode* a) **and** n' *postdominates* (*targetnode* a)
⟨*proof*⟩

lemma *Exit-no-postdominator*:
(*-Exit-*) *postdominates* $n \Longrightarrow$ *False*
⟨*proof*⟩

lemma *postdominate-path-targetnode*:
assumes n' *postdominates* n **and** $n\ -as\rightarrow*\ n''$ **and** $n' \notin$ *set*(*sourcenodes as*)
shows n' *postdominates* n''
⟨*proof*⟩

lemma *not-postdominate-source-not-postdominate-target*:
assumes $\neg\ n$ *postdominates* (*sourcenode* a) **and** *valid-node* n **and** *valid-edge* a
obtains ax **where** *sourcenode* a = *sourcenode* ax **and** *valid-edge* ax
and $\neg\ n$ *postdominates targetnode* ax
⟨*proof*⟩

lemma *inner-node-Entry-edge*:
assumes *inner-node* n
obtains a **where** *valid-edge* a **and** *inner-node* (*targetnode* a)
and *sourcenode* a = (*-Entry-*)
⟨*proof*⟩

lemma *inner-node-Exit-edge*:
assumes *inner-node* n
obtains a **where** *valid-edge* a **and** *inner-node* (*sourcenode* a)
and *targetnode* a = (*-Exit-*)
⟨*proof*⟩

end

1.3.2 Strong Postdomination

locale *StrongPostdomination* =

Postdomination sourcenode targetnode kind valid-edge Entry Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Exit* :: *'node* (*'('-Exit'-')*) +
assumes *successor-set-finite*: *valid-node n* ⟹
finite {*n'*. ∃ *a'*. *valid-edge a'* ∧ *sourcenode a'* = *n* ∧ *targetnode a'* = *n'*}

begin

definition *strong-postdominate* :: *'node* ⇒ *'node* ⇒ *bool*
(- *strongly−postdominates* - [*51*,*0*])
where *strong-postdominate-def*:*n' strongly−postdominates n* ≡
(*n' postdominates n* ∧
(∃ *k* ≥ *1*. ∀ *as nx*. *n* −*as*→∗ *nx* ∧
length as ≥ *k* ⟶ *n'* ∈ *set*(*sourcenodes as*)))

lemma *strong-postdominate-prop-smaller-path*:
assumes *all*:∀ *as nx*. *n* −*as*→∗ *nx* ∧ *length as* ≥ *k* ⟶ *n'* ∈ *set*(*sourcenodes as*)
and *n* −*as*→∗ *n''* **and** *length as* ≥ *k*
obtains *as' as''* **where** *n* −*as'*→∗ *n'* **and** *length as'* < *k* **and** *n'* −*as''*→∗ *n''*
and *as* = *as'*@*as''*
⟨*proof*⟩

lemma *strong-postdominate-refl*:
assumes *valid-node n* **and** *n* ≠ (*-Exit-*)
shows *n strongly−postdominates n*
⟨*proof*⟩

lemma *strong-postdominate-trans*:
assumes *n'' strongly−postdominates n* **and** *n' strongly−postdominates n''*
shows *n' strongly−postdominates n*
⟨*proof*⟩

lemma *strong-postdominate-antisym*:
⟦*n' strongly−postdominates n*; *n strongly−postdominates n'*⟧ ⟹ *n* = *n'*
⟨*proof*⟩

lemma *strong-postdominate-path-branch*:
assumes *n* −*as*→∗ *n''* **and** *n' strongly−postdominates n''*
and ¬ *n' strongly−postdominates n*
obtains *a as' as''* **where** *as* = *as'*@*a*#*as''* **and** *valid-edge a*
and ¬ *n' strongly−postdominates* (*sourcenode a*)
and *n' strongly−postdominates* (*targetnode a*)

⟨*proof*⟩

lemma *Exit-no-strong-postdominator*:
⟦(*-Exit-*) *strongly−postdominates n*; *n* −*as*→∗ (*-Exit-*)⟧ ⟹ *False*
⟨*proof*⟩

lemma *strong-postdominate-path-targetnode*:
assumes *n′ strongly−postdominates n* **and** *n* −*as*→∗ *n″*
and *n′* ∉ *set*(*sourcenodes as*)
shows *n′ strongly−postdominates n″*
⟨*proof*⟩

lemma *not-strong-postdominate-successor-set*:
assumes ¬ *n strongly−postdominates* (*sourcenode a*) **and** *valid-node n*
and *valid-edge a*
and *all*:∀ *nx* ∈ *N*. ∃ *a′*. *valid-edge a′* ∧ *sourcenode a′* = *sourcenode a* ∧
targetnode a′ = *nx* ∧ *n strongly−postdominates nx*
obtains *a′* **where** *valid-edge a′* **and** *sourcenode a′* = *sourcenode a*
and *targetnode a′* ∉ *N*
⟨*proof*⟩

lemma *not-strong-postdominate-predecessor-successor*:
assumes ¬ *n strongly−postdominates* (*sourcenode a*)
and *valid-node n* **and** *valid-edge a*
obtains *a′* **where** *valid-edge a′* **and** *sourcenode a′* = *sourcenode a*
and ¬ *n strongly−postdominates* (*targetnode a′*)
⟨*proof*⟩

end

end

1.4 CFG well-formedness

theory *CFG-wf* **imports** *CFG* **begin**

1.4.1 Well-formedness of the abstract CFG

locale *CFG-wf* = *CFG sourcenode targetnode kind valid-edge Entry*
for *sourcenode* :: *′edge* ⇒ *′node* **and** *targetnode* :: *′edge* ⇒ *′node*
and *kind* :: *′edge* ⇒ *′state edge-kind* **and** *valid-edge* :: *′edge* ⇒ *bool*

and *Entry* :: *'node* (*'('-Entry'-')*) +
fixes *Def*::*'node* ⇒ *'var set*
fixes *Use*::*'node* ⇒ *'var set*
fixes *state-val*::*'state* ⇒ *'var* ⇒ *'val*
assumes *Entry-empty*:*Def* (*-Entry-*) = {} ∧ *Use* (*-Entry-*) = {}
and *CFG-edge-no-Def-equal*:
⟦*valid-edge a*; *V* ∉ *Def* (*sourcenode a*); *pred* (*kind a*) *s*⟧
⟹ *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
and *CFG-edge-transfer-uses-only-Use*:
⟦*valid-edge a*; ∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s'* *V*;
pred (*kind a*) *s*; *pred* (*kind a*) *s'*⟧
⟹ ∀ *V* ∈ *Def* (*sourcenode a*). *state-val* (*transfer* (*kind a*) *s*) *V* =
state-val (*transfer* (*kind a*) *s'*) *V*
and *CFG-edge-Uses-pred-equal*:
⟦*valid-edge a*; *pred* (*kind a*) *s*;
∀ *V* ∈ *Use* (*sourcenode a*). *state-val s V* = *state-val s'* *V*⟧
⟹ *pred* (*kind a*) *s'*
and *deterministic*:⟦*valid-edge a*; *valid-edge a'*; *sourcenode a* = *sourcenode a'*;
targetnode a ≠ *targetnode a'*⟧
⟹ ∃ *Q Q'*. *kind a* = $(Q)_{\surd}$ ∧ *kind a'* = $(Q')_{\surd}$ ∧
(∀ *s*. (*Q s* ⟶ ¬ *Q' s*) ∧ (*Q' s* ⟶ ¬ *Q s*))

begin

lemma [*dest*!]: *V* ∈ *Use* (*-Entry-*) ⟹ *False*
⟨*proof*⟩

lemma [*dest*!]: *V* ∈ *Def* (*-Entry-*) ⟹ *False*
⟨*proof*⟩

lemma *CFG-path-no-Def-equal*:
⟦*n* −*as*→∗ *n'*; ∀ *n* ∈ *set* (*sourcenodes as*). *V* ∉ *Def n*; *preds* (*kinds as*) *s*⟧
⟹ *state-val* (*transfers* (*kinds as*) *s*) *V* = *state-val s V*
⟨*proof*⟩

end

end

theory *CFGExit-wf* **imports** *CFGExit CFG-wf* **begin**

1.4.2 New well-formedness lemmas using (*-Exit-*)

locale *CFGExit-wf* =
CFG-wf sourcenode targetnode kind valid-edge Entry Def Use state-val +
CFGExit sourcenode targetnode kind valid-edge Entry Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*

and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (*'('-Exit'-')*) +
assumes *Exit-empty*:*Def* (*-Exit-*) = {} ∧ *Use* (*-Exit-*) = {}

begin

lemma *Exit-Use-empty* [*dest*!]: *V* ∈ *Use* (*-Exit-*) ⟹ *False*
⟨*proof*⟩

lemma *Exit-Def-empty* [*dest*!]: *V* ∈ *Def* (*-Exit-*) ⟹ *False*
⟨*proof*⟩

end

end

1.5 CFG and semantics conform

theory *SemanticsCFG* **imports** *CFG* **begin**

locale *CFG-semantics-wf* = *CFG sourcenode targetnode kind valid-edge Entry*
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) +
fixes *sem*::*'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *bool*
(((*1*⟨*-*,*/-*⟩) ⇒*/* (*1*⟨*-*,*/-*⟩)) [*0*,*0*,*0*,*0*] *81*)
fixes *identifies*::*'node* ⇒ *'com* ⇒ *bool* (*-* ≙ *-* [*51*,*0*] *80*)
assumes *fundamental-property*:
⟦*n* ≙ *c*; ⟨*c*,*s*⟩ ⇒ ⟨*c'*,*s'*⟩⟧ ⟹
∃ *n'* *as*. *n* −*as*→∗ *n'* ∧ *transfers* (*kinds as*) *s* = *s'* ∧ *preds* (*kinds as*) *s* ∧
n' ≙ *c'*

end

1.6 Dynamic data dependence

theory *DynDataDependence* **imports** *CFG-wf* **begin**

context *CFG-wf* **begin**

definition *dyn-data-dependence* ::
'node ⇒ *'var* ⇒ *'node* ⇒ *'edge list* ⇒ *bool* (*- influences - in - via - [51,0,0]*)
where *n influences V in n' via as* ≡
((*V* ∈ *Def n*) ∧ (*V* ∈ *Use n'*) ∧ (*n* −*as*→∗ *n'*) ∧

$(\exists\, a'\ as'.\ (as = a'\#as') \wedge (\forall\, n'' \in set\ (sourcenodes\ as').\ V \notin Def\ n''))$

lemma *dyn-influence-Cons-source*:
n influences V in n' via a#as $\Longrightarrow$ *sourcenode a = n*
⟨*proof*⟩

lemma *dyn-influence-source-notin-tl-edges*:
assumes *n influences V in n' via a#as*
shows *n* $\notin$ *set* (*sourcenodes as*)
⟨*proof*⟩

lemma *dyn-influence-only-first-edge*:
assumes *n influences V in n' via a#as* **and** *preds* (*kinds* (*a#as*)) *s*
shows *state-val* (*transfers* (*kinds* (*a#as*)) *s*) *V* =
state-val (*transfer* (*kind a*) *s*) *V*
⟨*proof*⟩

end

end

1.7 Dynamic Standard Control Dependence

theory *DynStandardControlDependence* **imports** *Postdomination* **begin**

context *Postdomination* **begin**

definition
dyn-standard-control-dependence :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
(- *controls*$_s$ - *via* - [*51*,*0*,*0*])
where *dyn-standard-control-dependence-def*:*n controls*$_s$ *n' via as* ≡
$(\exists\, a\ a'\ as'.\ (as = a\#as') \wedge (n' \notin set(sourcenodes\ as)) \wedge (n\ -as\rightarrow*\ n') \wedge$
$(n'\ postdominates\ (targetnode\ a)) \wedge$
$(valid\text{-}edge\ a') \wedge (sourcenode\ a' = n) \wedge$
$(\neg\ n'\ postdominates\ (targetnode\ a')))$

lemma *Exit-not-dyn-standard-control-dependent*:
assumes *control*:*n controls*$_s$ (*-Exit-*) *via as* **shows** *False*
⟨*proof*⟩

lemma *dyn-standard-control-dependence-def-variant*:
n controls$_s$ *n' via as* = $((n\ -as\rightarrow*\ n') \wedge (n \neq n') \wedge$
$(\neg\ n'\ postdominates\ n) \wedge (n' \notin set(sourcenodes\ as)) \wedge$

$(\forall\, n''.\ set(targetnodes\ as).\ n'\ postdominates\ n''))$
⟨*proof*⟩

lemma *which-node-dyn-standard-control-dependence-source*:
assumes *path*:(*-Entry-*) $-as@a\#as'\rightarrow*\ n$
and *Exit-path*:$n\ -as''\rightarrow*$ (*-Exit-*) **and** *source*:*sourcenode* $a = n'$
and *source'*:*sourcenode* $a' = n'$
and *no-source*:$n \notin set(sourcenodes\ (a\#as'))$ **and** *valid-edge'*:*valid-edge* a'
and *inner-node*:*inner-node* n **and** *not-pd*:$\neg$ n *postdominates* (*targetnode* a')
and *last*:$\forall\, ax\ ax'.\ ax \in set\ as' \wedge sourcenode\ ax = sourcenode\ ax' \wedge$
valid-edge $ax' \longrightarrow n$ *postdominates targetnode* ax'
shows n' *controls*$_s$ n *via* $a\#as'$
⟨*proof*⟩

lemma *inner-node-dyn-standard-control-dependence-predecessor*:
assumes *inner-node*:*inner-node* n
obtains n' *as* **where** n' *controls*$_s$ n *via as*
⟨*proof*⟩

end

end

1.8 Dynamic Weak Control Dependence

theory *DynWeakControlDependence* **imports** *Postdomination* **begin**

context *StrongPostdomination* **begin**

definition
dyn-weak-control-dependence :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
(*- weakly controls - via -* [*51*,*0*,*0*])
where *dyn-weak-control-dependence-def*:n *weakly controls* n' *via as* ≡
$(\exists\, a\ a'\ as'.\ (as = a\#as') \wedge (n' \notin set(sourcenodes\ as)) \wedge (n\ -as\rightarrow*\ n') \wedge$
$(n'\ strongly{-}postdominates\ (targetnode\ a)) \wedge$
$(valid\text{-}edge\ a') \wedge (sourcenode\ a' = n) \wedge$
$(\neg\ n'\ strongly{-}postdominates\ (targetnode\ a')))$

lemma *Exit-not-dyn-weak-control-dependent*:
assumes *control*:n *weakly controls* (*-Exit-*) *via as* **shows** *False*
⟨*proof*⟩

end

end

Chapter 2

Dynamic Slicing

2.1 Dynamic Program Dependence Graph

theory *DynPDG* **imports**
../Basic/DynDataDependence
../Basic/CFGExit-wf
../Basic/DynStandardControlDependence
../Basic/DynWeakControlDependence
begin

2.1.1 The dynamic PDG

locale *DynPDG* =
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (*'('-Exit'-')*) +
fixes *dyn-control-dependence* :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
(- *controls* - *via* - [*51*,*0*,*0*])
assumes *Exit-not-dyn-control-dependent*:*n controls n'* *via as* ⟹ *n'* ≠ (-*Exit*-)
assumes *dyn-control-dependence-path*:
n controls n' *via as* ⟹ *n* −*as*→∗ *n'* ∧ *as* ≠ []

begin

inductive *cdep-edge* :: *'node* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(- −-→$_{cd}$ - [*51*,*0*,*0*] *80*)
and *ddep-edge* :: *'node* ⇒ *'var* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(- −*'{-'}*-→$_{dd}$ - [*51*,*0*,*0*,*0*] *80*)
and *DynPDG-edge* :: *'node* ⇒ *'var option* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*

where
— Syntax

$n\ -as\rightarrow_{cd}\ n' == DynPDG\text{-}edge\ n\ None\ as\ n'$
$|\ n\ -\{V\}as\rightarrow_{dd}\ n' == DynPDG\text{-}edge\ n\ (Some\ V)\ as\ n'$

— Rules
| *DynPDG-cdep-edge*:
$n\ controls\ n'\ via\ as \Longrightarrow n\ -as\rightarrow_{cd}\ n'$

| *DynPDG-ddep-edge*:
$n\ influences\ V\ in\ n'\ via\ as \Longrightarrow n\ -\{V\}as\rightarrow_{dd}\ n'$

inductive *DynPDG-path* :: *'node* ⇒ *'edge list* ⇒ *'node* ⇒ *bool*
(- −-→$_{d}$* - [51,0,0] 80)

where *DynPDG-path-Nil*:
$valid\text{-}node\ n \Longrightarrow n\ -[]\rightarrow_{d*}\ n$

| *DynPDG-path-Append-cdep*:
$[\![n\ -as\rightarrow_{d*}\ n''; n''\ -as'\rightarrow_{cd}\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{d*}\ n'$

| *DynPDG-path-Append-ddep*:
$[\![n\ -as\rightarrow_{d*}\ n''; n''\ -\{V\}as'\rightarrow_{dd}\ n']\!] \Longrightarrow n\ -as@as'\rightarrow_{d*}\ n'$

lemma *DynPDG-empty-path-eq-nodes*:$n\ -[]\rightarrow_{d*}\ n' \Longrightarrow n = n'$
⟨*proof*⟩

lemma *DynPDG-path-cdep*:$n\ -as\rightarrow_{cd}\ n' \Longrightarrow n\ -as\rightarrow_{d*}\ n'$
⟨*proof*⟩

lemma *DynPDG-path-ddep*:$n\ -\{V\}as\rightarrow_{dd}\ n' \Longrightarrow n\ -as\rightarrow_{d*}\ n'$
⟨*proof*⟩

lemma *DynPDG-path-Append*:
$[\![n''\ -as'\rightarrow_{d*}\ n'; n\ -as\rightarrow_{d*}\ n'']\!] \Longrightarrow n\ -as@as'\rightarrow_{d*}\ n'$
⟨*proof*⟩

lemma *DynPDG-path-Exit*:$[\![n\ -as\rightarrow_{d*}\ n'; n' = (\text{-}Exit\text{-})]\!] \Longrightarrow n = (\text{-}Exit\text{-})$
⟨*proof*⟩

lemma *DynPDG-path-not-inner*:
$[\![n\ -as\rightarrow_{d*}\ n'; \neg\ inner\text{-}node\ n']\!] \Longrightarrow n = n'$
⟨*proof*⟩

lemma *DynPDG-cdep-edge-CFG-path*:

assumes $n\ -as\rightarrow_{cd}\ n'$ **shows** $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
⟨*proof*⟩

lemma *DynPDG-ddep-edge-CFG-path*:
assumes $n\ -\{V\}as\rightarrow_{dd}\ n'$ **shows** $n\ -as\rightarrow*\ n'$ **and** $as \neq []$
⟨*proof*⟩

lemma *DynPDG-path-CFG-path*:
$n\ -as\rightarrow_{d}*\ n' \Longrightarrow n\ -as\rightarrow*\ n'$
⟨*proof*⟩

lemma *DynPDG-path-split*:
$n\ -as\rightarrow_{d}*\ n' \Longrightarrow$
$(as = [] \wedge n = n') \vee$
$(\exists\, n''\ asx\ asx'.\ (n\ -asx\rightarrow_{cd}\ n'') \wedge (n''\ -asx'\rightarrow_{d}*\ n') \wedge$
$\quad (as = asx@asx')) \vee$
$(\exists\, n''\ V\ asx\ asx'.\ (n\ -\{V\}asx\rightarrow_{dd}\ n'') \wedge (n''\ -asx'\rightarrow_{d}*\ n') \wedge$
$\quad (as = asx@asx'))$
⟨*proof*⟩

lemma *DynPDG-path-rev-cases* [*consumes 1*,
case-names DynPDG-path-Nil DynPDG-path-cdep-Append DynPDG-path-ddep-Append]:
$[\![n\ -as\rightarrow_{d}*\ n';\ [\![as = [];\ n = n']\!] \Longrightarrow Q;$
$\quad \bigwedge n''\ asx\ asx'.\ [\![n\ -asx\rightarrow_{cd}\ n'';\ n''\ -asx'\rightarrow_{d}*\ n';$
$\qquad\qquad as = asx@asx']\!] \Longrightarrow Q;$
$\quad \bigwedge V\ n''\ asx\ asx'.\ [\![n\ -\{V\}asx\rightarrow_{dd}\ n'';\ n''\ -asx'\rightarrow_{d}*\ n';$
$\qquad\qquad as = asx@asx']\!] \Longrightarrow Q]\!]$
$\Longrightarrow Q$
⟨*proof*⟩

lemma *DynPDG-ddep-edge-no-shorter-ddep-edge*:
assumes *ddep*:$n\ -\{V\}as\rightarrow_{dd}\ n'$
shows $\forall\, as'\ a\ as''.\ tl\ as = as'@a\#as'' \longrightarrow \neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$
⟨*proof*⟩

lemma *no-ddep-same-state*:
assumes *path*:$n\ -as\rightarrow*\ n'$ **and** *Uses*:$V \in Use\ n'$ **and** *preds*:*preds* (*kinds as*) *s*
and *no-dep*:$\forall\, as'\ a\ as''.\ as = as'@a\#as'' \longrightarrow \neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd}\ n'$
shows *state-val* (*transfers* (*kinds as*) *s*) V = *state-val s V*
⟨*proof*⟩

lemma *DynPDG-ddep-edge-only-first-edge*:
$[\![n -\{V\}a\#as\rightarrow_{dd} n'; preds\ (kinds\ (a\#as))\ s]\!] \Longrightarrow$
$state\text{-}val\ (transfers\ (kinds\ (a\#as))\ s)\ V = state\text{-}val\ (transfer\ (kind\ a)\ s)\ V$
⟨*proof*⟩

lemma *Use-value-change-implies-DynPDG-ddep-edge*:
assumes $n -as\rightarrow* n'$ **and** $V \in Use\ n'$ **and** $preds\ (kinds\ as)\ s$
and $preds\ (kinds\ as)\ s'$ **and** $state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
and $state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V \neq$
$state\text{-}val\ (transfers\ (kinds\ as)\ s')\ V$
obtains $as'\ a\ as''$ **where** $as = as'@a\#as''$
and $sourcenode\ a -\{V\}a\#as''\rightarrow_{dd} n'$
and $state\text{-}val\ (transfers\ (kinds\ as)\ s)\ V =$
$state\text{-}val\ (transfers\ (kinds\ (as'@[a]))\ s)\ V$
and $state\text{-}val\ (transfers\ (kinds\ as)\ s')\ V =$
$state\text{-}val\ (transfers\ (kinds\ (as'@[a]))\ s')\ V$
⟨*proof*⟩

end

2.1.2 Instantiate dynamic PDG

Standard control dependence

locale *DynStandardControlDependencePDG* =
Postdomination sourcenode targetnode kind valid-edge Entry Exit +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (*'('-Exit'-')*)

begin

lemma *DynPDG-scd*:
DynPDG sourcenode targetnode kind valid-edge (-Entry-)
Def Use state-val (-Exit-) dyn-standard-control-dependence
⟨*proof*⟩

end

Weak control dependence

locale *DynWeakControlDependencePDG* =
StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit

for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (*'('-Exit'-')*)

begin

lemma *DynPDG-wcd*:
DynPDG sourcenode targetnode kind valid-edge (*-Entry-*)
Def Use state-val (*-Exit-*) *dyn-weak-control-dependence*
⟨*proof*⟩

end

2.1.3 Data slice

definition (**in** *CFG*) *empty-control-dependence* :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
where *empty-control-dependence n n' as* ≡ *False*

lemma (**in** *CFGExit-wf*) *DynPDG-scd*:
DynPDG sourcenode targetnode kind valid-edge (*-Entry-*)
Def Use state-val (*-Exit-*) *empty-control-dependence*
⟨*proof*⟩

end

2.2 Dependent Live Variables

theory *DependentLiveVariables* **imports** *DynPDG* **begin**

dependent-live-vars calculates variables which can change
the value of the *Use* variables of the target node

context *DynPDG* **begin**

inductive-set
dependent-live-vars :: *'node* ⇒ (*'var* × *'edge list* × *'edge list*) *set*
for *n'* :: *'node*
where *dep-vars-Use*:
V ∈ *Use n'* ⟹ (*V*,[],[]) ∈ *dependent-live-vars n'*

| *dep-vars-Cons-cdep*:
⟦*V* ∈ *Use* (*sourcenode a*); *sourcenode a* −*a*#*as'*→$_{cd}$ *n''*; *n''* −*as''*→$_{d}$* *n'*⟧
⟹ (*V*,[],*a*#*as'*@*as''*) ∈ *dependent-live-vars n'*

| *dep-vars-Cons-ddep*:

$[\![(V,as',as) \in \mathit{dependent\text{-}live\text{-}vars}\ n';\ V' \in \mathit{Use}\ (\mathit{sourcenode}\ a);$
$n' = \mathit{last}(\mathit{targetnodes}\ (a\#as));$
$\mathit{sourcenode}\ a\ -\{V\}a\#as'\rightarrow_{dd} \mathit{last}(\mathit{targetnodes}\ (a\#as'))]\!]$
$\Longrightarrow (V',[],a\#as) \in \mathit{dependent\text{-}live\text{-}vars}\ n'$

| *dep-vars-Cons-keep*:
$[\![(V,as',as) \in \mathit{dependent\text{-}live\text{-}vars}\ n';\ n' = \mathit{last}(\mathit{targetnodes}\ (a\#as));$
$\neg\ \mathit{sourcenode}\ a\ -\{V\}a\#as'\rightarrow_{dd} \mathit{last}(\mathit{targetnodes}\ (a\#as'))]\!]$
$\Longrightarrow (V,a\#as',a\#as) \in \mathit{dependent\text{-}live\text{-}vars}\ n'$

lemma *dependent-live-vars-fst-prefix-snd*:
$(V,as',as) \in \mathit{dependent\text{-}live\text{-}vars}\ n' \Longrightarrow \exists\, as''.\ as'@as'' = as$
⟨*proof*⟩

lemma *dependent-live-vars-Exit-empty* [*dest*]:
$(V,as',as) \in \mathit{dependent\text{-}live\text{-}vars}\ (\text{-}\mathit{Exit}\text{-}) \Longrightarrow \mathit{False}$
⟨*proof*⟩

lemma *dependent-live-vars-lastnode*:
$[\![(V,as',as) \in \mathit{dependent\text{-}live\text{-}vars}\ n';\ as \neq []]\!]$
$\Longrightarrow n' = \mathit{last}(\mathit{targetnodes}\ as)$
⟨*proof*⟩

lemma *dependent-live-vars-Use-cases*:
$[\![(V,as',as) \in \mathit{dependent\text{-}live\text{-}vars}\ n';\ n\ -as\rightarrow*\ n']\!]$
$\Longrightarrow \exists\, nx\ as''.\ as = as'@as'' \wedge n\ -as'\rightarrow*\ nx \wedge nx\ -as''\rightarrow_d*\ n' \wedge V \in \mathit{Use}\ nx$
$\wedge$
$(\forall\, n'' \in \mathit{set}\ (\mathit{sourcenodes}\ as').\ V \notin \mathit{Def}\ n'')$
⟨*proof*⟩

lemma *dependent-live-vars-dependent-edge*:
assumes $(V,as',as) \in \mathit{dependent\text{-}live\text{-}vars}\ n'$
and $\mathit{targetnode}\ a\ -as\rightarrow*\ n'$
and $V \in \mathit{Def}\ (\mathit{sourcenode}\ a)$ **and** *valid-edge a*
obtains $nx\ as''$ **where** $as = as'@as''$ **and** $\mathit{sourcenode}\ a\ -\{V\}a\#as'\rightarrow_{dd} nx$
and $nx\ -as''\rightarrow_d*\ n'$
⟨*proof*⟩

lemma *dependent-live-vars-same-pathsI*:
assumes $V \in \mathit{Use}\ n'$
shows $[\![\forall\, as'\ a\ as''.\ as = as'@a\#as'' \longrightarrow \neg\ \mathit{sourcenode}\ a\ -\{V\}a\#as''\rightarrow_{dd} n';$

$as \neq [] \longrightarrow n' = last(targetnodes\ as)]\!]$
$\Longrightarrow (V,as,as) \in dependent\text{-}live\text{-}vars\ n'$
⟨*proof*⟩

lemma *dependent-live-vars-same-pathsD*:
$[\![(V,as,as) \in dependent\text{-}live\text{-}vars\ n';\ as \neq [] \longrightarrow n' = last(targetnodes\ as)]\!]$
$\Longrightarrow V \in Use\ n' \wedge (\forall as'\ a\ as''.\ as = as'@a\#as'' \longrightarrow$
$\neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd} n')$
⟨*proof*⟩

lemma *dependent-live-vars-same-paths*:
$as \neq [] \longrightarrow n' = last(targetnodes\ as) \Longrightarrow$
$(V,as,as) \in dependent\text{-}live\text{-}vars\ n' =$
$(V \in Use\ n' \wedge (\forall as'\ a\ as''.\ as = as'@a\#as'' \longrightarrow$
$\neg\ sourcenode\ a\ -\{V\}a\#as''\rightarrow_{dd} n'))$
⟨*proof*⟩

lemma *dependent-live-vars-cdep-empty-fst*:
assumes $n'' -as\rightarrow_{cd} n'$ **and** $V' \in Use\ n''$
shows $(V',[],as) \in dependent\text{-}live\text{-}vars\ n'$
⟨*proof*⟩

lemma *dependent-live-vars-ddep-empty-fst*:
assumes $n'' -\{V\}as\rightarrow_{dd} n'$ **and** $V' \in Use\ n''$
shows $(V',[],as) \in dependent\text{-}live\text{-}vars\ n'$
⟨*proof*⟩

lemma *ddep-dependent-live-vars-keep-notempty*:
assumes $n -\{V\}a\#as\rightarrow_{dd} n''$ **and** $as' \neq []$
and $(V,as'',as') \in dependent\text{-}live\text{-}vars\ n'$
shows $(V,as@as'',as@as') \in dependent\text{-}live\text{-}vars\ n'$
⟨*proof*⟩

lemma *dependent-live-vars-cdep-dependent-live-vars*:
assumes $n'' -as''\rightarrow_{cd} n'$ **and** $(V',as',as) \in dependent\text{-}live\text{-}vars\ n''$
shows $(V',as',as@as'') \in dependent\text{-}live\text{-}vars\ n'$
⟨*proof*⟩

lemma *dependent-live-vars-ddep-dependent-live-vars*:

assumes $n'' -\{V\}as''\rightarrow_{dd} n'$ **and** $(V',as',as) \in$ *dependent-live-vars* n''
shows $(V',as',as@as'') \in$ *dependent-live-vars* n'
⟨*proof*⟩

lemma *dependent-live-vars-dep-dependent-live-vars*:
$[\![n'' -as''\rightarrow_{d*} n'; (V',as',as) \in$ *dependent-live-vars* $n'']\!]$
$\Longrightarrow (V',as',as@as'') \in$ *dependent-live-vars* n'
⟨*proof*⟩

end

end

2.3 Formalization of Bit Vectors

theory *BitVector* **imports** *Main* **begin**

type-synonym *bit-vector* = *bool list*

fun *bv-leqs* :: *bit-vector* ⇒ *bit-vector* ⇒ *bool* (- $\preceq_b$ - *99*)
where *bv-Nils*:[] $\preceq_b$ [] = *True*
| *bv-Cons*:$(x\#xs) \preceq_b (y\#ys) = ((x \longrightarrow y) \wedge xs \preceq_b ys)$
| *bv-rest*:$xs \preceq_b ys$ = *False*

2.3.1 Some basic properties

lemma *bv-length*: $xs \preceq_b ys \Longrightarrow$ *length* xs = *length* ys
⟨*proof*⟩

lemma [*dest*!]: $xs \preceq_b [] \Longrightarrow xs = []$
⟨*proof*⟩

lemma *bv-leqs-AppendI*:
$[\![xs \preceq_b ys; xs' \preceq_b ys']\!] \Longrightarrow (xs@xs') \preceq_b (ys@ys')$
⟨*proof*⟩

lemma *bv-leqs-AppendD*:
$[\![(xs@xs') \preceq_b (ys@ys')$; *length* xs = *length* $ys]\!]$
$\Longrightarrow xs \preceq_b ys \wedge xs' \preceq_b ys'$
⟨*proof*⟩

lemma *bv-leqs-eq*:

$xs \preceq_b ys = ((\forall i < length\ xs.\ xs\ !\ i \longrightarrow ys\ !\ i) \wedge length\ xs = length\ ys)$
⟨*proof*⟩

2.3.2 $\preceq_b$ is an order on bit vectors with minimal and maximal element

lemma *minimal-element*:
$replicate\ (length\ xs)\ False \preceq_b xs$
⟨*proof*⟩

lemma *maximal-element*:
$xs \preceq_b replicate\ (length\ xs)\ True$
⟨*proof*⟩

lemma *bv-leqs-refl*:$xs \preceq_b xs$
⟨*proof*⟩

lemma *bv-leqs-trans*:$[\![xs \preceq_b ys;\ ys \preceq_b zs]\!] \Longrightarrow xs \preceq_b zs$
⟨*proof*⟩

lemma *bv-leqs-antisym*:$[\![xs \preceq_b ys;\ ys \preceq_b xs]\!] \Longrightarrow xs = ys$
⟨*proof*⟩

definition *bv-less* :: *bit-vector* ⇒ *bit-vector* ⇒ *bool* (- $\prec_b$ - *99*)
where $xs \prec_b ys \equiv xs \preceq_b ys \wedge xs \neq ys$

interpretation *order bv-leqs bv-less*
⟨*proof*⟩

end

2.4 Dynamic Backward Slice

theory *DynSlice* **imports** *DependentLiveVariables BitVector ../Basic/SemanticsCFG*
begin

2.4.1 Backward slice of paths

context *DynPDG* **begin**

fun *slice-path* :: *'edge list* ⇒ *bit-vector*
where *slice-path* [] = []
| *slice-path* (*a*#*as*) = (*let n'* = *last*(*targetnodes* (*a*#*as*)) *in*

$(sourcenode\ a\ -a\#as\rightarrow_d*\ n')\#slice\text{-}path\ as)$

lemma *slice-path-length*:
$length(slice\text{-}path\ as) = length\ as$
⟨*proof*⟩

lemma *slice-path-right-Cons*:
assumes *slice*:$slice\text{-}path\ as = x\#xs$
obtains $a'\ as'$ **where** $as = a'\#as'$ **and** $slice\text{-}path\ as' = xs$
⟨*proof*⟩

2.4.2 The proof of the fundamental property of (dynamic) slicing

fun *select-edge-kinds* :: *'edge list* ⇒ *bit-vector* ⇒ *'state edge-kind list*
where *select-edge-kinds* [] [] = []
| *select-edge-kinds* (*a*#*as*) (*b*#*bs*) = (*if b then kind a*
else (*case kind a of* ⇑*f* ⇒ ⇑*id* | $(Q)_{\surd}$ ⇒ $(\lambda s.\ True)_{\surd}$))#*select-edge-kinds as bs*

definition *slice-kinds* :: *'edge list* ⇒ *'state edge-kind list*
where *slice-kinds as* = *select-edge-kinds as* (*slice-path as*)

lemma *select-edge-kinds-max-bv*:
select-edge-kinds as (*replicate* (*length as*) *True*) = *kinds as*
⟨*proof*⟩

lemma *slice-path-leqs-information-same-Uses*:
⟦$n\ -as\rightarrow*\ n'$; $bs \preceq_b bs'$; *slice-path as* = *bs*;
select-edge-kinds as bs = *es*; *select-edge-kinds as bs'* = *es'*;
$\forall\ V\ xs.\ (V,xs,as) \in dependent\text{-}live\text{-}vars\ n' \longrightarrow state\text{-}val\ s\ V = state\text{-}val\ s'\ V$;
preds es' s'⟧
⟹ (∀ *V* ∈ *Use n'*. *state-val* (*transfers es s*) *V* =
state-val (*transfers es' s'*) *V*) ∧ *preds es s*
⟨*proof*⟩

theorem *fundamental-property-of-path-slicing*:
assumes $n\ -as\rightarrow*\ n'$ **and** *preds* (*kinds as*) *s*
shows (∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*slice-kinds as*) *s*) *V* =
state-val (*transfers* (*kinds as*) *s*) *V*)
and *preds* (*slice-kinds as*) *s*
⟨*proof*⟩

end

2.4.3 The fundamental property of (dynamic) slicing related to the semantics

locale *BackwardPathSlice-wf* =
DynPDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit dyn-control-dependence +
CFG-semantics-wf sourcenode targetnode kind valid-edge Entry sem identifies
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *dyn-control-dependence* :: *'node* ⇒ *'node* ⇒ *'edge list* ⇒ *bool*
(- *controls* - *via* - [*51*, *0*, *0*] *1000*)
and *Exit* :: *'node* (*'('-Exit'-')*)
and *sem* :: *'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *bool*
(((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩)) [*0*,*0*,*0*,*0*] *81*)
and *identifies* :: *'node* ⇒ *'com* ⇒ *bool* (- ≙ - [*51*, *0*] *80*)

begin

theorem *fundamental-property-of-path-slicing-semantically*:
assumes *n* ≙ *c* **and** ⟨*c*,*s*⟩ ⇒ ⟨*c'*,*s'*⟩
obtains *n'* *as* **where** *n* −*as*→∗ *n'* **and** *preds* (*slice-kinds as*) *s*
and *n'* ≙ *c'*
and ∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*slice-kinds as*) *s*) *V* =
state-val s' V
⟨*proof*⟩

end

end

2.5 Observable Sets of Nodes

theory *Observable* **imports** *../Basic/CFG* **begin**

context *CFG* **begin**

inductive-set *obs* :: *'node* ⇒ *'node set* ⇒ *'node set*
for *n*::*'node* **and** *S*::*'node set*
where *obs-elem*:
⟦*n* −*as*→∗ *n'*; ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ *S*; *n'* ∈ *S*⟧ ⟹ *n'* ∈ *obs n S*

lemma *obsE*:
assumes *n'* ∈ *obs n S*
obtains *as* **where** *n* −*as*→∗ *n'* **and** ∀ *nx* ∈ *set*(*sourcenodes as*). *nx* ∉ *S*

and $n' \in S$
⟨*proof*⟩

lemma *n-in-obs*:
assumes *valid-node* n **and** $n \in S$ **shows** *obs* n $S = \{n\}$
⟨*proof*⟩

lemma *in-obs-valid*:
assumes $n' \in$ *obs* n S **shows** *valid-node* n **and** *valid-node* n'
⟨*proof*⟩

lemma *edge-obs-subset*:
assumes*valid-edge* a **and** *sourcenode* $a \notin S$
shows *obs* (*targetnode* a) $S \subseteq$ *obs* (*sourcenode* a) S
⟨*proof*⟩

lemma *path-obs-subset*:
$[\![n\ -as\rightarrow*\ n'; \forall\, n' \in set(sourcenodes\ as).\ n' \notin S]\!]$
$\Longrightarrow$ *obs* n' $S \subseteq$ *obs* n S
⟨*proof*⟩

lemma *path-ex-obs*:
assumes $n\ -as\rightarrow*\ n'$ **and** $n' \in S$
obtains m **where** $m \in$ *obs* n S
⟨*proof*⟩

end

end

Chapter 3

Static Intraprocedural Slicing

Static Slicing analyses a CFG prior to execution. Whereas dynamic slicing can provide better results for certain inputs (i.e. trace and initial state), static slicing is more conservative but provides results independent of inputs.
Correctness for static slicing is defined differently than correctness of dynamic slicing by a weak simulation between nodes and states when traversing the original and the sliced graph. The weak simulation property demands that if a (node,state) tuples (n_1, s_1) simulates (n_2, s_2) and making an observable move in the original graph leads from (n_1, s_1) to (n_1', s_1'), this tuple simulates a tuple (n_2, s_2) which is the result of making an observable move in the sliced graph beginning in (n_2', s_2').
We also show how a "dynamic slicing style" correctness criterion for static slicing of a given trace and initial state could look like.
This formalization of static intraprocedural slicing is instantiable with three different kinds of control dependences: standard control, weak control and weak order dependence. The correctness proof for slicing is independent of the control dependence used, it bases only on one property every control dependence definition hass to fulfill.

3.1 Distance of Paths

theory *Distance* **imports** *../Basic/CFG* **begin**

context *CFG* **begin**

inductive *distance* :: *'node* ⇒ *'node* ⇒ *nat* ⇒ *bool*
where *distanceI*:
⟦*n* −*as*→∗ *n'*; *length as* = *x*; ∀ *as'*. *n* −*as'*→∗ *n'* ⟶ *x* ≤ *length as'*⟧
⟹ *distance n n' x*

lemma *every-path-distance*:
assumes *n* −*as*→∗ *n'*

obtains x **where** *distance* $n\ n'\ x$ **and** $x \leq$ *length as*
⟨*proof*⟩

lemma *distance-det*:
⟦*distance* $n\ n'\ x$; *distance* $n\ n'\ x'$⟧ $\Longrightarrow x = x'$
⟨*proof*⟩

lemma *only-one-SOME-dist-edge*:
assumes *valid*:*valid-edge a* **and** *dist*:*distance* (*targetnode a*) $n'\ x$
shows $\exists!a'$. *sourcenode a* = *sourcenode a'* ∧ *distance* (*targetnode a'*) $n'\ x$ ∧
valid-edge a' ∧
targetnode a' = (*SOME nx*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) $n'\ x$ ∧
valid-edge a' ∧ *targetnode a'* = *nx*)
⟨*proof*⟩

lemma *distance-successor-distance*:
assumes *distance* $n\ n'\ x$ **and** $x \neq 0$
obtains *a* **where** *valid-edge a* **and** n = *sourcenode a*
and *distance* (*targetnode a*) $n'\ (x - 1)$
and *targetnode a* = (*SOME nx*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) $n'\ (x - 1)$ ∧
valid-edge a' ∧ *targetnode a'* = *nx*)
⟨*proof*⟩

end

end

3.2 Static data dependence

theory *DataDependence* **imports** *../Basic/DynDataDependence* **begin**

context *CFG-wf* **begin**

definition *data-dependence* :: *'node* ⇒ *'var* ⇒ *'node* ⇒ *bool*
(- *influences* - *in* - [*51*,*0*])
where *data-dependences-eq*:*n influences V in n'* ≡ ∃ *as*. *n influences V in n' via as*

lemma *data-dependence-def*: *n influences V in n'* =
(∃ *a' as'*. (*V* ∈ *Def n*) ∧ (*V* ∈ *Use n'*) ∧
($n\ -a'\#as'\rightarrow*\ n'$) ∧ (∀ n'' ∈ *set* (*sourcenodes as'*). *V* ∉ *Def n''*))

⟨*proof*⟩

end

end

3.3 Static backward slice

theory *Slice*
imports *Observable Distance DataDependence ../Basic/SemanticsCFG*
begin

locale *BackwardSlice* =
CFG-wf sourcenode targetnode kind valid-edge Entry Def Use state-val
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val* +
fixes *backward-slice* :: *'node set* ⇒ *'node set*
assumes *valid-nodes*:*n* ∈ *backward-slice S* ⟹ *valid-node n*
and *refl*:⟦*valid-node n*; *n* ∈ *S*⟧ ⟹ *n* ∈ *backward-slice S*
and *dd-closed*:⟦*n'* ∈ *backward-slice S*; *n influences V in n'*⟧
⟹ *n* ∈ *backward-slice S*
and *obs-finite*:*finite* (*obs n* (*backward-slice S*))
and *obs-singleton*:*card* (*obs n* (*backward-slice S*)) ≤ *1*

begin

lemma *slice-n-in-obs*:
n ∈ *backward-slice S* ⟹ *obs n* (*backward-slice S*) = {*n*}
⟨*proof*⟩

lemma *obs-singleton-disj*:
(∃ *m*. *obs n* (*backward-slice S*) = {*m*}) ∨ *obs n* (*backward-slice S*) = {}
⟨*proof*⟩

lemma *obs-singleton-element*:
assumes *m* ∈ *obs n* (*backward-slice S*) **shows** *obs n* (*backward-slice S*) = {*m*}
⟨*proof*⟩

lemma *obs-the-element*:
m ∈ *obs n* (*backward-slice S*) ⟹ (*THE m*. *m* ∈ *obs n* (*backward-slice S*)) = *m*
⟨*proof*⟩

3.3.1 Traversing the sliced graph

slice-kind S a conforms to *kind a* in the sliced graph

definition *slice-kind* :: *'node set* ⇒ *'edge* ⇒ *'state edge-kind*
where *slice-kind S a* = (*let S'* = *backward-slice S*; *n* = *sourcenode a in*
(*if sourcenode a* ∈ *S' then kind a*
else (*case kind a of* ⇑*f* ⇒ ⇑*id* | (*Q*)$_{\surd}$ ⇒
(*if obs* (*sourcenode a*) *S'* = {} *then*
(*let nx* = (*SOME n'*. ∃ *a'*. *n* = *sourcenode a'* ∧ *valid-edge a'* ∧ *targetnode a'*
= *n'*)
in (*if* (*targetnode a* = *nx*) *then* (λ*s*. *True*)$_{\surd}$ *else* (λ*s*. *False*)$_{\surd}$))
else (*let m* = *THE m*. *m* ∈ *obs n S' in*
(*if* (∃ *x*. *distance* (*targetnode a*) *m x* ∧ *distance n m* (*x* + *1*) ∧
(*targetnode a* = (*SOME nx'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧
distance (*targetnode a'*) *m x* ∧
valid-edge a' ∧ *targetnode a'* = *nx'*)))
then (λ*s*. *True*)$_{\surd}$ *else* (λ*s*. *False*)$_{\surd}$
))
))
))

definition
slice-kinds :: *'node set* ⇒ *'edge list* ⇒ *'state edge-kind list*
where *slice-kinds S as* ≡ *map* (*slice-kind S*) *as*

lemma *slice-kind-in-slice*:
sourcenode a ∈ *backward-slice S* ⟹ *slice-kind S a* = *kind a*
⟨*proof*⟩

lemma *slice-kind-Upd*:
⟦*sourcenode a* ∉ *backward-slice S*; *kind a* = ⇑*f*⟧ ⟹ *slice-kind S a* = ⇑*id*
⟨*proof*⟩

lemma *slice-kind-Pred-empty-obs-SOME*:
⟦*sourcenode a* ∉ *backward-slice S*; *kind a* = (*Q*)$_{\surd}$;
obs (*sourcenode a*) (*backward-slice S*) = {};
targetnode a = (*SOME n'*. ∃ *a'*. *sourcenode a* = *sourcenode a'* ∧ *valid-edge a'*
∧
targetnode a' = *n'*)⟧
⟹ *slice-kind S a* = (λ*s*. *True*)$_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-empty-obs-not-SOME*:
⟦*sourcenode a* ∉ *backward-slice S*; *kind a* = (*Q*)$_{\surd}$;

$obs\ (sourcenode\ a)\ (backward\text{-}slice\ S) = \{\};$
$targetnode\ a \neq (SOME\ n'.\ \exists\, a'.\ sourcenode\ a = sourcenode\ a' \wedge valid\text{-}edge\ a' \wedge$
$targetnode\ a' = n')]\!]$
$\Longrightarrow slice\text{-}kind\ S\ a = (\lambda s.\ False)_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-obs-nearer-SOME*:
assumes $sourcenode\ a \notin backward\text{-}slice\ S$ **and** $kind\ a = (Q)_{\surd}$
and $m \in obs\ (sourcenode\ a)\ (backward\text{-}slice\ S)$
and $distance\ (targetnode\ a)\ m\ x\ distance\ (sourcenode\ a)\ m\ (x + 1)$
and $targetnode\ a = (SOME\ n'.\ \exists\, a'.\ sourcenode\ a = sourcenode\ a' \wedge$
$distance\ (targetnode\ a')\ m\ x \wedge$
$valid\text{-}edge\ a' \wedge targetnode\ a' = n')$
shows $slice\text{-}kind\ S\ a = (\lambda s.\ True)_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-obs-nearer-not-SOME*:
assumes $sourcenode\ a \notin backward\text{-}slice\ S$ **and** $kind\ a = (Q)_{\surd}$
and $m \in obs\ (sourcenode\ a)\ (backward\text{-}slice\ S)$
and $distance\ (targetnode\ a)\ m\ x\ distance\ (sourcenode\ a)\ m\ (x + 1)$
and $targetnode\ a \neq (SOME\ nx'.\ \exists\, a'.\ sourcenode\ a = sourcenode\ a' \wedge$
$distance\ (targetnode\ a')\ m\ x \wedge$
$valid\text{-}edge\ a' \wedge targetnode\ a' = nx')$
shows $slice\text{-}kind\ S\ a = (\lambda s.\ False)_{\surd}$
⟨*proof*⟩

lemma *slice-kind-Pred-obs-not-nearer*:
assumes $sourcenode\ a \notin backward\text{-}slice\ S$ **and** $kind\ a = (Q)_{\surd}$
and *in-obs*:$m \in obs\ (sourcenode\ a)\ (backward\text{-}slice\ S)$
and *dist*:$distance\ (sourcenode\ a)\ m\ (x + 1)$
$\neg\ distance\ (targetnode\ a)\ m\ x$
shows $slice\text{-}kind\ S\ a = (\lambda s.\ False)_{\surd}$
⟨*proof*⟩

lemma *kind-Predicate-notin-slice-slice-kind-Predicate*:
assumes $kind\ a = (Q)_{\surd}$ **and** $sourcenode\ a \notin backward\text{-}slice\ S$
obtains Q' **where** $slice\text{-}kind\ S\ a = (Q')_{\surd}$ **and** $Q' = (\lambda s.\ False) \vee Q' = (\lambda s.\ True)$
⟨*proof*⟩

lemma *only-one-SOME-edge*:
assumes *valid-edge a*
shows $\exists!a'.\ sourcenode\ a = sourcenode\ a' \wedge valid\text{-}edge\ a' \wedge$

$$targetnode\ a' = (SOME\ n'.\ \exists a'.\ sourcenode\ a = sourcenode\ a' \wedge valid\text{-}edge\ a' \wedge targetnode\ a' = n')$$

⟨*proof*⟩

lemma *slice-kind-only-one-True-edge*:
assumes *sourcenode a* = *sourcenode a'* **and** *targetnode a* ≠ *targetnode a'*
and *valid-edge a* **and** *valid-edge a'* **and** *slice-kind S a* = $(\lambda s.\ True)_{\surd}$
shows *slice-kind S a'* = $(\lambda s.\ False)_{\surd}$
⟨*proof*⟩

lemma *slice-deterministic*:
assumes *valid-edge a* **and** *valid-edge a'*
and *sourcenode a* = *sourcenode a'* **and** *targetnode a* ≠ *targetnode a'*
obtains *Q Q'* **where** *slice-kind S a* = $(Q)_{\surd}$ **and** *slice-kind S a'* = $(Q')_{\surd}$
and $\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s)$
⟨*proof*⟩

3.3.2 Observable and silent moves

inductive *silent-move* ::
'node set ⇒ (*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'state* ⇒ *'edge* ⇒
'node ⇒ *'state* ⇒ *bool* (-,- ⊢ '(-,-') −-→$_\tau$ '(-,-') [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

where *silent-moveI*:
⟦*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s'*; *sourcenode a* ∉ *backward-slice S*;
valid-edge a⟧
⟹ *S*,*f* ⊢ (*sourcenode a*,*s*) −*a*→$_\tau$ (*targetnode a*,*s'*)

inductive *silent-moves* ::
'node set ⇒ (*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'state* ⇒ *'edge list* ⇒
'node ⇒ *'state* ⇒ *bool* (-,- ⊢ '(-,-') =-⇒$_\tau$ '(-,-') [*51*,*50*,*0*,*0*,*50*,*0*,*0*] *51*)

where *silent-moves-Nil*: *S*,*f* ⊢ (*n*,*s*) =[]⇒$_\tau$ (*n*,*s*)

| *silent-moves-Cons*:
⟦*S*,*f* ⊢ (*n*,*s*) −*a*→$_\tau$ (*n'*,*s'*); *S*,*f* ⊢ (*n'*,*s'*) =*as*⇒$_\tau$ (*n''*,*s''*)⟧
⟹ *S*,*f* ⊢ (*n*,*s*) =*a*#*as*⇒$_\tau$ (*n''*,*s''*)

lemma *silent-moves-obs-slice*:
⟦*S*,*f* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n'*,*s'*); *nx* ∈ *obs n'* (*backward-slice S*)⟧
⟹ *nx* ∈ *obs n* (*backward-slice S*)
⟨*proof*⟩

lemma *silent-moves-preds-transfers-path*:

$[\![S,f \vdash (n,s) =as\Rightarrow_\tau (n',s'); \mathit{valid\text{-}node}\ n]\!]$
$\Longrightarrow \mathit{preds}\ (\mathit{map}\ f\ as)\ s \wedge \mathit{transfers}\ (\mathit{map}\ f\ as)\ s = s' \wedge n\ -as\rightarrow *\ n'$
⟨*proof*⟩

lemma *obs-silent-moves*:
assumes *obs n* (*backward-slice S*) = {*n'*}
obtains *as* **where** *S*,*slice-kind S* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n'*,*s*)
⟨*proof*⟩

inductive *observable-move* ::
'node set ⇒ (*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'state* ⇒ *'edge* ⇒
'node ⇒ *'state* ⇒ *bool* (-,- ⊢ '(-,-') −-→ '(-,-') [*51*,*50*,*0*,*0*,*50*,*0*,*0] *51*)

where *observable-moveI*:
[*pred* (*f a*) *s*; *transfer* (*f a*) *s* = *s'*; *sourcenode a* ∈ *backward-slice S*;
valid-edge a]
⟹ *S*,*f* ⊢ (*sourcenode a*,*s*) −*a*→ (*targetnode a*,*s'*)

inductive *observable-moves* ::
'node set ⇒ (*'edge* ⇒ *'state edge-kind*) ⇒ *'node* ⇒ *'state* ⇒ *'edge list* ⇒
'node ⇒ *'state* ⇒ *bool* (-,- ⊢ '(-,-') =-⇒ '(-,-') [*51*,*50*,*0*,*0*,*50*,*0*,*0] *51*)

where *observable-moves-snoc*:
[*S*,*f* ⊢ (*n*,*s*) =*as*⇒$_\tau$ (*n'*,*s'*); *S*,*f* ⊢ (*n'*,*s'*) −*a*→ (*n''*,*s''*)]
⟹ *S*,*f* ⊢ (*n*,*s*) =*as*@[*a*]⇒ (*n''*,*s''*)

lemma *observable-move-notempty*:
[*S*,*f* ⊢ (*n*,*s*) =*as*⇒ (*n'*,*s'*); *as* = []] ⟹ *False*
⟨*proof*⟩

lemma *silent-move-observable-moves*:
[*S*,*f* ⊢ (*n''*,*s''*) =*as*⇒ (*n'*,*s'*); *S*,*f* ⊢ (*n*,*s*) −*a*→$_\tau$ (*n''*,*s''*)]
⟹ *S*,*f* ⊢ (*n*,*s*) =*a*#*as*⇒ (*n'*,*s'*)
⟨*proof*⟩

lemma *observable-moves-preds-transfers-path*:
S,*f* ⊢ (*n*,*s*) =*as*⇒ (*n'*,*s'*)
⟹ *preds* (*map f as*) *s* ∧ *transfers* (*map f as*) *s* = *s'* ∧ *n* −*as*→* *n'*
⟨*proof*⟩

3.3.3 Relevant variables

inductive-set *relevant-vars* :: *'node set* ⇒ *'node* ⇒ *'var set* (*rv* -)

for $S :: 'node\ set$ **and** $n :: 'node$

where *rvI*:
$[\![n\ -as\rightarrow*\ n'; n' \in backward\text{-}slice\ S; V \in Use\ n';$
$\forall nx \in set(sourcenodes\ as).\ V \notin Def\ nx]\!]$
$\Longrightarrow V \in rv\ S\ n$

lemma *rvE*:
assumes *rv*:$V \in rv\ S\ n$
obtains *as* n' **where** $n\ -as\rightarrow*\ n'$ **and** $n' \in backward\text{-}slice\ S$ **and** $V \in Use\ n'$
and $\forall nx \in set(sourcenodes\ as).\ V \notin Def\ nx$
⟨*proof*⟩

lemma *eq-obs-in-rv*:
assumes *obs-eq*:$obs\ n\ (backward\text{-}slice\ S) = obs\ n'\ (backward\text{-}slice\ S)$
and $x \in rv\ S\ n$ **shows** $x \in rv\ S\ n'$
⟨*proof*⟩

lemma *closed-eq-obs-eq-rvs*:
fixes $S :: 'node\ set$
assumes *valid-node* n **and** *valid-node* n'
and *obs-eq*:$obs\ n\ (backward\text{-}slice\ S) = obs\ n'\ (backward\text{-}slice\ S)$
shows $rv\ S\ n = rv\ S\ n'$
⟨*proof*⟩

lemma *rv-edge-slice-kinds*:
assumes *valid-edge* a **and** $sourcenode\ a = n$ **and** $targetnode\ a = n''$
and $\forall\ V{\in}rv\ S\ n.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
and $preds\ (slice\text{-}kinds\ S\ (a\#as))\ s$ **and** $preds\ (slice\text{-}kinds\ S\ (a\#asx))\ s'$
shows $\forall\ V{\in}rv\ S\ n''.\ state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s)\ V =$
$state\text{-}val\ (transfer\ (slice\text{-}kind\ S\ a)\ s')\ V$
⟨*proof*⟩

lemma *rv-branching-edges-slice-kinds-False*:
assumes *valid-edge* a **and** *valid-edge* ax
and $sourcenode\ a = n$ **and** $sourcenode\ ax = n$
and $targetnode\ a = n''$ **and** $targetnode\ ax \neq n''$
and $preds\ (slice\text{-}kinds\ S\ (a\#as))\ s$ **and** $preds\ (slice\text{-}kinds\ S\ (ax\#asx))\ s'$
and $\forall\ V{\in}rv\ S\ n.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
shows *False*
⟨*proof*⟩

3.3.4 The set *WS*

inductive-set *WS* :: *'node set* ⇒ ((*'node* × *'state*) × (*'node* × *'state*)) *set*
for *S* :: *'node set*
where *WSI*:⟦*obs n* (*backward-slice S*) = *obs n'* (*backward-slice S*);
∀ *V* ∈ *rv S n*. *state-val s V* = *state-val s' V*;
valid-node n; *valid-node n'*⟧
⟹ ((*n*,*s*),(*n'*,*s'*)) ∈ *WS S*

lemma *WSD*:
((*n*,*s*),(*n'*,*s'*)) ∈ *WS S*
⟹ *obs n* (*backward-slice S*) = *obs n'* (*backward-slice S*) ∧
(∀ *V* ∈ *rv S n*. *state-val s V* = *state-val s' V*) ∧
valid-node n ∧ *valid-node n'*
⟨*proof*⟩

lemma *WS-silent-move*:
assumes ((*n*$_1$,*s*$_1$),(*n*$_2$,*s*$_2$)) ∈ *WS S* **and** *S*,*kind* ⊢ (*n*$_1$,*s*$_1$) −*a*→$_\tau$ (*n*$_1$',*s*$_1$') **and** *obs n*$_1$' (*backward-slice S*) ≠ {} **shows** ((*n*$_1$',*s*$_1$'),(*n*$_2$,*s*$_2$)) ∈ *WS S*
⟨*proof*⟩

lemma *WS-silent-moves*:
⟦*S*,*f* ⊢ (*n*$_1$,*s*$_1$) =*as*⇒$_\tau$ (*n*$_1$',*s*$_1$'); ((*n*$_1$,*s*$_1$),(*n*$_2$,*s*$_2$)) ∈ *WS S*; *f* = *kind*;
obs n$_1$' (*backward-slice S*) ≠ {}⟧
⟹ ((*n*$_1$',*s*$_1$'),(*n*$_2$,*s*$_2$)) ∈ *WS S*
⟨*proof*⟩

lemma *WS-observable-move*:
assumes ((*n*$_1$,*s*$_1$),(*n*$_2$,*s*$_2$)) ∈ *WS S* **and** *S*,*kind* ⊢ (*n*$_1$,*s*$_1$) −*a*→ (*n*$_1$',*s*$_1$')
obtains *as* **where** ((*n*$_1$',*s*$_1$'),(*n*$_1$',*transfer* (*slice-kind S a*) *s*$_2$)) ∈ *WS S*
and *S*,*slice-kind S* ⊢ (*n*$_2$,*s*$_2$) =*as*@[*a*]⇒ (*n*$_1$',*transfer* (*slice-kind S a*) *s*$_2$)
⟨*proof*⟩

definition *is-weak-sim* ::
((*'node* × *'state*) × (*'node* × *'state*)) *set* ⇒ *'node set* ⇒ *bool*
where *is-weak-sim R S* ≡
∀ *n*$_1$ *s*$_1$ *n*$_2$ *s*$_2$ *n*$_1$' *s*$_1$' *as*. ((*n*$_1$,*s*$_1$),(*n*$_2$,*s*$_2$)) ∈ *R* ∧ *S*,*kind* ⊢ (*n*$_1$,*s*$_1$) =*as*⇒ (*n*$_1$',*s*$_1$')
⟶ (∃ *n*$_2$' *s*$_2$' *as'*. ((*n*$_1$',*s*$_1$'),(*n*$_2$',*s*$_2$')) ∈ *R* ∧
S,*slice-kind S* ⊢ (*n*$_2$,*s*$_2$) =*as'*⇒ (*n*$_2$',*s*$_2$'))

lemma *WS-weak-sim*:
assumes ((*n*$_1$,*s*$_1$),(*n*$_2$,*s*$_2$)) ∈ *WS S*

and S,*kind* $\vdash$ (n_1,s_1) $=as\Rightarrow$ (n_1',s_1')
shows $((n_1',s_1'),(n_1',$*transfer* $($*slice-kind* S $($*last* $as))$ $s_2))$ $\in$ *WS* S $\wedge$
$(\exists\, as'.$ S,*slice-kind* $S \vdash (n_2,s_2)$ $=as'@[$*last* $as]\Rightarrow$
$(n_1',$*transfer* $($*slice-kind* S $($*last* $as))$ $s_2))$
⟨*proof*⟩

The following lemma states the correctness of static intraprocedural slicing: the simulation *WS* S is a desired weak simulation

theorem *WS-is-weak-sim*:*is-weak-sim* $($*WS* $S)$ S
⟨*proof*⟩

3.3.5 n $-as\rightarrow*$ n' and transitive closure of S,f $\vdash$ (n,s) $=as\Rightarrow_\tau$ (n',s')

inductive *trans-observable-moves* ::
'node set $\Rightarrow$ $($*'edge* $\Rightarrow$ *'state edge-kind*$)$ $\Rightarrow$ *'node* $\Rightarrow$ *'state* $\Rightarrow$ *'edge list* $\Rightarrow$
'node $\Rightarrow$ *'state* $\Rightarrow$ *bool* $(\text{-},\text{-} \vdash \text{'}(\text{-},\text{-}') =\text{-}\Rightarrow* \text{'}(\text{-},\text{-}')$ $[51,50,0,0,50,0,0]$ $51)$

where *tom-Nil*:
S,f $\vdash$ (n,s) $=[]\Rightarrow*$ (n,s)

| *tom-Cons*:
$[\![S,f \vdash (n,s)$ $=as\Rightarrow$ (n',s'); $S,f \vdash (n',s')$ $=as'\Rightarrow*$ $(n'',s'')]\!]$
$\Longrightarrow$ $S,f \vdash (n,s)$ $=($*last* $as)\#as'\Rightarrow*$ (n'',s'')

definition *slice-edges* :: *'node set* $\Rightarrow$ *'edge list* $\Rightarrow$ *'edge list*
where *slice-edges* S *as* $\equiv$ $[a \leftarrow as.$ *sourcenode* $a \in$ *backward-slice* $S]$

lemma *silent-moves-no-slice-edges*:
S,f $\vdash$ (n,s) $=as\Rightarrow_\tau$ (n',s') $\Longrightarrow$ *slice-edges* S *as* $=$ $[]$
⟨*proof*⟩

lemma *observable-moves-last-slice-edges*:
S,f $\vdash$ (n,s) $=as\Rightarrow$ (n',s') $\Longrightarrow$ *slice-edges* S *as* $=$ $[$*last* *as*$]$
⟨*proof*⟩

lemma *slice-edges-no-nodes-in-slice*:
slice-edges S *as* $=$ $[]$
$\Longrightarrow$ $\forall$ *nx* $\in$ *set*$($*sourcenodes* *as*$)$. *nx* $\notin$ $($*backward-slice* $S)$
⟨*proof*⟩

lemma *sliced-path-determ*:
$[\![n$ $-as\rightarrow*$ n'; n $-as'\rightarrow*$ n'; *slice-edges* S *as* $=$ *slice-edges* S *as'*;

$\mathit{preds}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as})\ s;\ \mathit{preds}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as}')\ s';\ n' \in S;$
$\forall\, V \in \mathit{rv}\ S\ n.\ \mathit{state\text{-}val}\ s\ V = \mathit{state\text{-}val}\ s'\ V\rrbracket \Longrightarrow \mathit{as} = \mathit{as}'$
⟨*proof*⟩

lemma *path-trans-observable-moves*:
assumes $n\ -\mathit{as}\rightarrow*\ n'$ **and** $\mathit{preds}\ (\mathit{kinds}\ \mathit{as})\ s$ **and** $\mathit{transfers}\ (\mathit{kinds}\ \mathit{as})\ s = s'$
obtains $n''\ s''\ \mathit{as}'\ \mathit{as}''$ **where** $S,\mathit{kind} \vdash (n,s)\ =\mathit{slice\text{-}edges}\ S\ \mathit{as}\Rightarrow*\ (n'',s'')$
and $S,\mathit{kind} \vdash (n'',s'')\ =\mathit{as}'\Rightarrow_{\tau}\ (n',s')$
and $\mathit{slice\text{-}edges}\ S\ \mathit{as} = \mathit{slice\text{-}edges}\ S\ \mathit{as}''$ **and** $n\ -\mathit{as}''@\mathit{as}'\rightarrow*\ n'$
⟨*proof*⟩

lemma *WS-weak-sim-trans*:
assumes $((n_1,s_1),(n_2,s_2)) \in \mathit{WS}\ S$
and $S,\mathit{kind} \vdash (n_1,s_1)\ =\mathit{as}\Rightarrow*\ (n_1',s_1')$ **and** $\mathit{as} \neq []$
shows $((n_1',s_1'),(n_1',\mathit{transfers}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as})\ s_2)) \in \mathit{WS}\ S \wedge$
$S,\mathit{slice\text{-}kind}\ S \vdash (n_2,s_2)\ =\mathit{as}\Rightarrow*\ (n_1',\mathit{transfers}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as})\ s_2)$
⟨*proof*⟩

lemma *transfers-slice-kinds-slice-edges*:
$\mathit{transfers}\ (\mathit{slice\text{-}kinds}\ S\ (\mathit{slice\text{-}edges}\ S\ \mathit{as}))\ s = \mathit{transfers}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as})\ s$
⟨*proof*⟩

lemma *trans-observable-moves-preds*:
assumes $S,f \vdash (n,s)\ =\mathit{as}\Rightarrow*\ (n',s')$ **and** $\mathit{valid\text{-}node}\ n$
obtains as' **where** $\mathit{preds}\ (\mathit{map}\ f\ \mathit{as}')\ s$ **and** $\mathit{slice\text{-}edges}\ S\ \mathit{as}' = \mathit{as}$
and $n\ -\mathit{as}'\rightarrow*\ n'$
⟨*proof*⟩

lemma *exists-sliced-path-preds*:
assumes $n\ -\mathit{as}\rightarrow*\ n'$ **and** $\mathit{slice\text{-}edges}\ S\ \mathit{as} = []$ **and** $n' \in \mathit{backward\text{-}slice}\ S$
obtains as' **where** $n\ -\mathit{as}'\rightarrow*\ n'$ **and** $\mathit{preds}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as}')\ s$
and $\mathit{slice\text{-}edges}\ S\ \mathit{as}' = []$
⟨*proof*⟩

theorem *fundamental-property-of-static-slicing*:
assumes $\mathit{path}{:}n\ -\mathit{as}\rightarrow*\ n'$ **and** $\mathit{preds}{:}\mathit{preds}\ (\mathit{kinds}\ \mathit{as})\ s$ **and** $n' \in S$
obtains as' **where** $\mathit{preds}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as}')\ s$
and $(\forall\, V \in \mathit{Use}\ n'.\ \mathit{state\text{-}val}\ (\mathit{transfers}\ (\mathit{slice\text{-}kinds}\ S\ \mathit{as}')\ s)\ V =$
$\mathit{state\text{-}val}\ (\mathit{transfers}\ (\mathit{kinds}\ \mathit{as})\ s)\ V)$
and $\mathit{slice\text{-}edges}\ S\ \mathit{as} = \mathit{slice\text{-}edges}\ S\ \mathit{as}'$ **and** $n\ -\mathit{as}'\rightarrow*\ n'$
⟨*proof*⟩

end

3.3.6 The fundamental property of (static) slicing related to the semantics

locale *BackwardSlice-wf* =
BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice +
CFG-semantics-wf sourcenode targetnode kind valid-edge Entry sem identifies
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *sem* :: *'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *bool*
(((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩)) [*0*,*0*,*0*,*0*] *81*)
and *identifies* :: *'node* ⇒ *'com* ⇒ *bool* (- ≙ - [*51*, *0*] *80*)

begin

theorem *fundamental-property-of-path-slicing-semantically*:
assumes *n* ≙ *c* **and** ⟨*c*,*s*⟩ ⇒ ⟨*c'*,*s'*⟩
obtains *n'* *as* **where** *n* −*as*→∗ *n'* **and** *preds* (*slice-kinds* {*n'*} *as*) *s* **and** *n'* ≙ *c'*
and ∀ *V* ∈ *Use n'*. *state-val* (*transfers* (*slice-kinds* {*n'*} *as*) *s*) *V* = *state-val s'* *V*
⟨*proof*⟩

end

end

3.4 Static Standard Control Dependence

theory *StandardControlDependence* **imports**
../Basic/Postdomination
../Basic/DynStandardControlDependence
begin

context *Postdomination* **begin**

Definition and some lemmas

definition *standard-control-dependence* :: *'node* ⇒ *'node* ⇒ *bool*

(- $controls_s$ - [51,0])
where *standard-control-dependences-eq*:$n\ controls_s\ n' \equiv \exists\, as.\ n\ controls_s\ n'\ via\ as$

lemma *standard-control-dependence-def*:$n\ controls_s\ n' =$
$(\exists\, a\ a'\ as.\ (n' \notin set(sourcenodes\ (a\#as))) \wedge (n\ -a\#as\rightarrow*\ n') \wedge$
$(n'\ postdominates\ (targetnode\ a)) \wedge$
$(valid\text{-}edge\ a') \wedge (sourcenode\ a' = n) \wedge$
$(\neg\ n'\ postdominates\ (targetnode\ a')))$
⟨*proof*⟩

lemma *Exit-not-standard-control-dependent*:
$n\ controls_s$ (-*Exit*-) $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *standard-control-dependence-def-variant*:
$n\ controls_s\ n' = (\exists\, as.\ (n\ -as\rightarrow*\ n') \wedge (n \neq n') \wedge$
$(\neg\ n'\ postdominates\ n) \wedge (n' \notin set(sourcenodes\ as)) \wedge$
$(\forall\, n'' \in set(targetnodes\ as).\ n'\ postdominates\ n''))$
⟨*proof*⟩

lemma *inner-node-standard-control-dependence-predecessor*:
assumes *inner-node* n (-*Entry*-) $-as\rightarrow*\ n\ n\ -as'\rightarrow*$ (-*Exit*-)
obtains n' **where** $n'\ controls_s\ n$
⟨*proof*⟩

end

end

3.5 Static Weak Control Dependence

theory *WeakControlDependence* **imports**
../Basic/Postdomination
../Basic/DynWeakControlDependence
begin

context *StrongPostdomination* **begin**

definition
weak-control-dependence :: $'node \Rightarrow 'node \Rightarrow bool$
(- *weakly controls* - [51,0])
where *weak-control-dependences-eq*:
$n\ weakly\ controls\ n' \equiv \exists\, as.\ n\ weakly\ controls\ n'\ via\ as$

lemma
weak-control-dependence-def:$n\ weakly\ controls\ n' =$

$(\exists\, a\ a'\ as.\ (n' \notin set(sourcenodes\ (a\#as))) \wedge (n\ -a\#as\rightarrow*\ n') \wedge$
$(n'\ strongly{-}postdominates\ (targetnode\ a)) \wedge$
$(valid\text{-}edge\ a') \wedge (sourcenode\ a' = n) \wedge$
$(\neg\ n'\ strongly{-}postdominates\ (targetnode\ a')))$
⟨*proof*⟩

lemma *Exit-not-weak-control-dependent*:
n weakly controls (*-Exit-*) ⟹ *False*
⟨*proof*⟩

end

end

3.6 Program Dependence Graph

theory *PDG* **imports**
DataDependence
StandardControlDependence
WeakControlDependence
../Basic/CFGExit-wf
begin

locale *PDG* =
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (*'('-Exit'-')*) +
fixes *control-dependence* :: *'node* ⇒ *'node* ⇒ *bool*
(*- controls -* [*51*,*0*])
assumes *Exit-not-control-dependent*:*n controls n'* ⟹ *n'* ≠ (*-Exit-*)
assumes *control-dependence-path*:
n controls n'
⟹ ∃ *as*. *CFG.path sourcenode targetnode valid-edge n as n'* ∧ *as* ≠ []

begin

inductive *cdep-edge* :: *'node* ⇒ *'node* ⇒ *bool*
(*-* $\longrightarrow_{cd}$ *-* [*51*,*0*] *80*)
and *ddep-edge* :: *'node* ⇒ *'var* ⇒ *'node* ⇒ *bool*
(*-* $-\!\!\rightarrow_{dd}$ *-* [*51*,*0*,*0*] *80*)
and *PDG-edge* :: *'node* ⇒ *'var option* ⇒ *'node* ⇒ *bool*

where

$n \longrightarrow_{cd} n' == PDG\text{-}edge\ n\ None\ n'$
$|\ n\ -V\rightarrow_{dd} n' == PDG\text{-}edge\ n\ (Some\ V)\ n'$

$|$ *PDG-cdep-edge*:
n *controls* $n' \Longrightarrow n \longrightarrow_{cd} n'$

$|$ *PDG-ddep-edge*:
n *influences* V *in* $n' \Longrightarrow n\ -V\rightarrow_{dd} n'$

inductive *PDG-path* :: $'node \Rightarrow 'node \Rightarrow bool$
$(\text{-} \longrightarrow_{d}* \text{-}\ [51,0]\ 80)$

where *PDG-path-Nil*:
$valid\text{-}node\ n \Longrightarrow n \longrightarrow_{d}* n$

$|$ *PDG-path-Append-cdep*:
$[\![n \longrightarrow_{d}* n''; n'' \longrightarrow_{cd} n']\!] \Longrightarrow n \longrightarrow_{d}* n'$

$|$ *PDG-path-Append-ddep*:
$[\![n \longrightarrow_{d}* n''; n''\ -V\rightarrow_{dd} n']\!] \Longrightarrow n \longrightarrow_{d}* n'$

lemma *PDG-path-cdep*:$n \longrightarrow_{cd} n' \Longrightarrow n \longrightarrow_{d}* n'$
⟨*proof*⟩

lemma *PDG-path-ddep*:$n\ -V\rightarrow_{dd} n' \Longrightarrow n \longrightarrow_{d}* n'$
⟨*proof*⟩

lemma *PDG-path-Append*:
$[\![n'' \longrightarrow_{d}* n'; n \longrightarrow_{d}* n'']\!] \Longrightarrow n \longrightarrow_{d}* n'$
⟨*proof*⟩

lemma *PDG-cdep-edge-CFG-path*:
assumes $n \longrightarrow_{cd} n'$ **obtains** as **where** $n\ -as\rightarrow* n'$ **and** $as \neq []$
⟨*proof*⟩

lemma *PDG-ddep-edge-CFG-path*:
assumes $n\ -V\rightarrow_{dd} n'$ **obtains** as **where** $n\ -as\rightarrow* n'$ **and** $as \neq []$
⟨*proof*⟩

lemma *PDG-path-CFG-path*:
assumes $n \longrightarrow_{d}* n'$ **obtains** as **where** $n\ -as\rightarrow* n'$
⟨*proof*⟩

lemma *PDG-path-Exit*:$[\![n \longrightarrow_{d}* n'; n' = (\text{-}Exit\text{-})]\!] \Longrightarrow n = (\text{-}Exit\text{-})$

⟨*proof*⟩

lemma *PDG-path-not-inner*:
⟦$n \longrightarrow_d* n'$; ¬ *inner-node* n'⟧ ⟹ $n = n'$
⟨*proof*⟩

3.6.1 Definition of the static backward slice

Node: instead of a single node, we calculate the backward slice of a set of nodes.

definition *PDG-BS* :: *'node set* ⇒ *'node set*
where *PDG-BS S* ≡ {n'. ∃ n. $n' \longrightarrow_d* n$ ∧ n ∈ *S* ∧ *valid-node n*}

lemma *PDG-BS-valid-node*:n ∈ *PDG-BS S* ⟹ *valid-node n*
⟨*proof*⟩

lemma *Exit-PDG-BS*:n ∈ *PDG-BS* {(*-Exit-*)} ⟹ n = (*-Exit-*)
⟨*proof*⟩

end

3.6.2 Instantiate static PDG

Standard control dependence

locale *StandardControlDependencePDG* =
Postdomination sourcenode targetnode kind valid-edge Entry Exit +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (*'('-Exit'-')*)

begin

lemma *PDG-scd*:
PDG sourcenode targetnode kind valid-edge (*-Entry-*)
Def Use state-val (*-Exit-*) *standard-control-dependence*
⟨*proof*⟩⟨*proof*⟩
end

Weak control dependence

locale *WeakControlDependencePDG* =
StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit +

CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *Exit* :: *'node* (*'('-Exit'-')*)

begin

lemma *PDG-wcd*:
PDG sourcenode targetnode kind valid-edge (*-Entry-*)
Def Use state-val (*-Exit-*) *weak-control-dependence*
⟨*proof*⟩⟨*proof*⟩
end

end

3.7 Weak Order Dependence

theory *WeakOrderDependence* **imports** *../Basic/CFG DataDependence* **begin**

Weak order dependence is just defined as a static control dependence

3.7.1 Definition and some lemmas

definition (**in** *CFG*) *weak-order-dependence* :: *'node* ⇒ *'node* ⇒ *'node* ⇒ *bool*
(*-* $\longrightarrow_{wod}$ *-*,*-*)
where *wod-def*:$n \longrightarrow_{wod} n_1,n_2 \equiv ((n_1 \neq n_2) \wedge$
$(\exists\, as.\ (n\ -as\rightarrow*\ n_1) \wedge (n_2 \notin set\ (sourcenodes\ as))) \wedge$
$(\exists\, as.\ (n\ -as\rightarrow*\ n_2) \wedge (n_1 \notin set\ (sourcenodes\ as))) \wedge$
$(\exists\, a.\ (valid\text{-}edge\ a) \wedge (n = sourcenode\ a) \wedge$
$((\exists\, as.\ (targetnode\ a\ -as\rightarrow*\ n_1) \wedge$
$(\forall\, as'.\ (targetnode\ a\ -as'\rightarrow*\ n_2) \longrightarrow n_1 \in set(sourcenodes\ as'))) \vee$
$(\exists\, as.\ (targetnode\ a\ -as\rightarrow*\ n_2) \wedge$
$(\forall\, as'.\ (targetnode\ a\ -as'\rightarrow*\ n_1) \longrightarrow n_2 \in set(sourcenodes\ as'))))))$

inductive-set (**in** *CFG-wf*) *wod-backward-slice* :: *'node set* ⇒ *'node set*
for *S* :: *'node set*
where *refl*:⟦*valid-node n*; *n* ∈ *S*⟧ ⟹ *n* ∈ *wod-backward-slice S*

| *cd-closed*:
⟦$n' \longrightarrow_{wod} n_1,n_2$; n_1 ∈ *wod-backward-slice S*; n_2 ∈ *wod-backward-slice S*⟧
⟹ *n'* ∈ *wod-backward-slice S*

| *dd-closed*:⟦*n' influences V in n''*; *n''* ∈ *wod-backward-slice S*⟧
⟹ *n'* ∈ *wod-backward-slice S*

lemma (**in** *CFG-wf*)
wod-backward-slice-valid-node:$n \in$ *wod-backward-slice* $S \Longrightarrow$ *valid-node* n
⟨*proof*⟩

end

3.8 Instantiate framework with control dependences

theory *CDepInstantiations* **imports**
Slice
PDG
WeakOrderDependence
begin

3.8.1 Standard control dependence

context *StandardControlDependencePDG* **begin**

lemma *Exit-in-obs-slice-node*:(*-Exit-*) $\in$ *obs* n' (*PDG-BS* S) $\Longrightarrow$ (*-Exit-*) $\in S$
⟨*proof*⟩

abbreviation *PDG-path'* :: *'node* $\Rightarrow$ *'node* $\Rightarrow$ *bool* (- $\longrightarrow_{d*}$ - [*51*,*0*] *80*)
where $n \longrightarrow_{d*} n' \equiv$ *PDG.PDG-path sourcenode targetnode valid-edge Def Use standard-control-dependence* n n'

lemma *cd-closed*:
$[\![n' \in$ *PDG-BS* S; n *controls*$_s$ $n'$$]\!] \Longrightarrow n \in$ *PDG-BS* S
⟨*proof*⟩

lemma *obs-postdominate*:
assumes $n \in$ *obs* n' (*PDG-BS* S) **and** $n \neq$ (*-Exit-*) **shows** n *postdominates* n'
⟨*proof*⟩

lemma *obs-singleton*:($\exists\, m.$ *obs* n (*PDG-BS* S) $= \{m\}$) $\vee$ *obs* n (*PDG-BS* S) $=$ $\{\}$
⟨*proof*⟩

lemma *PDGBackwardSliceCorrect*:
BackwardSlice sourcenode targetnode kind valid-edge
(*-Entry-*) *Def Use state-val PDG-BS*
⟨*proof*⟩

end

3.8.2 Weak control dependence

context *WeakControlDependencePDG* **begin**

lemma *Exit-in-obs-slice-node*:$(\text{-}Exit\text{-}) \in obs\ n'\ (PDG\text{-}BS\ S) \Longrightarrow (\text{-}Exit\text{-}) \in S$
⟨*proof*⟩

lemma *cd-closed*:
$[\![n' \in PDG\text{-}BS\ S;\ n\ weakly\ controls\ n']\!] \Longrightarrow n \in PDG\text{-}BS\ S$
⟨*proof*⟩

lemma *obs-strong-postdominate*:
assumes $n \in obs\ n'\ (PDG\text{-}BS\ S)$ **and** $n \neq (\text{-}Exit\text{-})$
shows $n\ strongly{-}postdominates\ n'$
⟨*proof*⟩

lemma *obs-singleton*:$(\exists\, m.\ obs\ n\ (PDG\text{-}BS\ S) = \{m\}) \vee obs\ n\ (PDG\text{-}BS\ S) = \{\}$
⟨*proof*⟩

lemma *WeakPDGBackwardSliceCorrect*:
BackwardSlice sourcenode targetnode kind valid-edge
(*-Entry-*) *Def Use state-val PDG-BS*
⟨*proof*⟩

end

3.8.3 Weak order dependence

context *CFG-wf* **begin**

lemma *obs-singleton*:

shows $(\exists\, m.\ obs\ n\ (wod\text{-}backward\text{-}slice\ S) = \{m\}) \vee obs\ n\ (wod\text{-}backward\text{-}slice\ S) = \{\}$
⟨*proof*⟩

lemma *WODBackwardSliceCorrect*:
BackwardSlice sourcenode targetnode kind valid-edge
(*-Entry-*) *Def Use state-val wod-backward-slice*
⟨*proof*⟩

end

end

3.9 Relations between control dependences

theory *ControlDependenceRelations*
imports *WeakOrderDependence StandardControlDependence*
begin

context *StrongPostdomination* **begin**

lemma *standard-control-implies-weak-order*:
assumes n $controls_s$ n' **shows** $n \longrightarrow_{wod} n'$,(*-Exit-*)
⟨*proof*⟩

end

end

Chapter 4

Instantiating the Framework with a simple While-Language

4.1 Commands

theory *Com* **imports** *Main* **begin**

4.1.1 Variables and Values

type-synonym *vname* = *string* — names for variables

datatype *val*
= *Bool bool* — Boolean value
| *Intg int* — integer value

abbreviation *true* == *Bool True*
abbreviation *false* == *Bool False*

4.1.2 Expressions and Commands

datatype *bop* = *Eq* | *And* | *Less* | *Add* | *Sub* — names of binary operations

datatype *expr*
= *Val val* — value
| *Var vname* — local variable
| *BinOp expr bop expr* (- ≪-≫ - [*80*,*0*,*81*] *80*) — binary operation

fun *binop* :: *bop* ⇒ *val* ⇒ *val* ⇒ *val option*
where *binop Eq* v_1 v_2 = *Some*(*Bool*(v_1 = v_2))
| *binop And* (*Bool* b_1) (*Bool* b_2) = *Some*(*Bool*(b_1 ∧ b_2))
| *binop Less* (*Intg* i_1) (*Intg* i_2) = *Some*(*Bool*(i_1 < i_2))
| *binop Add* (*Intg* i_1) (*Intg* i_2) = *Some*(*Intg*(i_1 + i_2))

```
  | binop Sub (Intg i1) (Intg i2)  = Some(Intg(i1 − i2))
  | binop bop v1 v2                = None
```

```
datatype cmd
  = Skip
  | LAss vname expr        (-:=- [70,70] 70)  — local assignment
  | Seq cmd cmd            (-;;/ - [61,60] 60)
  | Cond expr cmd cmd      (if '(-') -/ else - [80,79,79] 70)
  | While expr cmd         (while '(-') - [80,79] 70)
```

```
fun num-inner-nodes :: cmd ⇒ nat (#:-)
where #:Skip               = 1
  | #:(V:=e)               = 2
  | #:(c1;;c2)             = #:c1 + #:c2
  | #:(if (b) c1 else c2) = #:c1 + #:c2 + 1
  | #:(while (b) c)        = #:c + 2
```

```
lemma num-inner-nodes-gr-0:#:c > 0
⟨proof⟩

lemma [dest]:#:c = 0 ⟹ False
⟨proof⟩
```

4.1.3 The state

```
type-synonym state = vname ⇀ val

fun interpret :: expr ⇒ state ⇒ val option
where Val: interpret (Val v) s = Some v
  | Var: interpret (Var V) s = s V
  | BinOp: interpret (e1≪bop≫e2) s =
    (case interpret e1 s of None ⇒ None
                          | Some v1 ⇒ (case interpret e2 s of None ⇒ None
                                                            | Some v2 ⇒ (
      case binop bop v1 v2 of None ⇒ None | Some v ⇒ Some v)))

end
```

4.2 CFG

```
theory WCFG imports Com ../Basic/BasicDefs begin
```

4.2.1 CFG nodes

datatype *w-node* = *Node nat* (*'*(*'*- - *'*-*'*))
| *Entry* (*'*(*'*-*Entry*-*'*-*'*))
| *Exit* (*'*(*'*-*Exit*-*'*-*'*))

fun *label-incr* :: *w-node* ⇒ *nat* ⇒ *w-node* (- ⊕ - *60*)
where (- *l* -) ⊕ *i* = (- *l* + *i* -)
| (-*Entry*-) ⊕ *i* = (-*Entry*-)
| (-*Exit*-) ⊕ *i* = (-*Exit*-)

lemma *Exit-label-incr* [*dest*]: (-*Exit*-) = *n* ⊕ *i* ⟹ *n* = (-*Exit*-)
⟨*proof*⟩

lemma *label-incr-Exit* [*dest*]: *n* ⊕ *i* = (-*Exit*-) ⟹ *n* = (-*Exit*-)
⟨*proof*⟩

lemma *Entry-label-incr* [*dest*]: (-*Entry*-) = *n* ⊕ *i* ⟹ *n* = (-*Entry*-)
⟨*proof*⟩

lemma *label-incr-Entry* [*dest*]: *n* ⊕ *i* = (-*Entry*-) ⟹ *n* = (-*Entry*-)
⟨*proof*⟩

lemma *label-incr-inj*:
n ⊕ *c* = *n*′ ⊕ *c* ⟹ *n* = *n*′
⟨*proof*⟩

lemma *label-incr-simp*:*n* ⊕ *i* = *m* ⊕ (*i* + *j*) ⟹ *n* = *m* ⊕ *j*
⟨*proof*⟩

lemma *label-incr-simp-rev*:*m* ⊕ (*j* + *i*) = *n* ⊕ *i* ⟹ *m* ⊕ *j* = *n*
⟨*proof*⟩

lemma *label-incr-start-Node-smaller*:
(- *l* -) = *n* ⊕ *i* ⟹ *n* = (-(*l* − *i*)-)
⟨*proof*⟩

lemma *label-incr-ge*:(- *l* -) = *n* ⊕ *i* ⟹ *l* ≥ *i*
⟨*proof*⟩

lemma *label-incr-0* [*dest*]:
⟦(-*0*-) = *n* ⊕ *i*; *i* > *0*⟧ ⟹ *False*
⟨*proof*⟩

lemma *label-incr-0-rev* [*dest*]:
⟦*n* ⊕ *i* = (-*0*-); *i* > *0*⟧ ⟹ *False*
⟨*proof*⟩

4.2.2 CFG edges

type-synonym *w-edge* = (*w-node* × *state edge-kind* × *w-node*)

inductive *While-CFG* :: *cmd* ⇒ *w-node* ⇒ *state edge-kind* ⇒ *w-node* ⇒ *bool*
(- ⊢ - −-→ -)
where

WCFG-Entry-Exit:
prog ⊢ (*-Entry-*) −(λ*s*. *False*)$_{\surd}$→ (*-Exit-*)

| *WCFG-Entry*:
prog ⊢ (*-Entry-*) −(λ*s*. *True*)$_{\surd}$→ (*-0-*)

| *WCFG-Skip*:
Skip ⊢ (*-0-*) −⇑*id*→ (*-Exit-*)

| *WCFG-LAss*:
V:=*e* ⊢ (*-0-*) −⇑(λ*s*. *s*(*V*:=(*interpret e s*)))→ (*-1-*)

| *WCFG-LAssSkip*:
V:=*e* ⊢ (*-1-*) −⇑*id*→ (*-Exit-*)

| *WCFG-SeqFirst*:
⟦*c*$_1$ ⊢ *n* −*et*→ *n*′; *n*′ ≠ (*-Exit-*)⟧ ⟹ *c*$_1$;;*c*$_2$ ⊢ *n* −*et*→ *n*′

| *WCFG-SeqConnect*:
⟦*c*$_1$ ⊢ *n* −*et*→ (*-Exit-*); *n* ≠ (*-Entry-*)⟧ ⟹ *c*$_1$;;*c*$_2$ ⊢ *n* −*et*→ (*-0-*) ⊕ #:*c*$_1$

| *WCFG-SeqSecond*:
⟦*c*$_2$ ⊢ *n* −*et*→ *n*′; *n* ≠ (*-Entry-*)⟧ ⟹ *c*$_1$;;*c*$_2$ ⊢ *n* ⊕ #:*c*$_1$ −*et*→ *n*′ ⊕ #:*c*$_1$

| *WCFG-CondTrue*:
if (*b*) *c*$_1$ *else c*$_2$ ⊢ (*-0-*) −(λ*s*. *interpret b s* = *Some true*)$_{\surd}$→ (*-0-*) ⊕ *1*

| *WCFG-CondFalse*:
if (*b*) *c*$_1$ *else c*$_2$ ⊢ (*-0-*) −(λ*s*. *interpret b s* = *Some false*)$_{\surd}$→ (*-0-*) ⊕ (#:*c*$_1$ + *1*)

| *WCFG-CondThen*:
⟦*c*$_1$ ⊢ *n* −*et*→ *n*′; *n* ≠ (*-Entry-*)⟧ ⟹ *if* (*b*) *c*$_1$ *else c*$_2$ ⊢ *n* ⊕ *1* −*et*→ *n*′ ⊕ *1*

| *WCFG-CondElse*:
⟦*c*$_2$ ⊢ *n* −*et*→ *n*′; *n* ≠ (*-Entry-*)⟧
⟹ *if* (*b*) *c*$_1$ *else c*$_2$ ⊢ *n* ⊕ (#:*c*$_1$ + *1*) −*et*→ *n*′ ⊕ (#:*c*$_1$ + *1*)

| *WCFG-WhileTrue*:
while (*b*) *c*′ ⊢ (*-0-*) −(λ*s*. *interpret b s* = *Some true*)$_{\surd}$→ (*-0-*) ⊕ *2*

| *WCFG-WhileFalse*:

while (b) $c' \vdash$ *(-0-)* $-(\lambda s.\ \textit{interpret}\ b\ s = \textit{Some}\ \textit{false})_{\surd}\rightarrow$ *(-1-)*

| *WCFG-WhileFalseSkip*:
while (b) $c' \vdash$ *(-1-)* $-\Uparrow id\rightarrow$ *(-Exit-)*

| *WCFG-WhileBody*:
$[\![c' \vdash n\ -et\rightarrow\ n';\ n \neq$ *(-Entry-)*$;\ n' \neq$ *(-Exit-)*$]\!]$
$\Longrightarrow$ *while* (b) $c' \vdash n \oplus 2\ -et\rightarrow\ n' \oplus 2$

| *WCFG-WhileBodyExit*:
$[\![c' \vdash n\ -et\rightarrow$ *(-Exit-)*$;\ n \neq$ *(-Entry-)*$]\!] \Longrightarrow$ *while* (b) $c' \vdash n \oplus 2\ -et\rightarrow$ *(-0-)*

lemmas *WCFG-intros* $=$ *While-CFG.intros*[*split-format* $(complete)$]
lemmas *WCFG-elims* $=$ *While-CFG.cases*[*split-format* $(complete)$]
lemmas *WCFG-induct* $=$ *While-CFG.induct*[*split-format* $(complete)$]

4.2.3 Some lemmas about the CFG

lemma *WCFG-Exit-no-sourcenode* $[dest]$:
$prog \vdash$ *(-Exit-)* $-et\rightarrow\ n' \Longrightarrow$ *False*
⟨*proof*⟩

lemma *WCFG-Entry-no-targetnode* $[dest]$:
$prog \vdash n\ -et\rightarrow$ *(-Entry-)* $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *WCFG-sourcelabel-less-num-nodes*:
$prog \vdash$ *(-* l *-)* $-et\rightarrow\ n' \Longrightarrow l <$ #:*prog*
⟨*proof*⟩

lemma *WCFG-targetlabel-less-num-nodes*:
$prog \vdash n\ -et\rightarrow$ *(-* l *-)* $\Longrightarrow l <$ #:*prog*
⟨*proof*⟩

lemma *WCFG-EntryD*:
$prog \vdash$ *(-Entry-)* $-et\rightarrow\ n'$
$\Longrightarrow (n' =$ *(-Exit-)* $\wedge\ et = (\lambda s.\ \textit{False})_{\surd}) \vee (n' =$ *(-0-)* $\wedge\ et = (\lambda s.\ \textit{True})_{\surd})$
⟨*proof*⟩
lemma *WCFG-edge-det*:
$[\![prog \vdash n\ -et\rightarrow\ n';\ prog \vdash n\ -et'\rightarrow\ n']\!] \Longrightarrow et = et'$
⟨*proof*⟩
lemma *less-num-nodes-edge-Exit*:
obtains l *et* **where** $l <$ #:*prog* **and** $prog \vdash$ *(-* l *-)* $-et\rightarrow$ *(-Exit-)*
⟨*proof*⟩

lemma *less-num-nodes-edge*:
$l < \#{:}prog \Longrightarrow \exists\, n\ et.\ prog \vdash n\ -et\rightarrow\ (\text{-}\ l\ \text{-}) \vee prog \vdash (\text{-}\ l\ \text{-})\ -et\rightarrow\ n$
⟨*proof*⟩
lemma *WCFG-deterministic*:
$[\![prog \vdash n_1\ -et_1\rightarrow\ n_1'; prog \vdash n_2\ -et_2\rightarrow\ n_2'; n_1 = n_2; n_1' \neq n_2']\!]$
$\Longrightarrow \exists\, Q\ Q'.\ et_1 = (Q)_{\surd} \wedge et_2 = (Q')_{\surd} \wedge (\forall s.\ (Q\ s \longrightarrow \neg\ Q'\ s) \wedge (Q'\ s \longrightarrow \neg\ Q\ s))$
⟨*proof*⟩
end

4.3 Instantiate CFG locale with While CFG

theory *Interpretation* **imports**
WCFG
../Basic/CFGExit
begin

4.3.1 Instatiation of the *CFG* locale

abbreviation *sourcenode* :: *w-edge* ⇒ *w-node*
where *sourcenode e* ≡ *fst e*

abbreviation *targetnode* :: *w-edge* ⇒ *w-node*
where *targetnode e* ≡ *snd*(*snd e*)

abbreviation *kind* :: *w-edge* ⇒ *state edge-kind*
where *kind e* ≡ *fst*(*snd e*)

definition *valid-edge* :: *cmd* ⇒ *w-edge* ⇒ *bool*
where *valid-edge prog a* ≡ *prog* ⊢ *sourcenode a* −*kind a*→ *targetnode a*

definition *valid-node* ::*cmd* ⇒ *w-node* ⇒ *bool*
where *valid-node prog n* ≡
(∃ *a*. *valid-edge prog a* ∧ (*n* = *sourcenode a* ∨ *n* = *targetnode a*))

lemma *While-CFG-aux*:
CFG sourcenode targetnode (*valid-edge prog*) *Entry*
⟨*proof*⟩

interpretation *While-CFG*:
CFG sourcenode targetnode kind valid-edge prog Entry
for *prog*
⟨*proof*⟩

lemma *While-CFGExit-aux*:
CFGExit sourcenode targetnode kind (*valid-edge prog*) *Entry Exit*
⟨*proof*⟩

interpretation *While-CFGExit*:
CFGExit sourcenode targetnode kind valid-edge prog Entry Exit
for *prog*
⟨*proof*⟩

end

4.4 Labels

theory *Labels* **imports** *Com* **begin**

Labels describe a mapping from the inner node label to the matching command

inductive *labels* :: *cmd* ⇒ *nat* ⇒ *cmd* ⇒ *bool*
where

Labels-Base:
labels c 0 c

| *Labels-LAss*:
labels (*V*:=*e*) *1 Skip*

| *Labels-Seq1*:
labels c_1 *l c* ⟹ *labels* (c_1;;c_2) *l* (*c*;;c_2)

| *Labels-Seq2*:
labels c_2 *l c* ⟹ *labels* (c_1;;c_2) (*l* + #:c_1) *c*

| *Labels-CondTrue*:
labels c_1 *l c* ⟹ *labels* (*if* (*b*) c_1 *else* c_2) (*l* + *1*) *c*

| *Labels-CondFalse*:
labels c_2 *l c* ⟹ *labels* (*if* (*b*) c_1 *else* c_2) (*l* + #:c_1 + *1*) *c*

| *Labels-WhileBody*:
labels *c*′ *l c* ⟹ *labels* (*while*(*b*) *c*′) (*l* + *2*) (*c*;;*while*(*b*) *c*′)

| *Labels-WhileExit*:
labels (*while*(*b*) *c*′) *1 Skip*

lemma *label-less-num-inner-nodes*:
labels c l c′ ⟹ *l* < #:*c*
⟨*proof*⟩

declare *One-nat-def* [*simp del*]

lemma *less-num-inner-nodes-label*:
$l < \#{:}c \Longrightarrow \exists\, c'.\ labels\ c\ l\ c'$
⟨*proof*⟩

lemma *labels-det*:
$labels\ c\ l\ c' \Longrightarrow (\bigwedge c''.\ labels\ c\ l\ c'' \Longrightarrow c' = c'')$
⟨*proof*⟩

end

4.5 General well-formedness of While CFG

theory *WellFormed* **imports**
Interpretation
Labels
../Basic/CFGExit-wf
../StaticIntra/CDepInstantiations
begin

4.5.1 Definition of some functions

fun *lhs* :: *cmd* ⇒ *vname set*
where
$lhs\ Skip = \{\}$
$\mid lhs\ (V{:=}e) = \{V\}$
$\mid lhs\ (c_1;;c_2) = lhs\ c_1$
$\mid lhs\ (if\ (b)\ c_1\ else\ c_2) = \{\}$
$\mid lhs\ (while\ (b)\ c) = \{\}$

fun *rhs-aux* :: *expr* ⇒ *vname set*
where
$rhs\text{-}aux\ (Val\ v) = \{\}$
$\mid rhs\text{-}aux\ (Var\ V) = \{V\}$
$\mid rhs\text{-}aux\ (e1 \ll bop \gg e2) = (rhs\text{-}aux\ e1 \cup rhs\text{-}aux\ e2)$

fun *rhs* :: *cmd* ⇒ *vname set*
where
$rhs\ Skip = \{\}$
$\mid rhs\ (V{:=}e) = rhs\text{-}aux\ e$
$\mid rhs\ (c_1;;c_2) = rhs\ c_1$
$\mid rhs\ (if\ (b)\ c_1\ else\ c_2) = rhs\text{-}aux\ b$
$\mid rhs\ (while\ (b)\ c) = rhs\text{-}aux\ b$

lemma *rhs-interpret-eq*:
$[\![interpret\ b\ s = Some\ v'; \forall\ V \in rhs\text{-}aux\ b.\ s\ V = s'\ V]\!]$
$\Longrightarrow interpret\ b\ s' = Some\ v'$
⟨*proof*⟩

fun *Defs* :: *cmd* ⇒ *w-node* ⇒ *vname set*
where $Defs\ prog\ n = \{V.\ \exists\, l\ c.\ \ n = (\text{-}\ l\ \text{-}) \wedge labels\ prog\ l\ c \wedge V \in lhs\ c\}$

fun *Uses* :: *cmd* ⇒ *w-node* ⇒ *vname set*
where $Uses\ prog\ n = \{V.\ \exists\, l\ c.\ \ n = (\text{-}\ l\ \text{-}) \wedge labels\ prog\ l\ c \wedge V \in rhs\ c\}$

4.5.2 Lemmas about $prog \vdash n\ -et\rightarrow\ n'$ to show well-formed properties

lemma *WCFG-edge-no-Defs-equal*:
$[\![prog \vdash n\ -et\rightarrow\ n';\ V \notin Defs\ prog\ n]\!] \Longrightarrow (transfer\ et\ s)\ V = s\ V$
⟨*proof*⟩
lemma *WCFG-edge-transfer-uses-only-Uses*:
$[\![prog \vdash n\ -et\rightarrow\ n'; \forall\ V \in Uses\ prog\ n.\ s\ V = s'\ V]\!]$
$\Longrightarrow \forall\ V \in Defs\ prog\ n.\ (transfer\ et\ s)\ V = (transfer\ et\ s')\ V$
⟨*proof*⟩

lemma *WCFG-edge-Uses-pred-eq*:
$[\![prog \vdash n\ -et\rightarrow\ n'; \forall\ V \in Uses\ prog\ n.\ s\ V = s'\ V;\ pred\ et\ s]\!]$
$\Longrightarrow pred\ et\ s'$
⟨*proof*⟩
interpretation *While-CFG-wf*: *CFG-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id
for *prog*
⟨*proof*⟩

lemma *While-CFGExit-wf-aux*:*CFGExit-wf sourcenode targetnode kind*
(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id Exit*
⟨*proof*⟩

interpretation *While-CFGExit-wf*: *CFGExit-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id Exit
for *prog*
⟨*proof*⟩

end

4.6 Lemmas for the control dependences

theory *AdditionalLemmas* **imports** *WellFormed*
begin

4.6.1 Paths to (*-Exit-*) and from (*-Entry-*) exist

abbreviation *path* :: *cmd* ⇒ *w-node* ⇒ *w-edge list* ⇒ *w-node* ⇒ *bool*
(- ⊢ - −-→∗ -)
where *prog* ⊢ *n* −*as*→∗ *n′* ≡ *CFG.path sourcenode targetnode* (*valid-edge prog*)

n as n′

definition *label-incrs* :: *w-edge list* ⇒ *nat* ⇒ *w-edge list* (- ⊕*s* - *60*)
where *as* ⊕*s i* ≡ *map* (λ(*n*,*et*,*n′*). (*n* ⊕ *i*,*et*,*n′* ⊕ *i*)) *as*

lemma *path-SeqFirst*:
prog ⊢ *n* −*as*→∗ (- *l* -) ⟹ *prog*;;c_2 ⊢ *n* −*as*→∗ (- *l* -)
⟨*proof*⟩

lemma *path-SeqSecond*:
⟦*prog* ⊢ *n* −*as*→∗ *n′*; *n* ≠ (*-Entry-*); *as* ≠ []⟧
⟹ c_1;;*prog* ⊢ *n* ⊕ #:c_1 −*as* ⊕*s* #:c_1→∗ *n′* ⊕ #:c_1
⟨*proof*⟩

lemma *path-CondTrue*:
prog ⊢ (- *l* -) −*as*→∗ *n′*
⟹ *if* (*b*) *prog else* c_2 ⊢ (- *l* -) ⊕ *1* −*as* ⊕*s* *1*→∗ *n′* ⊕ *1*
⟨*proof*⟩

lemma *path-CondFalse*:
prog ⊢ (- *l* -) −*as*→∗ *n′*
⟹ *if* (*b*) c_1 *else prog* ⊢ (- *l* -) ⊕ (#:c_1 + *1*) −*as* ⊕*s* (#:c_1 + *1*)→∗ *n′* ⊕ (#:c_1 + *1*)
⟨*proof*⟩
lemma *path-While*:
prog ⊢ (- *l* -) −*as*→∗ (- *l′* -)
⟹ *while* (*b*) *prog* ⊢ (- *l* -) ⊕ *2* −*as* ⊕*s* *2*→∗ (- *l′* -) ⊕ *2*
⟨*proof*⟩

lemma *inner-node-Entry-Exit-path*:
l < #:*prog* ⟹ (∃ *as*. *prog* ⊢ (- *l* -) −*as*→∗ (*-Exit-*)) ∧
(∃ *as*. *prog* ⊢ (*-Entry-*) −*as*→∗ (- *l* -))
⟨*proof*⟩

lemma *valid-node-Exit-path*:
assumes *valid-node prog n* **shows** $\exists\, as.\ prog \vdash n\ -as\rightarrow*\ (\text{-}Exit\text{-})$
⟨*proof*⟩

lemma *valid-node-Entry-path*:
assumes *valid-node prog n* **shows** $\exists\, as.\ prog \vdash (\text{-}Entry\text{-})\ -as\rightarrow*\ n$
⟨*proof*⟩

4.6.2 Some finiteness considerations

lemma *finite-labels*:*finite* $\{l.\ \exists\, c.\ labels\ prog\ l\ c\}$
⟨*proof*⟩

lemma *finite-valid-nodes*:*finite* $\{n.\ valid\text{-}node\ prog\ n\}$
⟨*proof*⟩

lemma *finite-successors*:
finite $\{n'.\ \exists\, a'.\ valid\text{-}edge\ prog\ a' \wedge sourcenode\ a' = n \wedge$
$targetnode\ a' = n'\}$
⟨*proof*⟩

end

4.7 Interpretations of the various dynamic control dependences

theory *DynamicControlDependences* **imports** *AdditionalLemmas ../Dynamic/DynPDG*
begin

interpretation *WDynStandardControlDependence*:
DynStandardControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
for *prog*
⟨*proof*⟩

interpretation *WDynWeakControlDependence*:
DynWeakControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
for *prog*
⟨*proof*⟩

end

4.8 Semantics

theory *Semantics* **imports** *Labels Com* **begin**

4.8.1 Small Step Semantics

inductive *red* :: *cmd* ∗ *state* ⇒ *cmd* ∗ *state* ⇒ *bool*
and *red′* :: *cmd* ⇒ *state* ⇒ *cmd* ⇒ *state* ⇒ *bool*
(((1⟨-,/-⟩) →/ (1⟨-,/-⟩)) [0,0,0,0] 81)
where
⟨*c*,*s*⟩ → ⟨*c′*,*s′*⟩ == *red* (*c*,*s*) (*c′*,*s′*)
| *RedLAss*:
⟨*V*:=*e*,*s*⟩ → ⟨*Skip*,*s*(*V*:=(*interpret e s*))⟩

| *SeqRed*:
⟨*c*1,*s*⟩ → ⟨*c*1′,*s′*⟩ ⟹ ⟨*c*1;;*c*2,*s*⟩ → ⟨*c*1′;;*c*2,*s′*⟩

| *RedSeq*:
⟨*Skip*;;*c*2,*s*⟩ → ⟨*c*2,*s*⟩

| *RedCondTrue*:
interpret b s = *Some true* ⟹ ⟨*if* (*b*) *c*1 *else c*2,*s*⟩ → ⟨*c*1,*s*⟩

| *RedCondFalse*:
interpret b s = *Some false* ⟹ ⟨*if* (*b*) *c*1 *else c*2,*s*⟩ → ⟨*c*2,*s*⟩

| *RedWhileTrue*:
interpret b s = *Some true* ⟹ ⟨*while* (*b*) *c*,*s*⟩ → ⟨*c*;;*while* (*b*) *c*,*s*⟩

| *RedWhileFalse*:
interpret b s = *Some false* ⟹ ⟨*while* (*b*) *c*,*s*⟩ → ⟨*Skip*,*s*⟩

lemmas *red-induct* = *red.induct*[*split-format* (*complete*)]

abbreviation *reds* ::*cmd* ⇒ *state* ⇒ *cmd* ⇒ *state* ⇒ *bool*
(((1⟨-,/-⟩) →∗/ (1⟨-,/-⟩)) [0,0,0,0] 81) **where**
⟨*c*,*s*⟩ →∗ ⟨*c′*,*s′*⟩ == *red*∗∗ (*c*,*s*) (*c′*,*s′*)

4.8.2 Label Semantics

inductive *step* :: *cmd* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *bool*
((- ⊢ (1⟨-,/-,/-⟩) ⇝/ (1⟨-,/-,/-⟩)) [51,0,0,0,0,0,0] 81)
where

StepLAss:
V:=*e* ⊢ ⟨*V*:=*e*,*s*,0⟩ ⇝ ⟨*Skip*,*s*(*V*:=(*interpret e s*)),1⟩

| *StepSeq*:
⟦*labels* (*c*1;;*c*2) *l* (*Skip*;;*c*2); *labels* (*c*1;;*c*2) #:*c*1 *c*2; *l* < #:*c*1⟧
⟹ *c*1;;*c*2 ⊢ ⟨*Skip*;;*c*2,*s*,*l*⟩ ⇝ ⟨*c*2,*s*,#:*c*1⟩

| *StepSeqWhile*:
labels (*while* (*b*) *c′*) *l* (*Skip*;;*while* (*b*) *c′*)
⟹ *while* (*b*) *c′* ⊢ ⟨*Skip*;;*while* (*b*) *c′*,*s*,*l*⟩ ⇝ ⟨*while* (*b*) *c′*,*s*,*0*⟩

| *StepCondTrue*:
interpret b s = *Some true*
⟹ *if* (*b*) c_1 *else* c_2 ⊢ ⟨*if* (*b*) c_1 *else* c_2,*s*,*0*⟩ ⇝ ⟨c_1,*s*,*1*⟩

| *StepCondFalse*:
interpret b s = *Some false*
⟹ *if* (*b*) c_1 *else* c_2 ⊢ ⟨*if* (*b*) c_1 *else* c_2,*s*,*0*⟩ ⇝ ⟨c_2,*s*,#:c_1 + *1*⟩

| *StepWhileTrue*:
interpret b s = *Some true*
⟹ *while* (*b*) *c* ⊢ ⟨*while* (*b*) *c*,*s*,*0*⟩ ⇝ ⟨*c*;;*while* (*b*) *c*,*s*,*2*⟩

| *StepWhileFalse*:
interpret b s = *Some false* ⟹ *while* (*b*) *c* ⊢ ⟨*while* (*b*) *c*,*s*,*0*⟩ ⇝ ⟨*Skip*,*s*,*1*⟩

| *StepRecSeq1*:
prog ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩
⟹ *prog*;;c_2 ⊢ ⟨*c*;;c_2,*s*,*l*⟩ ⇝ ⟨*c′*;;c_2,*s′*,*l′*⟩

| *StepRecSeq2*:
prog ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩
⟹ c_1;;*prog* ⊢ ⟨*c*,*s*,*l* + #:c_1⟩ ⇝ ⟨*c′*,*s′*,*l′* + #:c_1⟩

| *StepRecCond1*:
prog ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩
⟹ *if* (*b*) *prog* *else* c_2 ⊢ ⟨*c*,*s*,*l* + *1*⟩ ⇝ ⟨*c′*,*s′*,*l′* + *1*⟩

| *StepRecCond2*:
prog ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩
⟹ *if* (*b*) c_1 *else* *prog* ⊢ ⟨*c*,*s*,*l* + #:c_1 + *1*⟩ ⇝ ⟨*c′*,*s′*,*l′* + #:c_1 + *1*⟩

| *StepRecWhile*:
cx ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩
⟹ *while* (*b*) *cx* ⊢ ⟨*c*;;*while* (*b*) *cx*,*s*,*l* + *2*⟩ ⇝ ⟨*c′*;;*while* (*b*) *cx*,*s′*,*l′* + *2*⟩

lemma *step-label-less*:
prog ⊢ ⟨*c*,*s*,*l*⟩ ⇝ ⟨*c′*,*s′*,*l′*⟩ ⟹ *l* < #:*prog* ∧ *l′* < #:*prog*
⟨*proof*⟩

abbreviation *steps* :: *cmd* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *cmd* ⇒ *state* ⇒ *nat* ⇒ *bool*
((- ⊢ (*1*⟨-,/-,/-⟩) ⇝*/ (*1*⟨-,/-,/-⟩)) [*51*,*0*,*0*,*0*,*0*,*0*,*0*] *81*) **where**

$prog \vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle$ ==
  $(\lambda(c,s,l)\ (c',s',l').\ prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle)^{**}\ (c,s,l)\ (c',s',l')$

4.8.3 Proof of bisimulation of $\langle c,s\rangle \to \langle c',s'\rangle$ and $prog \vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle$ via *labels*

From $prog \vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle$ **to** $\langle c,s\rangle \to \langle c',s'\rangle$

lemma *step-red*:
  $prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle \Longrightarrow \langle c,s\rangle \to \langle c',s'\rangle$
⟨*proof*⟩

lemma *steps-reds*:
  $prog \vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle \Longrightarrow \langle c,s\rangle \to* \langle c',s'\rangle$
⟨*proof*⟩

From $\langle c,s\rangle \to \langle c',s'\rangle$ **and** *labels* **to** $prog \vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle$

lemma *red-step*:
  $[\![labels\ prog\ l\ c;\ \langle c,s\rangle \to \langle c',s'\rangle]\!]$
  $\Longrightarrow \exists\, l'.\ prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle \wedge labels\ prog\ l'\ c'$
⟨*proof*⟩

lemma *reds-steps*:
  $[\![\langle c,s\rangle \to* \langle c',s'\rangle;\ labels\ prog\ l\ c]\!]$
  $\Longrightarrow \exists\, l'.\ prog \vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle \wedge labels\ prog\ l'\ c'$
⟨*proof*⟩

The bisimulation theorem

theorem *reds-steps-bisimulation*:
  $labels\ prog\ l\ c \Longrightarrow (\langle c,s\rangle \to* \langle c',s'\rangle) =$
    $(\exists\, l'.\ prog \vdash \langle c,s,l\rangle \rightsquigarrow* \langle c',s',l'\rangle \wedge labels\ prog\ l'\ c')$
  ⟨*proof*⟩

end

4.9 Equivalence

theory *WEquivalence* **imports** *Semantics WCFG* **begin**

4.9.1 From $prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$ to $c \vdash (\text{-}\ l\ \text{-}) -et\to (\text{-}\ l'\ \text{-})$ with *transfers* and *preds*

lemma *Skip-WCFG-edge-Exit*:

$[\![labels\ prog\ l\ Skip]\!] \Longrightarrow prog \vdash (\text{-}\ l\ \text{-}) -\Uparrow id\rightarrow (\text{-}Exit\text{-})$
⟨*proof*⟩

lemma *step-WCFG-edge*:
assumes $prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$
obtains *et* **where** $prog \vdash (\text{-}\ l\ \text{-}) -et\rightarrow (\text{-}\ l'\ \text{-})$ **and** $transfer\ et\ s = s'$
and *pred et s*
⟨*proof*⟩

4.9.2 From $c \vdash (\text{-}\ l\ \text{-}) -et\rightarrow (\text{-}\ l'\ \text{-})$ with *transfers* and *preds* to $prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$

lemma *WCFG-edge-Exit-Skip*:
$[\![prog \vdash n -et\rightarrow (\text{-}Exit\text{-});\ n \neq (\text{-}Entry\text{-})]\!]$
$\Longrightarrow \exists l.\ n = (\text{-}\ l\ \text{-}) \wedge labels\ prog\ l\ Skip \wedge et = \Uparrow id$
⟨*proof*⟩

lemma *WCFG-edge-step*:
$[\![prog \vdash (\text{-}\ l\ \text{-}) -et\rightarrow (\text{-}\ l'\ \text{-});\ transfer\ et\ s = s';\ pred\ et\ s]\!]$
$\Longrightarrow \exists c\ c'.\ prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle \wedge labels\ prog\ l\ c \wedge labels\ prog\ l'\ c'$
⟨*proof*⟩

end

4.10 Semantic well-formedness of While CFG

theory *SemanticsWellFormed*
imports *WellFormed WEquivalence ../Basic/SemanticsCFG*
begin

4.10.1 Instatiation of the *CFG-semantics-wf* locale

fun *labels-nodes* :: *cmd* ⇒ *w-node* ⇒ *cmd* ⇒ *bool* **where**
labels-nodes prog (- *l* -) *c* = *labels prog l c*
| *labels-nodes prog* (*-Entry-*) *c* = *False*
| *labels-nodes prog* (*-Exit-*) *c* = *False*

interpretation *While-semantics-CFG-wf*: *CFG-semantics-wf*
sourcenode targetnode kind valid-edge prog Entry reds labels-nodes prog
for *prog*
⟨*proof*⟩

end

4.11 Interpretations of the various static control dependences

theory *StaticControlDependences* **imports**
AdditionalLemmas
SemanticsWellFormed
begin

lemma *WhilePostdomination-aux*:
Postdomination sourcenode targetnode kind (*valid-edge prog*) *Entry Exit*
⟨*proof*⟩

interpretation *WhilePostdomination*:
Postdomination sourcenode targetnode kind valid-edge prog Entry Exit
⟨*proof*⟩

lemma *WhileStrongPostdomination-aux*:
StrongPostdomination sourcenode targetnode kind (*valid-edge prog*) *Entry Exit*
⟨*proof*⟩

interpretation *WhileStrongPostdomination*:
StrongPostdomination sourcenode targetnode kind valid-edge prog Entry Exit
⟨*proof*⟩

4.11.1 Standard Control Dependence

lemma *WStandardControlDependence-aux*:
StandardControlDependencePDG sourcenode targetnode kind (*valid-edge prog*)
Entry (*Defs prog*) (*Uses prog*) *id Exit*
⟨*proof*⟩

interpretation *WStandardControlDependence*:
StandardControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
⟨*proof*⟩

lemma *Fundamental-property-scd-aux*: *BackwardSlice-wf sourcenode targetnode kind*

(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id*
(*WStandardControlDependence.PDG-BS-s prog*) *reds* (*labels-nodes prog*)
⟨*proof*⟩

interpretation *Fundamental-property-scd*: *BackwardSlice-wf sourcenode targetnode kind*

valid-edge prog Entry Defs prog Uses prog id
WStandardControlDependence.PDG-BS-s prog reds labels-nodes prog
⟨*proof*⟩

4.11.2 Weak Control Dependence

lemma *WWeakControlDependence-aux*:
WeakControlDependencePDG sourcenode targetnode kind (*valid-edge prog*)
Entry (*Defs prog*) (*Uses prog*) *id Exit*
⟨*proof*⟩

interpretation *WWeakControlDependence*:
WeakControlDependencePDG sourcenode targetnode kind valid-edge prog
Entry Defs prog Uses prog id Exit
⟨*proof*⟩

lemma *Fundamental-property-wcd-aux*: *BackwardSlice-wf sourcenode targetnode kind*

(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id*
(*WWeakControlDependence.PDG-BS-w prog*) *reds* (*labels-nodes prog*)
⟨*proof*⟩

interpretation *Fundamental-property-wcd*: *BackwardSlice-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id
WWeakControlDependence.PDG-BS-w prog reds labels-nodes prog
⟨*proof*⟩

4.11.3 Weak Order Dependence

lemma *Fundamental-property-wod-aux*: *BackwardSlice-wf sourcenode targetnode kind*

(*valid-edge prog*) *Entry* (*Defs prog*) (*Uses prog*) *id*
(*While-CFG-wf.wod-backward-slice prog*) *reds* (*labels-nodes prog*)
⟨*proof*⟩

interpretation *Fundamental-property-wod*: *BackwardSlice-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id
While-CFG-wf.wod-backward-slice prog reds labels-nodes prog
⟨*proof*⟩

end

4.12 Slicing guarantees IFC Noninterference

theory *NonInterferenceIntra* **imports**
../Slicing/StaticIntra/Slice

../Slicing/Basic/CFGExit-wf
begin

4.12.1 Assumptions of this Approach

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

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

locale *NonInterferenceIntraGraph* =
BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *Exit* :: *'node* (*'('-Exit'-')*) +
fixes *H* :: *'var set*
fixes *L* :: *'var set*
fixes *High* :: *'node* (*'('-High'-')*)
fixes *Low* :: *'node* (*'('-Low'-')*)
assumes *Entry-edge-Exit-or-High*:
⟦*valid-edge a*; *sourcenode a* = (*-Entry-*)⟧
⟹ *targetnode a* = (*-Exit-*) ∨ *targetnode a* = (*-High-*)
and *High-target-Entry-edge*:
∃ *a*. *valid-edge a* ∧ *sourcenode a* = (*-Entry-*) ∧ *targetnode a* = (*-High-*) ∧
kind a = $(\lambda s.\ True)_{\surd}$
and *Entry-predecessor-of-High*:
⟦*valid-edge a*; *targetnode a* = (*-High-*)⟧ ⟹ *sourcenode a* = (*-Entry-*)
and *Exit-edge-Entry-or-Low*: ⟦*valid-edge a*; *targetnode a* = (*-Exit-*)⟧
⟹ *sourcenode a* = (*-Entry-*) ∨ *sourcenode a* = (*-Low-*)
and *Low-source-Exit-edge*:
∃ *a*. *valid-edge a* ∧ *sourcenode a* = (*-Low-*) ∧ *targetnode a* = (*-Exit-*) ∧
kind a = $(\lambda s.\ True)_{\surd}$
and *Exit-successor-of-Low*:

⟦*valid-edge a*; *sourcenode a* = (*-Low-*)⟧ ⟹ *targetnode a* = (*-Exit-*)
and *DefHigh*: *Def* (*-High-*) = *H*
and *UseHigh*: *Use* (*-High-*) = *H*
and *UseLow*: *Use* (*-Low-*) = *L*
and *HighLowDistinct*: *H* ∩ *L* = {}
and *HighLowUNIV*: *H* ∪ *L* = *UNIV*

begin

lemma *Low-neq-Exit*: **assumes** *L* ≠ {} **shows** (*-Low-*) ≠ (*-Exit-*)
⟨*proof*⟩

lemma *Entry-path-High-path*:
assumes (*-Entry-*) −*as*→∗ *n* **and** *inner-node n*
obtains *a*′ *as*′ **where** *as* = *a*′#*as*′ **and** (*-High-*) −*as*′→∗ *n*
and *kind a*′ = (λ*s*. *True*)$_{\surd}$
⟨*proof*⟩

lemma *Exit-path-Low-path*:
assumes *n* −*as*→∗ (*-Exit-*) **and** *inner-node n*
obtains *a*′ *as*′ **where** *as* = *as*′@[*a*′] **and** *n* −*as*′→∗ (*-Low-*)
and *kind a*′ = (λ*s*. *True*)$_{\surd}$
⟨*proof*⟩

lemma *not-Low-High*: *V* ∉ *L* ⟹ *V* ∈ *H*
⟨*proof*⟩

lemma *not-High-Low*: *V* ∉ *H* ⟹ *V* ∈ *L*
⟨*proof*⟩

4.12.2 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 ≈$_L$:

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

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

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

lemma *lowEquivalence-relevant-nodes-Entry*:
assumes $s \approx_L s'$ **and** (*-High-*) $\notin$ *backward-slice S*
shows $\forall V \in rv\ S$ (*-Entry-*). *state-val s V* = *state-val s' V*
⟨*proof*⟩

lemma *rv-Low-Use-Low*:
assumes (*-Low-*) $\in S$
shows ⟦n $-as\rightarrow*$ (*-Low-*); n $-as'\rightarrow*$ (*-Low-*);
$\forall V \in rv\ S\ n$. *state-val s V* = *state-val s' V*;
preds (*slice-kinds S as*) *s*; *preds* (*slice-kinds S as'*) *s'*⟧
$\Longrightarrow \forall V \in$ *Use* (*-Low-*). *state-val* (*transfers* (*slice-kinds S as*) *s*) *V* =
state-val (*transfers* (*slice-kinds S as'*) *s'*) *V*
⟨*proof*⟩

4.12.3 The Correctness Proofs

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

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

lemma *nonInterference-path-to-Low*:
assumes $s \approx_L s'$ **and** (*-High-*) $\notin$ *backward-slice S* **and** (*-Low-*) $\in S$
and (*-Entry-*) $-as\rightarrow*$ (*-Low-*) **and** *preds* (*kinds as*) *s*
and (*-Entry-*) $-as'\rightarrow*$ (*-Low-*) **and** *preds* (*kinds as'*) *s'*
shows *transfers* (*kinds as*) *s* $\approx_L$ *transfers* (*kinds as'*) *s'*
⟨*proof*⟩

theorem *nonInterference-path*:
assumes $s \approx_L s'$ **and** (*-High-*) $\notin$ *backward-slice S* **and** (*-Low-*) $\in S$
and (*-Entry-*) $-as\rightarrow*$ (*-Exit-*) **and** *preds* (*kinds as*) *s*
and (*-Entry-*) $-as'\rightarrow*$ (*-Exit-*) **and** *preds* (*kinds as'*) *s'*
shows *transfers* (*kinds as*) *s* $\approx_L$ *transfers* (*kinds as'*) *s'*
⟨*proof*⟩

end

The second theorem assumes that we have a operational semantics, whose evaluations are written $\langle c,s\rangle \Rightarrow \langle c',s'\rangle$ and which conforms to the CFG. The correctness theorem then states that if no high variable influenced a low

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

locale *NonInterferenceIntra* =
NonInterferenceIntraGraph sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice Exit H L High Low +
BackwardSlice-wf sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice sem identifies
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (*'('-Entry'-')*) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *sem* :: *'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *bool*
(((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩)) [*0*,*0*,*0*,*0*] *81*)
and *identifies* :: *'node* ⇒ *'com* ⇒ *bool* (- ≙ - [*51*, *0*] *80*)
and *Exit* :: *'node* (*'('-Exit'-')*)
and *H* :: *'var set* **and** *L* :: *'var set*
and *High* :: *'node* (*'('-High'-')*) **and** *Low* :: *'node* (*'('-Low'-')*) +
fixes *final* :: *'com* ⇒ *bool*
assumes *final-edge-Low*: ⟦*final c*; *n* ≙ *c*⟧
⟹ ∃ *a*. *valid-edge a* ∧ *sourcenode a* = *n* ∧ *targetnode a* = (*-Low-*) ∧ *kind a* = ⇑*id*
begin

The following theorem needs the explicit edge from (*-High-*) to *n*. An approach using a *init* predicate for initial statements, being reachable from (*-High-*) via a $(\lambda s.\ True)_{\surd}$ edge, does not work as the same statement could be identified by several nodes, some initial, some not. E.g., in the program `while (True) Skip;;Skip` two nodes identify this inital statement: the initial node and the node within the loop (because of loop unrolling).

theorem *nonInterference*:
assumes $s_1 \approx_L s_2$ **and** (*-High-*) ∉ *backward-slice S* **and** (*-Low-*) ∈ *S*
and *valid-edge a* **and** *sourcenode a* = (*-High-*) **and** *targetnode a* = *n*
and *kind a* = $(\lambda s.\ True)_{\surd}$ **and** *n* ≙ *c* **and** *final c'*
and ⟨*c*,s_1⟩ ⇒ ⟨*c'*,s_1'⟩ **and** ⟨*c*,s_2⟩ ⇒ ⟨*c'*,s_2'⟩
shows $s_1' \approx_L s_2'$
⟨*proof*⟩

end

end

4.13 Framework Graph Lifting for Noninterference

theory *LiftingIntra*

imports *NonInterferenceIntra ../Slicing/StaticIntra/CDepInstantiations*
begin

In this section, we show how a valid CFG from the slicing framework in [**?**] 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.

4.13.1 Liftings

The datatypes

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

type-synonym *('edge,'node,'state) LDCFG-edge =*
'node LDCFG-node × *('state edge-kind)* × *'node LDCFG-node*

Lifting *valid-edge*

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

where *lve-edge*:
⟦*valid-edge a*; *src a* ≠ *E* ∨ *trg a* ≠ *X*;
e = (*Node* (*src a*),*knd a*,*Node* (*trg a*))⟧
⟹ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Entry-edge*:
e = (*NewEntry*,$(\lambda s.\ True)_{\surd}$,*Node E*)
⟹ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Exit-edge*:
e = (*Node X*,$(\lambda s.\ True)_{\surd}$,*NewExit*)
⟹ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Entry-Exit-edge*:
e = (*NewEntry*,$(\lambda s.\ False)_{\surd}$,*NewExit*)
⟹ *lift-valid-edge valid-edge src trg knd E X e*

lemma [*simp*]:¬ *lift-valid-edge valid-edge src trg knd E X* (*Node E*,*et*,*Node X*)
⟨*proof*⟩

Lifting *Def* and *Use* sets

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

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

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

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

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

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

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

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

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

4.13.2 The lifting lemmas

Lifting the basic locales

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

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

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

lemma *lift-CFG*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFG src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
⟨*proof*⟩

lemma *lift-CFG-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFG-wf src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val*
⟨*proof*⟩

lemma *lift-CFGExit*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFGExit src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*)
NewEntry NewExit
⟨*proof*⟩

lemma *lift-CFGExit-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFGExit-wf src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
⟨*proof*⟩

Lifting *wod-backward-slice*

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

and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val*
(*CFG-wf.wod-backward-slice src trg lve lDef lUse*)
NewExit H L (*Node Entry*) (*Node Exit*)
⟨*proof*⟩

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

lemma *lift-Postdomination*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *Postdomination src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry NewExit*
⟨*proof*⟩

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

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

NewExit H L (*Node Entry*) (*Node Exit*)
⟨*proof*⟩

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

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

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

lemma *lift-PDG-weak-backward-slice*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit* **and** *Def* **and** *Use* **and** *H* **and** *L*
defines *lve*:*lve* ≡ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lDef*:*lDef* ≡ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* ≡ *lift-Use Use Entry Exit H L*
assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*StrongPostdomination.weak-control-dependence sourcenode targetnode valid-edge Exit*)
and *spd*:*StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val*

(*PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*))
NewExit H L (*Node Entry*) (*Node Exit*)
⟨*proof*⟩

end

4.14 Information Flow for While

theory *NonInterferenceWhile* **imports**
SemanticsWellFormed
StaticControlDependences
../../InformationFlowSlicing/LiftingIntra
begin

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

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

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

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

lemma *lift-valid-node*:
assumes *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
shows *CFG.valid-node src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind* (*-Entry-*) (*-Exit-*))
(*Node n*)
⟨*proof*⟩

lemma *lifted-CFG-fund-prop*:
assumes *labels-LDCFG-nodes prog n c* **and** $\langle c,s\rangle \rightarrow* \langle c',s'\rangle$
shows $\exists\, n'\ as.$ *CFG.path src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind* (*-Entry-*) (*-Exit-*))
n as n' $\wedge$ *transfers* (*CFG.kinds knd as*) *s* = *s'* $\wedge$
preds (*CFG.kinds knd as*) *s* $\wedge$ *labels-LDCFG-nodes prog n' c'*
⟨*proof*⟩

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

lemma *final-edge*:
labels-nodes prog n Skip ⟹ *prog* ⊢ *n* −⇑*id*→ (*-Exit-*)
⟨*proof*⟩

4.14.2 Semantic Non-Interference for Weak Order Dependence

lemmas *WODNonInterferenceGraph* =
lift-wod-backward-slice[*OF While-CFGExit-wf-aux HighLowDistinct HighLowUNIV*]

lemma *WODNonInterference*:
NonInterferenceIntra src trg knd
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
NewEntry (*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *id*
(*CFG-wf.wod-backward-slice src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*))
reds (*labels-LDCFG-nodes prog*)
NewExit H L (*LDCFG-node.Node* (*-Entry-*)) (*LDCFG-node.Node* (*-Exit-*)) *final*
⟨*proof*⟩

4.14.3 Semantic Non-Interference for Standard Control Dependence

lemma *inner-node-exists*:∃ *n. CFGExit.inner-node sourcenode targetnode*
(*valid-edge prog*) (*-Entry-*) (*-Exit-*) *n*
⟨*proof*⟩

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

lemma *SCDNonInterference*:
NonInterferenceIntra src trg knd
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
NewEntry (*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *id*
(*PDG.PDG-BS src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*Postdomination.standard-control-dependence src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*)) *NewExit*))
reds (*labels-LDCFG-nodes prog*)
NewExit H L (*LDCFG-node.Node* (*-Entry-*)) (*LDCFG-node.Node* (*-Exit-*)) *final*
⟨*proof*⟩

4.14.4 Semantic Non-Interference for Weak Control Dependence

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

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

end

end

Chapter 5

A Control Flow Graph for Jinja Byte Code

This work was done by Denis Lohner (denis.lohner@kit.edu).

5.1 Formalizing the CFG

theory *JVMCFG* **imports** *../Basic/BasicDefs ../../Jinja/BV/BVExample* **begin**

declare *lesub-list-impl-same-size* [*simp del*]
declare *listE-length* [*simp del*]

5.1.1 Type definitions

Wellformed Programs

definition *wf-jvmprog* = {(*P*, *Phi*). *wf-jvm-prog*$_{Phi}$ *P*}

typedef *wf-jvmprog* = *wf-jvmprog*
⟨*proof*⟩

hide-const *Phi E*

abbreviation *rep-jvmprog-jvm-prog* :: *wf-jvmprog* ⇒ *jvm-prog*
(-$_{wf}$)
  where P_{wf} ≡ *fst*(*Rep-wf-jvmprog*(*P*))

abbreviation *rep-jvmprog-phi* :: *wf-jvmprog* ⇒ ty_P
(-$_\Phi$)
  where P_Φ ≡ *snd*(*Rep-wf-jvmprog*(*P*))

lemma *wf-jvmprog-is-wf*: *wf-jvm-prog*$_{P_\Phi}$ (P_{wf})
⟨*proof*⟩

Basic Types

We consider a program to be a well-formed Jinja program, together with a given base class and a main method

type-synonym *jvmprog* = *wf-jvmprog* × *cname* × *mname*
type-synonym *callstack* = (*cname* × *mname* × *pc*) *list*

The state is modeled as heap × stack-variables × local-variables

stack and local variables are modeled as pairs of natural numbers. The first number gives the position in the call stack (i.e. the method in which the variable is used), the second the position in the method's stack or array of local variables resp.

The stack variables are numbered from bottom up (which is the reverse order of the array for the stack in Jinja's state representation), whereas local variables are identified by their position in the array of local variables of Jinja's state representation.

type-synonym *state* = *heap* × ((*nat* × *nat*) ⇒ *val*) × ((*nat* × *nat*) ⇒ *val*)

abbreviation *heap-of* :: *state* ⇒ *heap*
where
heap-of s ≡ *fst*(*s*)

abbreviation *stk-of* :: *state* ⇒ ((*nat* × *nat*) ⇒ *val*)
where
stk-of s ≡ *fst*(*snd*(*s*))

abbreviation *loc-of* :: *state* ⇒ ((*nat* × *nat*) ⇒ *val*)
where
loc-of s ≡ *snd*(*snd*(*s*))

5.1.2 Basic Definitions

State update (instruction execution)

This function models instruction execution for our state representation.

Additional parameters are the call depth of the current program point, the stack length of the current program point, the length of the stack in the underlying call frame (needed for RETURN), and (for INVOKE) the length of the array of local variables of the invoked method.

Exception handling is not covered by this function.

fun *exec-instr* :: *instr* ⇒ *wf-jvmprog* ⇒ *state* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *state*
where
exec-instr-Load:
exec-instr (*Load n*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*

```
   in (h, stk((calldepth,stk-length):=loc(calldepth,n)), loc))

| exec-instr-Store:
  exec-instr (Store n) P s calldepth stk-length rs ill =
  (let (h,stk,loc) = s
   in (h, stk, loc((calldepth,n):=stk(calldepth,stk-length − 1))))

| exec-instr-Push:
  exec-instr (Push v) P s calldepth stk-length rs ill =
  (let (h,stk,loc) = s
   in (h, stk((calldepth,stk-length):=v), loc))

| exec-instr-New:
  exec-instr (New C) P s calldepth stk-length rs ill =
  (let (h,stk,loc) = s;
     a = the(new-Addr h)
   in (h(a ↦ (blank (P_wf) C)), stk((calldepth,stk-length):=Addr a), loc))

| exec-instr-Getfield:
  exec-instr (Getfield F C) P s calldepth stk-length rs ill =
  (let (h,stk,loc) = s;
     a = the-Addr (stk (calldepth,stk-length − 1));
     (D,fs) = the(h a)
   in (h, stk((calldepth,stk-length − 1) := the(fs(F,C))), loc))

| exec-instr-Putfield:
  exec-instr (Putfield F C) P s calldepth stk-length rs ill =
  (let (h,stk,loc) = s;
     v = stk (calldepth,stk-length − 1);
     a = the-Addr (stk (calldepth,stk-length − 2));
     (D,fs) = the(h a)
   in (h(a ↦ (D,fs((F,C) ↦ v))), stk, loc))

| exec-instr-Checkcast:
  exec-instr (Checkcast C) P s calldepth stk-length rs ill = s

| exec-instr-Pop:
  exec-instr (Pop) P s calldepth stk-length rs ill = s

| exec-instr-IAdd:
  exec-instr (IAdd) P s calldepth stk-length rs ill =
  (let (h,stk,loc) = s;
     i_1 = the-Intg (stk (calldepth, stk-length − 1));
     i_2 = the-Intg (stk (calldepth, stk-length − 2))
   in (h, stk((calldepth, stk-length − 2) := Intg (i_1 + i_2)), loc))

| exec-instr-IfFalse:
  exec-instr (IfFalse b) P s calldepth stk-length rs ill = s
```

| *exec-instr-CmpEq*:
exec-instr (*CmpEq*) *P s calldepth stk-length rs ill* =
(*let* (*h*,*stk*,*loc*) = *s*;
v_1 = *stk* (*calldepth*, *stk-length* − *1*);
v_2 = *stk* (*calldepth*, *stk-length* − *2*)
in (*h*, *stk*((*calldepth*, *stk-length* − *2*) := *Bool* (v_1 = v_2)), *loc*))

| *exec-instr-Goto*:
exec-instr (*Goto i*) *P s calldepth stk-length rs ill* = *s*

| *exec-instr-Throw*:
exec-instr (*Throw*) *P s calldepth stk-length rs ill* = *s*

| *exec-instr-Invoke*:
exec-instr (*Invoke M n*) *P s calldepth stk-length rs invoke-loc-length* =
(*let* (*h*,*stk*,*loc*) = *s*;
loc′ = (λ(*a*,*b*). *if* (*a* ≠ *Suc calldepth* ∨ *b* ≥ *invoke-loc-length*) *then loc*(*a*,*b*) *else*
(*if* (*b* ≤ *n*) *then stk*(*calldepth*, *stk-length* − (*Suc n* − *b*)) *else*
arbitrary))
in (*h*,*stk*,*loc′*))

| *exec-instr-Return*:
exec-instr (*Return*) *P s calldepth stk-length ret-stk-length ill* =
(*if* (*calldepth* = *0*)
then s
else
(*let* (*h*,*stk*,*loc*) = *s*;
v = *stk*(*calldepth*, *stk-length* − *1*)
in (*h*,*stk*((*calldepth* − *1*, *ret-stk-length* − *1*) := *v*),*loc*))
)

length of stack and local variables

The following terms extract the stack length at a given program point from the well-typing of the given program

abbreviation *stkLength* :: *wf-jvmprog* ⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *nat*
where
stkLength P C M pc ≡ *length* (*fst*(*the*(((*P*$_\Phi$) *C M*)!*pc*)))

abbreviation *locLength* :: *wf-jvmprog* ⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *nat*
where
locLength P C M pc ≡ *length* (*snd*(*the*(((*P*$_\Phi$) *C M*)!*pc*)))

Conversion functions

This function takes a natural number n and a function f with domain *nat* and creates the array [f 0, f 1, f 2, ..., f (n - 1)].

This is used for extracting the array of local variables

abbreviation *locs* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ *'a list*
where *locs n loc* ≡ *map loc* [*0*..<*n*]

This function takes a natural number n and a function f with domain *nat* and creates the array [f (n - 1), ..., f 1, f 0].
This is used for extracting the stack as a list

abbreviation *stks* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ *'a list*
where *stks n stk* ≡ *map stk* (*rev* [*0*..<*n*])

This function creates a list of the arrays for local variables from the given state corresponding to the given callstack

fun *locss* :: *wf-jvmprog* ⇒ *callstack* ⇒ ((*nat* × *nat*) ⇒ *'a*) ⇒ *'a list list*
where
  locss P [] *loc* = []
| *locss P* ((*C*,*M*,*pc*)#*cs*) *loc* =
    (*locs* (*locLength P C M pc*) (λ*a*. *loc* (*length cs*, *a*)))#(*locss P cs loc*)

This function creates a list of the (methods') stacks from the given state corresponding to the given callstack

fun *stkss* :: *wf-jvmprog* ⇒ *callstack* ⇒ ((*nat* × *nat*) ⇒ *'a*) ⇒ *'a list list*
where
  stkss P [] *stk* = []
| *stkss P* ((*C*,*M*,*pc*)#*cs*) *stk* =
  (*stks* (*stkLength P C M pc*) (λ*a*. *stk* (*length cs*, *a*)))#(*stkss P cs stk*)

Given a callstack and a state, this abbreviation converts the state to Jinja's state representation

abbreviation *state-to-jvm-state* :: *wf-jvmprog* ⇒ *callstack* ⇒ *state* ⇒ *jvm-state*
where *state-to-jvm-state P cs s* ≡
  (*None*, *heap-of s*, *zip* (*stkss P cs* (*stk-of s*)) (*zip* (*locss P cs* (*loc-of s*)) *cs*))

This function extracts the call stack from a given frame stack (as it is given by Jinja's state representation)

definition *framestack-to-callstack* :: *frame list* ⇒ *callstack*
where *framestack-to-callstack frs* ≡ *map snd* (*map snd frs*)

State Conformance

Now we lift byte code verifier conformance to our state representation

definition *bv-conform* :: *wf-jvmprog* ⇒ *callstack* ⇒ *state* ⇒ *bool*
  (-,- ⊢$_{BV}$ - √)
where *P*,*cs* ⊢$_{BV}$ *s* √ ≡ *correct-state* (*P*$_{wf}$) (*P*$_{\Phi}$) (*state-to-jvm-state P cs s*)

Statically determine catch-block

This function is equivalent to Jinja's *find-handler* function

fun *find-handler-for* :: *wf-jvmprog* ⇒ *cname* ⇒ *callstack* ⇒ *callstack*
where
  find-handler-for P C [] = []
| *find-handler-for P C* (*c*#*cs*) = (*let* (*C'*,*M'*,*pc'*) = *c in*
    (*case match-ex-table* (*P*$_{wf}$) *C pc'* (*ex-table-of* (*P*$_{wf}$) *C' M'*) *of*
        None ⇒ *find-handler-for P C cs*
      | *Some pc-d* ⇒ (*C'*, *M'*, *fst pc-d*)#*cs*))

5.1.3 Simplification lemmas

lemma *find-handler-decr* [*simp*]: *find-handler-for P Exc cs* ≠ *c*#*cs*
⟨*proof*⟩

lemma *stkss-length* [*simp*]: *length* (*stkss P cs stk*) = *length cs*
⟨*proof*⟩

lemma *locss-length* [*simp*]: *length* (*locss P cs loc*) = *length cs*
⟨*proof*⟩

lemma *nth-stkss*:
  ⟦ *a* < *length cs*; *b* < *length* (*stkss P cs stk* ! (*length cs* − *Suc a*)) ⟧
  ⟹ *stkss P cs stk* ! (*length cs* − *Suc a*) !
    (*length* (*stkss P cs stk* ! (*length cs* − *Suc a*)) − *Suc b*) = *stk* (*a*,*b*)
⟨*proof*⟩

lemma *nth-locss*:
  ⟦ *a* < *length cs*; *b* < *length* (*locss P cs loc* ! (*length cs* − *Suc a*)) ⟧
  ⟹ *locss P cs loc* ! (*length cs* − *Suc a*) ! *b* = *loc* (*a*,*b*)
⟨*proof*⟩

lemma *hd-stks* [*simp*]: *n* ≠ *0* ⟹ *hd* (*stks n stk*) = *stk*(*n* − *1*)
⟨*proof*⟩

lemma *hd-tl-stks*: *n* > *1* ⟹ *hd* (*tl* (*stks n stk*)) = *stk*(*n* − *2*)
⟨*proof*⟩

lemma *stkss-purge*:
  length cs ≤ *a* ⟹ *stkss P cs* (*stk*((*a*,*b*) := *c*)) = *stkss P cs stk*
⟨*proof*⟩

lemma *stkss-purge'*:

length cs $\leq$ *a* $\Longrightarrow$ *stkss P cs* (λ*s*. *if s* = (*a*, *b*) *then c else stk s*) = *stkss P cs stk*
⟨*proof*⟩

lemma *locss-purge*:
length cs $\leq$ *a* $\Longrightarrow$ *locss P cs* (*loc*((*a*,*b*) := *c*)) = *locss P cs loc*
⟨*proof*⟩

lemma *locss-purge'*:
length cs $\leq$ *a* $\Longrightarrow$ *locss P cs* (λ*s*. *if s* = (*a*, *b*) *then c else loc s*) = *locss P cs loc*
⟨*proof*⟩

lemma *locs-pullout* [*simp*]:
locs b (*loc*(*n* := *e*)) = *locs b loc* [*n* := *e*]
⟨*proof*⟩

lemma *locs-pullout'* [*simp*]:
locs b (λ*a*. *if a* = *n then e else loc* (*c*, *a*)) = *locs b* (λ*a*. *loc* (*c*, *a*)) [*n* := *e*]
⟨*proof*⟩

lemma *stks-pullout*:
n < *b* $\Longrightarrow$ *stks b* (*stk*(*n* := *e*)) = *stks b stk* [*b* − *Suc n* := *e*]
⟨*proof*⟩

lemma *nth-tl* : *xs* $\neq$ [] $\Longrightarrow$ *tl xs* ! *n* = *xs* ! (*Suc n*)
⟨*proof*⟩

lemma *f2c-Nil* [*simp*]: *framestack-to-callstack* [] = []
⟨*proof*⟩

lemma *f2c-Cons* [*simp*]:
framestack-to-callstack ((*stk*,*loc*,*C*,*M*,*pc*)#*frs*) = (*C*,*M*,*pc*)#(*framestack-to-callstack frs*)
⟨*proof*⟩

lemma *f2c-length* [*simp*]:
length (*framestack-to-callstack frs*) = *length frs*
⟨*proof*⟩

lemma *f2c-s2jvm-id* [*simp*]:
framestack-to-callstack
(*snd*(*snd*(*state-to-jvm-state P cs s*))) =
cs
⟨*proof*⟩

lemma *f2c-s2jvm-id'* [*simp*]:
framestack-to-callstack
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) = *cs*

⟨*proof*⟩

lemma *f2c-append* [*simp*]:
framestack-to-callstack (*frs* @ *frs'*) =
(*framestack-to-callstack frs*) @ (*framestack-to-callstack frs'*)
⟨*proof*⟩

5.1.4 CFG construction

5.1.5 Datatypes

Nodes are labeled with a callstack and an optional tuple (consisting of a callstack and a flag).

The first callstack determines the current program point (i.e. the next statement to execute). If the second parameter is not None, we are at an intermediate state, where the target of the instruction is determined (the second callstack) and the flag is set to whether an exception is thrown or not.

datatype *j-node* =
Entry ('(*'-Entry'-*'))
| *Node callstack* (*callstack* × *bool*) *option* ('(*'- -,- '-*'))

The empty callstack indicates the exit node

abbreviation *j-node-Exit* :: *j-node* ('(*'-Exit'-*'))
where *j-node-Exit* ≡ (- [],*None* -)

An edge is a triple, consisting of two nodes and the edge kind

type-synonym *j-edge* = (*j-node* × *state edge-kind* × *j-node*)

5.1.6 CFG

The CFG is constructed by a case analysis on the instructions and their different behavior in different states. E.g. the exceptional behavior of New, if there is no more space in the heap, vs. the normal behavior.

Note: The set of edges defined by this predicate is a first approximation to the real set of edges in the CFG. We later (theory JVMInterpretation) add some well-formedness requirements to the nodes.

inductive *JVM-CFG* :: *jvmprog* ⇒ *j-node* ⇒ *state edge-kind* ⇒ *j-node* ⇒ *bool*
(- ⊢ - −-→ -)
where
JCFG-EntryExit:
prog ⊢ (-*Entry*-) −(λ*s*. *False*)$_{\surd}$→ (-*Exit*-)

| *JCFG-EntryStart*:
prog = (*P*, *C0*, *Main*) ⟹ *prog* ⊢ (-*Entry*-) −(λ*s*. *True*)$_{\surd}$→ (- [(*C0*, *Main*, *0*)],*None* -)

| *JCFG-ReturnExit*:
  ⟦ *prog* = (*P*,*C0*,*Main*);
    (*instrs-of* (P_{wf}) *C M*) ! *pc* = *Return* ⟧
    ⟹ *prog* ⊢ (- [(*C*, *M*, *pc*)],*None* -) −⇑*id*→ (-*Exit*-)

| *JCFG-Straight-NoExc*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    instrs-of (P_{wf}) *C M* ! *pc* ∈ {*Load idx*, *Store idx*, *Push val*, *Pop*, *IAdd*, *CmpEq*};
    ek = ⇑(λ*s*. *exec-instr* ((*instrs-of* (P_{wf}) *C M*) ! *pc*) *P s*
                    (*length cs*) (*stkLength P C M pc*) *arbitrary arbitrary*) ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-New-Normal-Pred*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
    ek = $(\lambda(h,stk,loc).\ new\text{-}Addr\ h \neq None)_{\surd}$⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*,*False*)⌋ -)

| *JCFG-New-Normal-Update*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
    ek = ⇑(λ*s*. *exec-instr* (*New Cl*) *P s* (*length cs*) (*stkLength P C M pc*) *arbitrary arbitrary*) ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*, *M*, *Suc pc*)#*cs*, *False*)⌋ -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-New-Exc-Pred*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
    find-handler-for P OutOfMemory ((*C*, *M*, *pc*)#*cs*) = *cs*′;
    ek = $(\lambda(h,stk,loc).\ new\text{-}Addr\ h = None)_{\surd}$ ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′,*True*)⌋ -)

| *JCFG-New-Exc-Update*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (P_{wf}) *C M*) ! *pc* = (*New Cl*);
    find-handler-for P OutOfMemory ((*C*, *M*, *pc*)#*cs*) = (*C*′, *M*′, *pc*′)#*cs*′;
    ek = ⇑(λ(*h*,*stk*,*loc*).
     (*h*,
       stk((*length cs*′,(*stkLength P C*′ *M*′ *pc*′) − 1) := *Addr* (*addr-of-sys-xcpt OutOfMemory*)),
      loc)
) ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*′, *M*′, *pc*′)#*cs*′, *True*)⌋ -) −*ek*→ (- (*C*′, *M*′, *pc*′)#*cs*′,*None* -)

| *JCFG-New-Exc-Exit*:
  ⟦ *prog* = (*P*, *C0*, *Main*);

$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (New\ Cl)$;
$find\text{-}handler\text{-}for\ P\ OutOfMemory\ ((C,\ M,\ pc)\#cs) = []$ ⟧
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor([],\ True)\rfloor\ \text{-}) -\Uparrow id\rightarrow (\text{-}Exit\text{-})$

| *JCFG-Getfield-Normal-Pred*:
⟦ $prog = (P,\ C0,\ Main)$;
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Getfield\ Fd\ Cl)$;
$ek = (\lambda(h,stk,loc).\ \ stk(length\ cs,\ stkLength\ P\ C\ M\ pc - 1) \neq Null)_{\surd}$ ⟧
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-}) -ek\rightarrow (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor((C,\ M,\ Suc\ pc)\#cs,\ False)\rfloor\ \text{-})$

| *JCFG-Getfield-Normal-Update*:
⟦ $prog = (P,\ C0,\ Main)$;
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Getfield\ Fd\ Cl)$;
$ek = \Uparrow(\lambda s.\ exec\text{-}instr\ (Getfield\ Fd\ Cl)\ P\ s\ (length\ cs)\ (stkLength\ P\ C\ M\ pc)\ arbitrary\ arbitrary)$ ⟧
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor((C,\ M,\ Suc\ pc)\#cs,\ False)\rfloor\ \text{-}) -ek\rightarrow (\text{-}\ (C,\ M,\ Suc\ pc)\#cs,None\ \text{-})$

| *JCFG-Getfield-Exc-Pred*:
⟦ $prog = (P,\ C0,\ Main)$;
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Getfield\ Fd\ Cl)$;
$find\text{-}handler\text{-}for\ P\ NullPointer\ ((C,\ M,\ pc)\#cs) = cs'$;
$ek = (\lambda(h,stk,loc).\ stk(length\ cs,\ stkLength\ P\ C\ M\ pc - 1) = Null)_{\surd}$ ⟧
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-}) -ek\rightarrow (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor(cs',\ True)\rfloor\ \text{-})$

| *JCFG-Getfield-Exc-Update*:
⟦ $prog = (P,\ C0,\ Main)$;
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Getfield\ Fd\ Cl)$;
$find\text{-}handler\text{-}for\ P\ NullPointer\ ((C,\ M,\ pc)\#cs) = (C',\ M',\ pc')\#cs'$;
$ek = \Uparrow(\lambda(h,stk,loc).$
$(h,$
$stk((length\ cs',(stkLength\ P\ C'\ M'\ pc') - 1) := Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer)),$
$loc)$
$)$ ⟧
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor((C',\ M',\ pc')\#cs',\ True)\rfloor\ \text{-}) -ek\rightarrow (\text{-}\ (C',\ M',\ pc')\#cs',None\ \text{-})$

| *JCFG-Getfield-Exc-Exit*:
⟦ $prog = (P,\ C0,\ Main)$;
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Getfield\ Fd\ Cl)$;
$find\text{-}handler\text{-}for\ P\ NullPointer\ ((C,\ M,\ pc)\#cs) = []$ ⟧
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor([],\ True)\rfloor\ \text{-}) -\Uparrow id\rightarrow (\text{-}Exit\text{-})$

| *JCFG-Putfield-Normal-Pred*:
⟦ $prog = (P,\ C0,\ Main)$;
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Putfield\ Fd\ Cl)$;
$ek = (\lambda(h,stk,loc).\ \ stk(length\ cs,\ stkLength\ P\ C\ M\ pc - 2) \neq Null)_{\surd}$ ⟧

$\Longrightarrow$ *prog* $\vdash$ (- (*C*, *M*, *pc*)#*cs*,*None* -) $-ek\rightarrow$ (- (*C*, *M*, *pc*)#*cs*,$\lfloor$((*C*, *M*, *Suc pc*)#*cs*, *False*)$\rfloor$ -)

| *JCFG-Putfield-Normal-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
ek = $\Uparrow$(λ*s*. *exec-instr* (*Putfield Fd Cl*) *P s* (*length cs*) (*stkLength P C M pc*)
arbitrary arbitrary) ⟧
$\Longrightarrow$ *prog* $\vdash$ (- (*C*, *M*, *pc*)#*cs*,$\lfloor$((*C*, *M*, *Suc pc*)#*cs*, *False*)$\rfloor$ -) $-ek\rightarrow$ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-Putfield-Exc-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = *cs*′;
ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *2*) = *Null*)$_{\surd}$ ⟧
$\Longrightarrow$ *prog* $\vdash$ (- (*C*, *M*, *pc*)#*cs*,*None* -) $-ek\rightarrow$ (- (*C*, *M*, *pc*)#*cs*,$\lfloor$(*cs*′, *True*)$\rfloor$ -)

| *JCFG-Putfield-Exc-Update*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = (*C*′, *M*′, *pc*′)#*cs*′;
ek = $\Uparrow$(λ(*h*,*stk*,*loc*).
(*h*,
stk((*length cs*′,(*stkLength P C*′ *M*′ *pc*′) − *1*) := *Addr* (*addr-of-sys-xcpt NullPointer*)),
loc)
) ⟧
$\Longrightarrow$ *prog* $\vdash$ (- (*C*, *M*, *pc*)#*cs*,$\lfloor$((*C*′, *M*′, *pc*′)#*cs*′, *True*)$\rfloor$ -) $-ek\rightarrow$ (- (*C*′, *M*′, *pc*′)#*cs*′,*None* -)

| *JCFG-Putfield-Exc-Exit*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Putfield Fd Cl*);
find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = [] ⟧
$\Longrightarrow$ *prog* $\vdash$ (- (*C*, *M*, *pc*)#*cs*,$\lfloor$([], *True*)$\rfloor$ -) $-\Uparrow id\rightarrow$ (-*Exit*-)

| *JCFG-Checkcast-Normal-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Checkcast Cl*);
ek = (λ(*h*,*stk*,*loc*). *cast-ok* (P_{wf}) *Cl h* (*stk*(*length cs*, *stkLength P C M pc* − *Suc 0*)))$_{\surd}$ ⟧
$\Longrightarrow$ *prog* $\vdash$ (- (*C*, *M*, *pc*)#*cs*,*None* -) $-ek\rightarrow$ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-Checkcast-Exc-Pred*:
⟦ *prog* = (*P*, *C0*, *Main*);
(*instrs-of* (P_{wf}) *C M*) ! *pc* = (*Checkcast Cl*);
find-handler-for P ClassCast ((*C*, *M*, *pc*)#*cs*) = *cs*′;
ek = (λ(*h*,*stk*,*loc*). ¬ *cast-ok* (P_{wf}) *Cl h* (*stk*(*length cs*, *stkLength P C M pc* −

$Suc\ 0)))_{\surd}\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-})\ -ek\rightarrow\ (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor(cs',\ True)\rfloor\ \text{-})$

| *JCFG-Checkcast-Exc-Update*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Checkcast\ Cl);$
$find\text{-}handler\text{-}for\ P\ ClassCast\ ((C,\ M,\ pc)\#cs) = (C',\ M',\ pc')\#cs';$
$ek = \Uparrow(\lambda(h,stk,loc).$
$(h,$
$stk((length\ cs',(stkLength\ P\ C'\ M'\ pc')\ -\ 1)\ :=\ Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ ClassCast)),$
$loc)$
$)\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor((C',\ M',\ pc')\#cs',\ True)\rfloor\ \text{-})\ -ek\rightarrow\ (\text{-}\ (C',\ M',\ pc')\#cs',None\ \text{-})$

| *JCFG-Checkcast-Exc-Exit*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Checkcast\ Cl);$
$find\text{-}handler\text{-}for\ P\ ClassCast\ ((C,\ M,\ pc)\#cs) = []\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor([],\ True)\rfloor\ \text{-})\ -\Uparrow id\rightarrow\ (\text{-}Exit\text{-})$

| *JCFG-Invoke-Normal-Pred*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Invoke\ M2\ n);$
$cd = length\ cs;$
$stk\text{-}length = stkLength\ P\ C\ M\ pc;$
$ek = (\lambda(h,stk,loc).$
$stk(cd,\ stk\text{-}length\ -\ Suc\ n) \neq Null\ \wedge$
$fst(method\ (P_{wf})\ (cname\text{-}of\ h\ (the\text{-}Addr(stk(cd,\ stk\text{-}length\ -\ Suc\ n))))\ M2) = D$
$)_{\surd}\ ]\!]$
$\Longrightarrow$
$prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-})\ -ek\rightarrow\ (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor((D,\ M2,\ 0)\#(C,\ M,\ pc)\#cs,\ False)\rfloor\ \text{-})$

| *JCFG-Invoke-Normal-Update*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Invoke\ M2\ n);$
$stk\text{-}length = stkLength\ P\ C\ M\ pc;$
$loc\text{-}length = locLength\ P\ D\ M2\ 0;$
$ek = \Uparrow(\lambda s.\ exec\text{-}instr\ (Invoke\ M2\ n)\ P\ s\ (length\ cs)\ stk\text{-}length\ arbitrary\ loc\text{-}length)$
$]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor((D,\ M2,\ 0)\#(C,\ M,\ pc)\#cs,\ False)\rfloor\ \text{-})\ -ek\rightarrow$
$(\text{-}\ (D,\ M2,\ 0)\#(C,\ M,\ pc)\#cs,None\ \text{-})$

| *JCFG-Invoke-Exc-Pred*:
$[\![\ prog = (P,\ C0,\ Main);$

$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Invoke\ m2\ n);$
$find\text{-}handler\text{-}for\ P\ NullPointer\ ((C,\ M,\ pc)\#cs) = cs';$
$ek = (\lambda(h,stk,loc).\ stk(length\ cs,\ stkLength\ P\ C\ M\ pc - Suc\ n) = Null)_{\surd}\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-}) -ek\rightarrow (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor(cs',\ True)\rfloor\ \text{-})$

| *JCFG-Invoke-Exc-Update*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Invoke\ M2\ n);$
$find\text{-}handler\text{-}for\ P\ NullPointer\ ((C,\ M,\ pc)\#cs) = (C',\ M',\ pc')\#cs';$
$ek = \Uparrow(\lambda(h,stk,loc).$
$(h,$
$stk((length\ cs',(stkLength\ P\ C'\ M'\ pc') - 1) := Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer)),$
$loc)$
$)$
$]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor((C',\ M',\ pc')\#cs',\ True)\rfloor\ \text{-}) -ek\rightarrow (\text{-}\ (C',\ M',\ pc')\#cs',None\ \text{-})$

| *JCFG-Invoke-Exc-Exit*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (Invoke\ M2\ n);$
$find\text{-}handler\text{-}for\ P\ NullPointer\ ((C,\ M,\ pc)\#cs) = []\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,\lfloor([],\ True)\rfloor\ \text{-}) -\Uparrow id\rightarrow (\text{-}Exit\text{-})$

| *JCFG-Return-Update*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = Return;$
$stk\text{-}length = stkLength\ P\ C\ M\ pc;$
$r\text{-}stk\text{-}length = stkLength\ P\ C'\ M'\ (Suc\ pc');$
$ek = \Uparrow(\lambda s.\ exec\text{-}instr\ Return\ P\ s\ (Suc\ (length\ cs))\ stk\text{-}length\ r\text{-}stk\text{-}length\ arbitrary)\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#(C',\ M',\ pc')\#cs,None\ \text{-}) -ek\rightarrow (\text{-}\ (C',\ M',\ Suc\ pc')\#cs,None\ \text{-})$

| *JCFG-Goto-Update*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = Goto\ idx\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-}) -\Uparrow id\rightarrow (\text{-}\ (C,\ M,\ nat\ (int\ pc + idx))\#cs,None\ \text{-})$

| *JCFG-IfFalse-False*:
$[\![\ prog = (P,\ C0,\ Main);$
$(instrs\text{-}of\ (P_{wf})\ C\ M)\ !\ pc = (IfFalse\ b);$
$b \neq 1;$
$ek = (\lambda(h,stk,loc).\ stk(length\ cs,\ stkLength\ P\ C\ M\ pc - 1) = Bool\ False)_{\surd}\ ]\!]$
$\Longrightarrow prog \vdash (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-}) -ek\rightarrow (\text{-}\ (C,\ M,\ nat\ (int\ pc + b))\#cs,None\ \text{-})$

| *JCFG-IfFalse-Next*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (*P*$_{wf}$) *C M*) ! *pc* = (*IfFalse b*);
    ek = (λ(*h*,*stk*,*loc*). *stk*(*length cs*, *stkLength P C M pc* − *1*) ≠ *Bool False* ∨ *b* = *1*)$_{\surd}$ ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *Suc pc*)#*cs*,*None* -)

| *JCFG-Throw-Pred*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (*P*$_{wf}$) *C M*) ! *pc* = *Throw*;
    cd = *length cs*;
    stk-length = *stkLength P C M pc*;
    ∃ *Exc*. *find-handler-for P Exc* ((*C*, *M*, *pc*)#*cs*) = *cs*′;
    ek = (λ(*h*,*stk*,*loc*).
      (*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null* ∧
        find-handler-for P NullPointer ((*C*, *M*, *pc*)#*cs*) = *cs*′) ∨
      (*stk*(*length cs*, *stkLength P C M pc* − *1*) ≠ *Null* ∧
         find-handler-for P (*cname-of h* (*the-Addr*(*stk*(*cd*, *stk-length* − *1*)))) ((*C*, *M*, *pc*)#*cs*) = *cs*′)
)$_{\surd}$ ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,*None* -) −*ek*→ (- (*C*, *M*, *pc*)#*cs*,⌊(*cs*′, *True*)⌋ -)

| *JCFG-Throw-Update*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (*P*$_{wf}$) *C M*) ! *pc* = *Throw*;
    ek = ⇑(λ(*h*,*stk*,*loc*).
      (*h*,
       stk((*length cs*′,(*stkLength P C*′ *M*′ *pc*′) − *1*) :=
         if (*stk*(*length cs*, *stkLength P C M pc* − *1*) = *Null*) *then*
           Addr (*addr-of-sys-xcpt NullPointer*)
         else (*stk*(*length cs*, *stkLength P C M pc* − *1*))),
       loc)
) ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊((*C*′, *M*′, *pc*′)#*cs*′, *True*)⌋ -) −*ek*→ (- (*C*′, *M*′, *pc*′)#*cs*′,*None* -)

| *JCFG-Throw-Exit*:
  ⟦ *prog* = (*P*, *C0*, *Main*);
    (*instrs-of* (*P*$_{wf}$) *C M*) ! *pc* = *Throw* ⟧
    ⟹ *prog* ⊢ (- (*C*, *M*, *pc*)#*cs*,⌊([],*True*)⌋ -) −⇑*id*→ (-*Exit*-)

5.1.7 CFG properties

lemma *JVMCFG-Exit-no-sourcenode* [*dest*]:
  assumes *edge*:*prog* ⊢ (-*Exit*-) −*et*→ *n*′
  shows *False*
⟨*proof*⟩

lemma *JVMCFG-Entry-no-targetnode* [*dest*]:

assumes *edge*:*prog* $\vdash$ n $-et\rightarrow$ (*-Entry-*)
shows *False*
⟨*proof*⟩

lemma *JVMCFG-EntryD*:
$[\![(P,C,M) \vdash n -et\rightarrow n'; n = (\text{-}Entry\text{-})]\!]$
$\Longrightarrow (n' = (\text{-}Exit\text{-}) \wedge et = (\lambda s.\ False)_{\surd}) \vee (n' = (\text{-}\ [(C,M,0)],None\ \text{-}) \wedge et = (\lambda s.\ True)_{\surd})$
⟨*proof*⟩

declare *split-def* [*simp add*]
declare *find-handler-for.simps* [*simp del*]

lemma *JVMCFG-edge-det*:
$[\![prog \vdash n -et\rightarrow n'; prog \vdash n -et'\rightarrow n']\!] \Longrightarrow et = et'$
⟨*proof*⟩

declare *split-def* [*simp del*]
declare *find-handler-for.simps* [*simp add*]

end

theory *JVMInterpretation* **imports** *JVMCFG* ../*Basic*/*CFGExit* **begin**

5.2 Instatiation of the *CFG* locale

abbreviation *sourcenode* :: *j-edge* ⇒ *j-node*
where *sourcenode e* ≡ *fst e*

abbreviation *targetnode* :: *j-edge* ⇒ *j-node*
where *targetnode e* ≡ *snd*(*snd e*)

abbreviation *kind* :: *j-edge* ⇒ *state edge-kind*
where *kind e* ≡ *fst*(*snd e*)

The following predicates define the aforementioned well-formedness requirements for nodes. Later, *valid-callstack* will be implied by Jinja's state conformance.

fun *valid-callstack* :: *jvmprog* ⇒ *callstack* ⇒ *bool*
where
valid-callstack prog [] = *True*
| *valid-callstack* (*P*, *C0*, *Main*) [(*C*, *M*, *pc*)] ⟷
C = *C0* ∧ *M* = *Main* ∧
(P_Φ) *C M* ! *pc* ≠ *None* ∧
(∃ *T Ts mxs mxl is xt*. (P_{wf}) ⊢ *C sees M*:*Ts*→*T*=(*mxs*, *mxl*, *is*, *xt*) *in C* ∧ *pc* < *length is*)

| *valid-callstack* $(P,\ C0,\ Main)\ ((C,\ M,\ pc)\#(C',\ M',\ pc')\#cs) \longleftrightarrow$
  instrs-of $(P_{wf})\ C'\ M'\ !\ pc' =$
    Invoke M (*locLength* $P\ C\ M\ 0\ -\ Suc$ (*fst*(*snd*(*snd*(*snd*(*snd*(*method* $(P_{wf})\ C\ M$))))))) $\wedge$
  $(P_{\Phi})\ C\ M\ !\ pc \neq None \wedge$
  ($\exists\ T\ Ts\ mxs\ mxl\ is\ xt.\ (P_{wf}) \vdash C$ *sees* $M{:}Ts\rightarrow T{=}(mxs,\ mxl,\ is,\ xt)$ *in* $C \wedge pc <$ *length* $is) \wedge$
  valid-callstack $(P,\ C0,\ Main)\ ((C',\ M',\ pc')\#cs)$

fun *valid-node* :: *jvmprog* $\Rightarrow$ *j-node* $\Rightarrow$ *bool*
where
  valid-node prog (*-Entry-*) $=$ *True*

| *valid-node prog* (*- cs,None -*) $\longleftrightarrow$ *valid-callstack prog cs*
| *valid-node prog* (*- cs,*$\lfloor$(*cs'*, *xf*)$\rfloor$ *-*) $\longleftrightarrow$
  valid-callstack prog cs $\wedge$ *valid-callstack prog cs'* $\wedge$
  ($\exists\ Q.$ *prog* $\vdash$ (*- cs,None -*) $-(Q)_{\surd}\rightarrow$ (*- cs,*$\lfloor$(*cs'*, *xf*)$\rfloor$ *-*)) $\wedge$
  ($\exists f.$ *prog* $\vdash$ (*- cs,*$\lfloor$(*cs'*, *xf*)$\rfloor$ *-*) $-\Uparrow f\rightarrow$ (*- cs',None -*))

fun *valid-edge* :: *jvmprog* $\Rightarrow$ *j-edge* $\Rightarrow$ *bool*
where
  valid-edge prog a $\longleftrightarrow$
    (*prog* $\vdash$ (*sourcenode a*) $-$(*kind a*)$\rightarrow$ (*targetnode a*))
    $\wedge$ (*valid-node prog* (*sourcenode a*))
    $\wedge$ (*valid-node prog* (*targetnode a*))

interpretation *JVM-CFG-Interpret*:
  CFG sourcenode targetnode kind valid-edge prog Entry
  for *prog*
⟨*proof*⟩

interpretation *JVM-CFGExit-Interpret*:
  CFGExit sourcenode targetnode kind valid-edge prog Entry (*-Exit-*)
  for *prog*
⟨*proof*⟩

end

Chapter 6

Standard and Weak Control Dependence

6.1 A type for well-formed programs

theory *JVMPostdomination* **imports** *JVMInterpretation ../Basic/Postdomination*
begin

For instantiating *Postdomination* every node in the CFG of a program must be reachable from the (*-Entry-*) node and there must be a path to the (*-Exit-*) node from each node.

Therefore, we restrict the set of allowed programs to those, where the CFG fulfills these requirements. This is done by defining a new type for well-formed programs. The universe of every type in Isabelle must be non-empty. That's why we first define an example program *EP* and its typing *Phi-EP*, which is a member of the carrier set of the later defined type.

Restricting the set of allowed programs in this way is reasonable, as Jinja's compiler only produces byte code programs, that are members of this type (A proof for this is current work).

definition *EP* :: *jvm-prog*
  where *EP* = (*''C''*, *Object*, [], [(*''M''*, [], *Void*, *1*::*nat*, *0*::*nat*, [*Push Unit*, *Return*], [])]) #
  SystemClasses

definition *Phi-EP* :: ty_P
 where *Phi-EP C M* = (*if C* = *''C''* ∧ *M* = *''M'' then* [⌊([],[*OK* (*Class ''C''*)])⌋,⌊([*Void*],[*OK* (*Class ''C''*)])⌋] *else* [])

Now we show, that *EP* is indeed a well-formed program in the sense of Jinja's byte code verifier

lemma *distinct-classes''*:
  ''C'' ≠ *Object*
  ''C'' ≠ *NullPointer*

$''C'' \neq OutOfMemory$
$''C'' \neq ClassCast$
⟨*proof*⟩

lemmas *distinct-classes* =
distinct-classes distinct-classes'' distinct-classes'' [*symmetric*]

declare *distinct-classes* [*simp add*]

lemma *i-max-2D*: $i < Suc\ (Suc\ 0) \Longrightarrow i = 0 \vee i = 1$
⟨*proof*⟩

lemma *EP-wf*: *wf-jvm-prog*$_{Phi\text{-}EP}$ *EP*
⟨*proof*⟩

lemma [*simp*]: *Abs-wf-jvmprog* $(EP,\ Phi\text{-}EP)_{wf} = EP$
⟨*proof*⟩

lemma [*simp*]: *Abs-wf-jvmprog* $(EP,\ Phi\text{-}EP)_{\Phi} = Phi\text{-}EP$
⟨*proof*⟩

lemma *method-in-EP-is-M*:
$EP \vdash C\ sees\ M: Ts \to T = (mxs,\ mxl,\ is,\ xt)\ in\ D$
$\Longrightarrow C = ''C'' \wedge$
$M = ''M'' \wedge$
$Ts = [] \wedge$
$T = Void \wedge$
$mxs = 1 \wedge$
$mxl = 0 \wedge$
$is = [Push\ Unit,\ Return] \wedge$
$xt = [] \wedge$
$D = ''C''$
⟨*proof*⟩

lemma [*simp*]:
$\exists\, T\ Ts\ mxs\ mxl\ is.\ (\exists\, xt.\ EP \vdash ''C''\ sees\ ''M'': Ts \to T = (mxs,\ mxl,\ is,\ xt)\ in\ ''C'') \wedge is \neq []$
⟨*proof*⟩

lemma [*simp*]:
$\exists\, T\ Ts\ mxs\ mxl\ is.\ (\exists\, xt.\ EP \vdash ''C''\ sees\ ''M'': Ts \to T = (mxs,\ mxl,\ is,\ xt)\ in\ ''C'') \wedge$
$Suc\ 0 < length\ is$
⟨*proof*⟩

lemma *C-sees-M-in-EP* [*simp*]:
$EP \vdash ''C''\ sees\ ''M'': [] \to Void = (1,\ 0,\ [Push\ Unit,\ Return],\ [])\ in\ ''C''$

⟨*proof*⟩

lemma *instrs-of-EP-C-M* [*simp*]:
instrs-of EP ''C'' ''M'' = [Push Unit, Return]
⟨*proof*⟩

lemma *valid-node-in-EP-D*:
valid-node (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'') n
⟹ n ∈ {(-Entry-), (- [(''C'', ''M'', 0)],None -), (- [(''C'', ''M'', 1)],None -), (-Exit-)}
⟨*proof*⟩

lemma *EP-C-M-0-valid* [*simp*]:
JVM-CFG-Interpret.valid-node (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'')
(- [(''C'', ''M'', 0)],None -)
⟨*proof*⟩

lemma *EP-C-M-Suc-0-valid* [*simp*]:
JVM-CFG-Interpret.valid-node (Abs-wf-jvmprog (EP, Phi-EP), ''C'', ''M'')
(- [(''C'', ''M'', Suc 0)],None -)
⟨*proof*⟩

definition
cfg-wf-prog =
{P. (∀ n. valid-node P n ⟶
(∃ as. JVM-CFG-Interpret.path P (-Entry-) as n) ∧
(∃ as. JVM-CFG-Interpret.path P n as (-Exit-)))}

typedef *cfg-wf-prog = cfg-wf-prog*
⟨*proof*⟩

abbreviation *lift-to-cfg-wf-prog* :: *(jvmprog ⇒ 'a) ⇒ (cfg-wf-prog ⇒ 'a)*
(-$_{CFG}$)
where *f $_{CFG}$ ≡ (λP. f (Rep-cfg-wf-prog P))*

6.2 Interpretation of the *Postdomination* locale

interpretation *JVM-CFG-Postdomination*:
Postdomination sourcenode targetnode kind valid-edge $_{CFG}$ prog Entry (-Exit-)
for *prog*
⟨*proof*⟩

6.3 Interpretation of the *StrongPostdomination* locale

6.3.1 Some helpfull lemmas

lemma *find-handler-for-tl-eq*:
find-handler-for P Exc cs = (*C*,*M*,*pcx*)#*cs'* ⟹ ∃ *cs''* *pc*. *cs* = *cs''* @ [(*C*,*M*,*pc*)] @ *cs'*
⟨*proof*⟩

lemma *valid-callstack-tl*:
valid-callstack prog ((*C*,*M*,*pc*)#*cs*) ⟹ *valid-callstack prog cs*
⟨*proof*⟩

lemma *find-handler-Throw-Invoke-pc-in-range*:
⟦*cs* = (*C'*,*M'*,*pc'*)#*cs'*; *valid-callstack* (*P*,*C0*,*Main*) *cs*;
instrs-of (P_{wf}) *C' M'* ! *pc'* = *Throw* ∨ (∃ *M'' n''*. *instrs-of* (P_{wf}) *C' M'* ! *pc'* = *Invoke M'' n''*);
find-handler-for P Exc cs = (*C*,*M*,*pc*)#*cs''* ⟧
⟹ *pc* < *length* (*instrs-of* (P_{wf}) *C M*)
⟨*proof*⟩

6.3.2 Every node has only finitely many successors

lemma *successor-set-finite*:
JVM-CFG-Interpret.valid-node prog n
⟹ *finite* {*n'*. ∃ *a'*. *valid-edge prog a'* ∧ *sourcenode a'* = *n* ∧
targetnode a' = *n'*}
⟨*proof*⟩

6.3.3 Interpretation of the locale

interpretation *JVM-CFG-StrongPostdomination*:
StrongPostdomination sourcenode targetnode kind valid-edge $_{CFG}$ *prog Entry* (*-Exit-*)
for *prog*
⟨*proof*⟩

end

theory *JVMCFG-wf* **imports** *JVMInterpretation* ../*Basic*/*CFGExit-wf* **begin**

6.4 Instantiation of the *CFG-wf* locale

6.4.1 Variables and Values

datatype *jinja-var* = *HeapVar addr* | *Stk nat nat* | *Loc nat nat*
datatype *jinja-val* = *Object obj option* | *Primitive val*

fun *state-val* :: *state* ⇒ *jinja-var* ⇒ *jinja-val*
where
  state-val (*h*, *stk*, *loc*) (*HeapVar a*) = *Object* (*h a*)
| *state-val* (*h*, *stk*, *loc*) (*Stk cd idx*) = *Primitive* (*stk* (*cd*, *idx*))
| *state-val* (*h*, *stk*, *loc*) (*Loc cd idx*) = *Primitive* (*loc* (*cd*, *idx*))

6.4.2 The *Def* and *Use* sets

inductive-set *Def* :: *wf-jvmprog* ⇒ *j-node* ⇒ *jinja-var set*
  for *P* :: *wf-jvmprog*
  and *n* :: *j-node*
where
  Def-Load:
  ⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
  instrs-of (*P*$_{wf}$) *C M* ! *pc* = *Load idx*;
  cd = *length cs*;
  i = *stkLength P C M pc*⟧
  ⟹ *Stk cd i* ∈ *Def P n*

| *Def-Store*:
  ⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
  instrs-of (*P*$_{wf}$) *C M* ! *pc* = *Store idx*;
  cd = *length cs* ⟧
  ⟹ *Loc cd idx* ∈ *Def P n*

| *Def-Push*:
  ⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
  instrs-of (*P*$_{wf}$) *C M* ! *pc* = *Push v*;
  cd = *length cs*;
  i = *stkLength P C M pc* ⟧
  ⟹ *Stk cd i* ∈ *Def P n*

| *Def-New-Normal-Stk*:
  ⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,⌊(*cs'*,*False*)⌋ -);
  instrs-of (*P*$_{wf}$) *C M* ! *pc* = *New Cl*;
  cd = *length cs*;
  i = *stkLength P C M pc* ⟧
  ⟹ *Stk cd i* ∈ *Def P n*

| *Def-New-Normal-Heap*:
  ⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,⌊(*cs'*,*False*)⌋ -);
  instrs-of (*P*$_{wf}$) *C M* ! *pc* = *New Cl* ⟧
  ⟹ *HeapVar a* ∈ *Def P n*

| *Def-Exc-Stk*:
  ⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,⌊(*cs'*,*True*)⌋ -);
  cs' ≠ [];
  cd = *length cs'* − *1*;

$(C',M',pc') = hd\ cs';$
$i = stkLength\ P\ C'\ M'\ pc' - 1 ⟧$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-Getfield-Stk*:
$⟦\ n = (\text{-}\ (C,\ M,\ pc)\#cs,⌊(cs',False)⌋\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Getfield\ Fd\ Cl;$
$cd = length\ cs;$
$i = stkLength\ P\ C\ M\ pc - 1\ ⟧$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-Putfield-Heap*:
$⟦\ n = (\text{-}\ (C,\ M,\ pc)\#cs,⌊(cs',False)⌋\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Putfield\ Fd\ Cl\ ⟧$
$\Longrightarrow HeapVar\ a \in Def\ P\ n$

| *Def-Invoke-Loc*:
$⟦\ n = (\text{-}\ (C,\ M,\ pc)\#cs,⌊(cs',False)⌋\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Invoke\ M'\ n';$
$cs' \neq [];$
$hd\ cs' = (C',M',0);$
$i < locLength\ P\ C'\ M'\ 0;$
$cd = Suc\ (length\ cs)\ ⟧$
$\Longrightarrow Loc\ cd\ i \in Def\ P\ n$

| *Def-Return-Stk*:
$⟦\ n = (\text{-}\ (C,\ M,\ pc)\#(D,M',pc')\#cs,None\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Return;$
$cd = length\ cs;$
$i = stkLength\ P\ D\ M'\ (Suc\ pc') - 1\ ⟧$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-IAdd-Stk*:
$⟦\ n = (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = IAdd;$
$cd = length\ cs;$
$i = stkLength\ P\ C\ M\ pc - 2\ ⟧$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-CmpEq-Stk*:
$⟦\ n = (\text{-}\ (C,\ M,\ pc)\#cs,None\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = CmpEq;$
$cd = length\ cs;$
$i = stkLength\ P\ C\ M\ pc - 2\ ⟧$
$\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

inductive-set *Use* :: *wf-jvmprog* ⇒ *j-node* ⇒ *jinja-var set*
for P :: *wf-jvmprog*
and n :: *j-node*

where
Use-Load:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, None\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Load\ idx;$
$cd = length\ cs\ ]\!]$
$\Longrightarrow (Loc\ cd\ idx) \in Use\ P\ n$

| *Use-Store*:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, None\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Store\ idx;$
$cd = length\ cs;$
$Suc\ i = (stkLength\ P\ C\ M\ pc)\ ]\!]$
$\Longrightarrow (Stk\ cd\ i) \in Use\ P\ n$

| *Use-New*:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, x\ \text{-});$
$x = None \vee x = \lfloor(cs', False)\rfloor;$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = New\ Cl\ ]\!]$
$\Longrightarrow HeapVar\ a \in Use\ P\ n$

| *Use-Getfield-Stk*:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, x\ \text{-});$
$x = None \vee x = \lfloor(cs', False)\rfloor;$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Getfield\ Fd\ Cl;$
$cd = length\ cs;$
$Suc\ i = stkLength\ P\ C\ M\ pc\ ]\!]$
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Getfield-Heap*:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, \lfloor(cs', False)\rfloor\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Getfield\ Fd\ Cl\ ]\!]$
$\Longrightarrow HeapVar\ a \in Use\ P\ n$

| *Use-Putfield-Stk-Pred*:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, None\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Putfield\ Fd\ Cl;$
$cd = length\ cs;$
$i = stkLength\ P\ C\ M\ pc - 2\ ]\!]$
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Putfield-Stk-Update*:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, \lfloor(cs', False)\rfloor\ \text{-});$
$instrs\text{-}of\ (P_{wf})\ C\ M\ !\ pc = Putfield\ Fd\ Cl;$
$cd = length\ cs;$
$i = stkLength\ P\ C\ M\ pc - 2 \vee i = stkLength\ P\ C\ M\ pc - 1\ ]\!]$
$\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Putfield-Heap*:
$[\![\ n = (\text{-}\ (C,\ M,\ pc)\#cs, \lfloor(cs', False)\rfloor\ \text{-});$

instrs-of (P_{wf}) *C M* ! *pc* = *Putfield Fd Cl* ⟧
⟹ *HeapVar a* ∈ *Use P n*

| *Use-Checkcast-Stk*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*x* -);
x = *None* ∨ *x* = ⌊(*cs'*,*False*)⌋;
instrs-of (P_{wf}) *C M* ! *pc* = *Checkcast Cl*;
cd = *length cs*;
i = *stkLength P C M pc* − *Suc 0* ⟧
⟹ *Stk cd i* ∈ *Use P n*

| *Use-Checkcast-Heap*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Checkcast Cl* ⟧
⟹ *HeapVar a* ∈ *Use P n*

| *Use-Invoke-Stk-Pred*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Invoke M' n'*;
cd = *length cs*;
i = *stkLength P C M pc* − *Suc n'* ⟧
⟹ *Stk cd i* ∈ *Use P n*

| *Use-Invoke-Heap-Pred*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Invoke M' n'* ⟧
⟹ *HeapVar a* ∈ *Use P n*

| *Use-Invoke-Stk-Update*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,⌊(*cs'*,*False*)⌋ -);
instrs-of (P_{wf}) *C M* ! *pc* = *Invoke M' n'*;
cd = *length cs*;
i < *stkLength P C M pc*;
i ≥ *stkLength P C M pc* − *Suc n'* ⟧
⟹ *Stk cd i* ∈ *Use P n*

| *Use-Return-Stk*:
⟦ *n* = (- (*C*, *M*, *pc*)#(*D*,*M'*,*pc'*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *Return*;
cd = *Suc* (*length cs*);
i = *stkLength P C M pc* − *1* ⟧
⟹ *Stk cd i* ∈ *Use P n*

| *Use-IAdd-Stk*:
⟦ *n* = (- (*C*, *M*, *pc*)#*cs*,*None* -);
instrs-of (P_{wf}) *C M* ! *pc* = *IAdd*;
cd = *length cs*;
i = *stkLength P C M pc* − *1* ∨ *i* = *stkLength P C M pc* − *2* ⟧
⟹ *Stk cd i* ∈ *Use P n*

| *Use-IfFalse-Stk*:
  ⟦ $n = (\text{-}\ (C,\ M,\ pc)\#cs, None\ \text{-})$;
  instrs-of $(P_{wf})\ C\ M\ !\ pc = (IfFalse\ b)$;
  $cd = length\ cs$;
  $i = stkLength\ P\ C\ M\ pc - 1$ ⟧
  $\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-CmpEq-Stk*:
  ⟦ $n = (\text{-}\ (C,\ M,\ pc)\#cs, None\ \text{-})$;
  instrs-of $(P_{wf})\ C\ M\ !\ pc = CmpEq$;
  $cd = length\ cs$;
  $i = stkLength\ P\ C\ M\ pc - 1 \vee i = stkLength\ P\ C\ M\ pc - 2$ ⟧
  $\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Throw-Stk*:
  ⟦ $n = (\text{-}\ (C,\ M,\ pc)\#cs, x\ \text{-})$;
  $x = None \vee x = \lfloor (cs', True)\rfloor$;
  instrs-of $(P_{wf})\ C\ M\ !\ pc = Throw$;
  $cd = length\ cs$;
  $i = stkLength\ P\ C\ M\ pc - 1$ ⟧
  $\Longrightarrow Stk\ cd\ i \in Use\ P\ n$

| *Use-Throw-Heap*:
  ⟦ $n = (\text{-}\ (C,\ M,\ pc)\#cs, x\ \text{-})$;
  $x = None \vee x = \lfloor (cs', True)\rfloor$;
  instrs-of $(P_{wf})\ C\ M\ !\ pc = Throw$ ⟧
  $\Longrightarrow HeapVar\ a \in Use\ P\ n$

declare *correct-state-def* [*simp del*]

lemma *edge-transfer-uses-only-Use*:
  ⟦*valid-edge* $(P, C0, Main)\ a$; $\forall V \in Use\ P\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$⟧
  $\Longrightarrow \forall V \in Def\ P\ (sourcenode\ a).\ state\text{-}val\ (BasicDefs.transfer\ (kind\ a)\ s)\ V =$
  $state\text{-}val\ (BasicDefs.transfer\ (kind\ a)\ s')\ V$
⟨*proof*⟩

lemma *CFG-edge-Uses-pred-equal*:
  ⟦ *valid-edge* $(P, C0, Main)\ a$;
  pred $(kind\ a)\ s$;
  $\forall V \in Use\ P\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$⟧
  $\Longrightarrow$ *pred* $(kind\ a)\ s'$
⟨*proof*⟩

lemma *edge-no-Def-equal*:
  ⟦ *valid-edge* $(P,\ C0,\ Main)\ a$;
    $V \notin Def\ P\ (sourcenode\ a)$ ⟧

$\Longrightarrow$ *state-val* (*transfer* (*kind a*) *s*) *V* = *state-val s V*
⟨*proof*⟩

interpretation *JVM-CFG-wf*: *CFG-wf*
sourcenode targetnode kind valid-edge prog (*-Entry-*)
Def (*fst prog*) *Use* (*fst prog*) *state-val*
for *prog*
⟨*proof*⟩

interpretation *JVM-CFGExit-wf*: *CFGExit-wf*
sourcenode targetnode kind valid-edge prog (*-Entry-*)
Def (*fst prog*) *Use* (*fst prog*) *state-val* (*-Exit-*)
⟨*proof*⟩

end

6.5 Instantiating the control dependences

theory *JVMControlDependences* **imports**
JVMPostdomination
JVMCFG-wf
../Dynamic/DynPDG
../StaticIntra/CDepInstantiations
begin

6.5.1 Dynamic dependences

interpretation *JVMDynStandardControlDependence*:
DynStandardControlDependencePDG sourcenode targetnode kind
valid-edge$_{CFG}$ *prog* (*-Entry-*) *Def* (*fst*$_{CFG}$ *prog*) *Use* (*fst*$_{CFG}$ *prog*)
state-val (*-Exit-*) ⟨*proof*⟩

interpretation *JVMDynWeakControlDependence*:
DynWeakControlDependencePDG sourcenode targetnode kind
valid-edge$_{CFG}$ *prog* (*-Entry-*) *Def* (*fst*$_{CFG}$ *prog*) *Use* (*fst*$_{CFG}$ *prog*)
state-val (*-Exit-*) ⟨*proof*⟩

6.5.2 Static dependences

interpretation *JVMStandardControlDependence*:
StandardControlDependencePDG sourcenode targetnode kind
valid-edge$_{CFG}$ *prog* (*-Entry-*) *Def* (*fst*$_{CFG}$ *prog*) *Use* (*fst*$_{CFG}$ *prog*)
state-val (*-Exit-*) ⟨*proof*⟩

interpretation *JVMWeakControlDependence*:
WeakControlDependencePDG sourcenode targetnode kind
valid-edge$_{CFG}$ *prog* (*-Entry-*) *Def* (*fst*$_{CFG}$ *prog*) *Use* (*fst*$_{CFG}$ *prog*)

state-val (*-Exit-*) ⟨*proof*⟩

end

Chapter 7

Equivalence of the CFG and Jinja

theory *SemanticsWF* **imports** *JVMInterpretation ../Basic/SemanticsCFG* **begin**

declare *rev-nth* [*simp add*]

7.1 State updates

The following abbreviations update the stack and the local variables (in the representation as used in the CFG) according to a *frame list* as it is used in Jinja's state representation.

abbreviation *update-stk* :: ((*nat* × *nat*) ⇒ *val*) ⇒ (*frame list*) ⇒ ((*nat* × *nat*) ⇒ *val*)
where
update-stk stk frs ≡ (λ(*a*, *b*).
if length frs ≤ *a then stk* (*a*, *b*)
else let xs = *fst* (*frs* ! (*length frs* − *Suc a*))
in if length xs ≤ *b then stk* (*a*, *b*) *else xs* ! (*length xs* − *Suc b*))

abbreviation *update-loc* :: ((*nat* × *nat*) ⇒ *val*) ⇒ (*frame list*) ⇒ ((*nat* × *nat*) ⇒ *val*)
where
update-loc loc frs ≡ (λ(*a*, *b*).
if length frs ≤ *a then loc* (*a*, *b*)
else let xs = *fst* (*snd* (*frs* ! (*length frs* − *Suc a*)))
in if length xs ≤ *b then loc* (*a*, *b*) *else xs* ! *b*)

7.1.1 Some simplification lemmas

lemma *update-loc-s2jvm* [*simp*]:
update-loc loc (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = *loc*
⟨*proof*⟩

lemma *update-stk-s2jvm* [*simp*]:
update-stk stk (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = *stk*
⟨*proof*⟩

lemma *update-loc-s2jvm′* [*simp*]:
update-loc loc (*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) = *loc*
⟨*proof*⟩

lemma *update-stk-s2jvm′* [*simp*]:
update-stk stk (*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) = *stk*
⟨*proof*⟩

lemma *find-handler-find-handler-forD*:
find-handler (P_{wf}) *a h frs* = (*xp′*,*h′*,*frs′*)
⟹ *find-handler-for P* (*cname-of h a*) (*framestack-to-callstack frs*) =
framestack-to-callstack frs′
⟨*proof*⟩

lemma *find-handler-nonempty-frs* [*simp*]:
(*find-handler P a h frs* ≠ (*None*, *h′*, []))
⟨*proof*⟩

lemma *find-handler-heap-eqD*:
find-handler P a h frs = (*xp*, *h′*, *frs′*) ⟹ *h′* = *h*
⟨*proof*⟩

lemma *find-handler-frs-decrD*:
find-handler P a h frs = (*xp*, *h′*, *frs′*) ⟹ *length frs′* ≤ *length frs*
⟨*proof*⟩

lemma *find-handler-decrD* [*dest*]:
find-handler P a h frs = (*xp*, *h′*, *f*#*frs*) ⟹ *False*
⟨*proof*⟩

lemma *find-handler-decrD′* [*dest*]:
⟦ *find-handler P a h frs* = (*xp*,*h′*,*f*#*frs′*); *length frs* = *length frs′* ⟧ ⟹ *False*
⟨*proof*⟩

lemma *Suc-minus-Suc-Suc* [*simp*]:
b < *n* − *1* ⟹ *Suc* (*n* − *Suc* (*Suc b*)) = *n* − *Suc b*
⟨*proof*⟩

lemma *find-handler-loc-fun-eq′*:
find-handler (P_{wf}) *a h*
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) =
(*xf*, *h′*, *frs*)
⟹ *update-loc loc frs* = *loc*
⟨*proof*⟩

lemma *find-handler-loc-fun-eq*:
find-handler (P_{wf}) *a h* (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = (*xf*,*h'*,*frs*)
$\Longrightarrow$ *update-loc loc frs* = *loc*
⟨*proof*⟩

lemma *find-handler-stk-fun-eq'*:
⟦*find-handler* (P_{wf}) *a h*
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) =
(*None*, *h'*, *frs*);
cd = *length frs* − *1*;
i = *length* (*fst*(*hd*(*frs*))) − *1* ⟧
$\Longrightarrow$ *update-stk stk frs* = *stk*((*cd*, *i*) := *Addr a*)
⟨*proof*⟩

lemma *find-handler-stk-fun-eq*:
find-handler (P_{wf}) *a h* (*snd*(*snd*(*state-to-jvm-state P cs* (*h*,*stk*,*loc*)))) = (*None*,*h'*,*frs*)
$\Longrightarrow$ *update-stk stk frs* = *stk*((*length frs* − *1*, *length* (*fst*(*hd*(*frs*))) − *1*) := *Addr*
a)
⟨*proof*⟩

lemma *f2c-emptyD* [*dest*]:
framestack-to-callstack frs = [] $\Longrightarrow$ *frs* = []
⟨*proof*⟩

lemma *f2c-emptyD'* [*dest*]:
[] = *framestack-to-callstack frs* $\Longrightarrow$ *frs* = []
⟨*proof*⟩

lemma *correct-state-imp-valid-callstack*:
⟦ *P*,*cs* $\vdash_{BV}$ *s* $\surd$; *fst* (*last cs*) = *C0*; *fst*(*snd* (*last cs*)) = *Main* ⟧
$\Longrightarrow$ *valid-callstack* (*P*,*C0*,*Main*) *cs*
⟨*proof*⟩

declare *correct-state-def* [*simp del*]

lemma *bool-sym*: *Bool* (*a* = *b*) = *Bool* (*b* = *a*)
⟨*proof*⟩

lemma *find-handler-exec-correct*:
⟦(P_{wf}),(P_{Φ}) $\vdash$ *state-to-jvm-state P cs* (*h*,*stk*,*loc*) $\surd$;
(P_{wf}),(P_{Φ}) $\vdash$ *find-handler* (P_{wf}) *a h*
(*zip* (*stkss P cs stk*) (*zip* (*locss P cs loc*) *cs*)) $\surd$;
find-handler-for P (*cname-of h a*) *cs* = (*C'*, *M'*, *pc'*) # *cs'*
⟧ $\Longrightarrow$
(P_{wf}),(P_{Φ}) $\vdash$ (*None*, *h*,
(*stks* (*stkLength P C' M' pc'*)
(λ*a'*. (*stk*((*length cs'*, *stkLength P C' M' pc'* − *Suc 0*) := *Addr a*)) (*length*
cs', *a'*)),

$locs\ (locLength\ P\ C'\ M'\ pc')\ (\lambda a.\ loc\ (length\ cs',\ a)),\ C',\ M',\ pc')\ \#$
$zip\ (stkss\ P\ cs'\ stk)\ (zip\ (locss\ P\ cs'\ loc)\ cs'))\ \surd$
⟨*proof*⟩

lemma *locs-rev-stks*:
$x \geq z \Longrightarrow$
$locs\ z$
$(\lambda b.$
$if\ z < b\ then\ loc\ (Suc\ y,\ b)$
$else\ if\ b \leq z$
$then\ stk\ (y,\ x + b - Suc\ z)$
$else\ arbitrary)$
$@\ [stk\ (y,\ x - Suc\ 0)]$
$=$
$stk\ (y,\ x - Suc\ (z))$
$\#\ rev\ (take\ z\ (stks\ x\ (\lambda a.\ stk(y,\ a))))$
⟨*proof*⟩

lemma *locs-invoke-purge*:
$(z::nat) > c \Longrightarrow$
$locs\ l$
$(\lambda b.\ if\ z = c \longrightarrow Q\ b\ then\ loc\ (c,\ b)\ else\ u\ b) =$
$locs\ l\ (\lambda a.\ loc\ (c,\ a))$
⟨*proof*⟩

lemma *nth-rev-equalityI*:
$[\![length\ xs = length\ ys;\ \forall i{<}length\ xs.\ xs\ !\ (length\ xs - Suc\ i) = ys\ !\ (length\ ys - Suc\ i)]\!]$
$\Longrightarrow xs = ys$
⟨*proof*⟩

lemma *length-locss*:
$i < length\ cs$
$\Longrightarrow length\ (locss\ P\ cs\ loc\ !\ (length\ cs - Suc\ i)) =$
$locLength\ P\ (fst(cs\ !\ (length\ cs - Suc\ i)))$
$(fst(snd(cs\ !\ (length\ cs - Suc\ i))))$
$(snd(snd(cs\ !\ (length\ cs - Suc\ i))))$
⟨*proof*⟩

lemma *locss-invoke-purge*:
$z > length\ cs \Longrightarrow$
$locss\ P\ cs$
$(\lambda(a,\ b).\ if\ (a = z \longrightarrow Q\ b)$
$then\ loc\ (a,\ b)$
$else\ u\ b)$
$= locss\ P\ cs\ loc$
⟨*proof*⟩

lemma *stks-purge'*:
$d \geq b \Longrightarrow stks\ b\ (\lambda x.\ if\ x = d\ then\ e\ else\ stk\ x) = stks\ b\ stk$
⟨*proof*⟩

7.1.2 Byte code verifier conformance

Here we prove state conformance invariant under *transfer* for our CFG. Therefore, we must assume, that the predicate of a potential preceding predicate-edge holds for every update-edge.

theorem *bv-invariant*:
⟦ *valid-edge* (*P*,*C0*,*Main*) *a*;
sourcenode a = (- (*C*,*M*,*pc*)#*cs*,*x* -);
targetnode a = (- (*C'*,*M'*,*pc'*)#*cs'*,*x'* -);
pred (*kind a*) *s*;
$x \neq None \longrightarrow (\exists\ a\text{-}pred.$
sourcenode a-pred = (- (*C*,*M*,*pc*)#*cs*,*None* -) ∧
targetnode a-pred = *sourcenode a* ∧
valid-edge (*P*,*C0*,*Main*) *a-pred* ∧
pred (*kind a-pred*) *s*
);
$P,((C,M,pc)\#cs) \vdash_{BV} s\ \surd$ ⟧
$\Longrightarrow P,((C',M',pc')\#cs') \vdash_{BV} transfer\ (kind\ a)\ s\ \surd$
⟨*proof*⟩

7.2 CFG simulates Jinja's semantics

7.2.1 Definitions

The following predicate defines the semantics of Jinja lifted to our state representation. Thereby, we require the state to be byte code verifier conform; otherwise the step in the semantics is undefined.

The predicate *valid-callstack* is actually an implication of the byte code verifier conformance. But we list it explicitly for convenience.

inductive *sem* :: *jvmprog* ⇒ *callstack* ⇒ *state* ⇒ *callstack* ⇒ *state* ⇒ *bool*
(- ⊢ ⟨-,-⟩ ⇒ ⟨-,-⟩)
where *Step*:
⟦ *prog* = (*P*,*C0*,*Main*);
$P,cs \vdash_{BV} s\ \surd$;
valid-callstack prog cs;
$JVMExec.exec\ ((P_{wf}),\ state\text{-}to\text{-}jvm\text{-}state\ P\ cs\ s) = \lfloor(None,h',frs')\rfloor$;
cs' = *framestack-to-callstack frs'*;
s = (*h*, *stk*, *loc*);
s' = (*h'*, *update-stk stk frs'*, *update-loc loc frs'*) ⟧
$\Longrightarrow prog \vdash \langle cs,s\rangle \Rightarrow \langle cs',s'\rangle$

abbreviation *identifies* :: *j-node* ⇒ *callstack* ⇒ *bool*
where *identifies n cs* ≡ (*n* = (- *cs*,*None* -))

7.2.2 Some more simplification lemmas

lemma *valid-callstack-tl*:
valid-callstack prog ((*C*,*M*,*pc*)#*cs*) ⟹ *valid-callstack prog cs*
⟨*proof*⟩

lemma *stkss-cong* [*cong*]:
⟦ *P* = *P'*;
cs = *cs'*;
⋀*a b*. ⟦ *a* < *length cs*;
b < *stkLength P* (*fst*(*cs* ! (*length cs* − *Suc a*)))
(*fst*(*snd*(*cs* ! (*length cs* − *Suc a*))))
(*snd*(*snd*(*cs* ! (*length cs* − *Suc a*)))) ⟧
⟹ *stk* (*a*, *b*) = *stk'* (*a*, *b*) ⟧
⟹ *stkss P cs stk* = *stkss P' cs' stk'*
⟨*proof*⟩

lemma *locss-cong* [*cong*]:
⟦ *P* = *P'*;
cs = *cs'*;
⋀*a b*. ⟦ *a* < *length cs*;
b < *locLength P* (*fst*(*cs* ! (*length cs* − *Suc a*)))
(*fst*(*snd*(*cs* ! (*length cs* − *Suc a*))))
(*snd*(*snd*(*cs* ! (*length cs* − *Suc a*)))) ⟧
⟹ *loc* (*a*, *b*) = *loc'* (*a*, *b*) ⟧
⟹ *locss P cs loc* = *locss P' cs' loc'*
⟨*proof*⟩

lemma *hd-tl-equalityI*:
⟦ *length xs* = *length ys*; *hd xs* = *hd ys*; *tl xs* = *tl ys* ⟧ ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *stkLength-is-length-stk*:
P_{wf},P_{Φ} ⊢ (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs'*) √ ⟹ *stkLength P C M pc* = *length stk*
⟨*proof*⟩

lemma *locLength-is-length-loc*:
P_{wf},P_{Φ} ⊢ (*None*, *h*, (*stk*, *loc*, *C*, *M*, *pc*) # *frs'*) √ ⟹ *locLength P C M pc* = *length loc*
⟨*proof*⟩

lemma *correct-state-frs-tlD*:
(P_{wf}),(P_{Φ}) ⊢ (*None*, *h*, *a* # *frs'*) √ ⟹ (P_{wf}),(P_{Φ}) ⊢ (*None*, *h*, *frs'*) √
⟨*proof*⟩

lemma *update-stk-Cons* [*simp*]:
stkss P (*framestack-to-callstack frs'*) (*update-stk stk* ((*stk'*, *loc'*, *C'*, *M'*, *pc'*) # *frs'*)) =
stkss P (*framestack-to-callstack frs'*) (*update-stk stk frs'*)
⟨*proof*⟩

lemma *update-loc-Cons* [*simp*]:
locss P (*framestack-to-callstack frs'*) (*update-loc loc* ((*stk'*, *loc'*, *C'*, *M'*, *pc'*) # *frs'*)) =
locss P (*framestack-to-callstack frs'*) (*update-loc loc frs'*)
⟨*proof*⟩

lemma *s2j-id*:
(P_{wf}),$(P_\Phi) \vdash$ (*None*,*h'*,*frs'*) $\surd$
$\Longrightarrow$ *state-to-jvm-state P* (*framestack-to-callstack frs'*)
(*h*, *update-stk stk frs'*, *update-loc loc frs'*) = (*None*, *h*, *frs'*)
⟨*proof*⟩

lemma *find-handler-last-cs-eqD*:
⟦ *find-handler* P_{wf} *a h frs* = (*None*, *h'*, *frs'*);
last frs = (*stk*,*loc*,*C*,*M*,*pc*);
last frs' = (*stk'*,*loc'*,*C'*,*M'*,*pc'*) ⟧
$\Longrightarrow C = C' \wedge M = M'$
⟨*proof*⟩

lemma *exec-last-frs-eq-class*:
⟦ *JVMExec.exec* (P_{wf}, *None*, *h*, *frs*) = ⌊(*None*, *h'*, *frs'*)⌋;
last frs = (*stk*, *loc*, *C*, *M*, *pc*);
last frs' = (*stk'*, *loc'*, *C'*, *M'*, *pc'*);
frs $\neq$ [];
frs' $\neq$ [] ⟧
$\Longrightarrow C = C'$
⟨*proof*⟩

lemma *exec-last-frs-eq-method*:
⟦ *JVMExec.exec* (P_{wf}, *None*, *h*, *frs*) = ⌊(*None*, *h'*, *frs'*)⌋;
last frs = (*stk*, *loc*, *C*, *M*, *pc*);
last frs' = (*stk'*, *loc'*, *C'*, *M'*, *pc'*);
frs $\neq$ [];
frs' $\neq$ [] ⟧
$\Longrightarrow M = M'$
⟨*proof*⟩

lemma *valid-callstack-append-last-class*:
valid-callstack (*P*,*C0*,*Main*) (*cs*@[(*C*,*M*,*pc*)]) $\Longrightarrow C = C0$
⟨*proof*⟩

lemma *valid-callstack-append-last-method*:

valid-callstack (*P*,*C0*,*Main*) (*cs*@[(*C*,*M*,*pc*)]) $\Longrightarrow$ *M* = *Main*
⟨*proof*⟩

lemma *zip-stkss-locss-append-single* [*simp*]:
zip (*stkss P* (*cs* @ [(*C*, *M*, *pc*)]) *stk*)
(*zip* (*locss P* (*cs* @ [(*C*, *M*, *pc*)]) *loc*) (*cs* @ [(*C*, *M*, *pc*)]))
= (*zip* (*stkss P* (*cs* @ [(*C*, *M*, *pc*)]) *stk*) (*zip* (*locss P* (*cs* @ [(*C*, *M*, *pc*)]) *loc*) *cs*))
@ [(*stks* (*stkLength P C M pc*) (λ*a*. *stk* (*0*, *a*)),
locs (*locLength P C M pc*) (λ*a*. *loc* (*0*, *a*)), *C*, *M*, *pc*)]
⟨*proof*⟩

7.2.3 Interpretation of the *CFG-semantics-wf* locale

interpretation *JVM-semantics-CFG-wf*:
CFG-semantics-wf sourcenode targetnode kind valid-edge prog (*-Entry-*)
sem prog identifies
for *prog*
⟨*proof*⟩

end

theory *Slicing*
imports
Basic/Postdomination
Basic/CFGExit-wf
Basic/SemanticsCFG
Dynamic/DynSlice
StaticIntra/CDepInstantiations
StaticIntra/ControlDependenceRelations
While/DynamicControlDependences
While/NonInterferenceWhile
JinjaVM/JVMControlDependences
JinjaVM/SemanticsWF
begin

end

Bibliography

[1] Daniel Wasserrab and Denis Lohner and Gregor Snelting On PDG-Based Noninterference and its Modular Proof In *Proc. of PLAS'09*, pages 31–44. ACM, 2009.

[2] Daniel Wasserrab and Andreas Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in Isabelle/HOL. In *Proc. of TPHOLS'08*, pages 294–309. Springer-Verlag, 2008.

[3] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.