

Towards Certified Slicing

Daniel Wasserrab

December 12, 2009

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 $\exists x \in \text{set } xs. P x$

obtains $zs \ x' \ ys$ **where** $xs = zs @ x' \# ys$ **and** $P x'$ **and** $\forall z \in \text{set } zs. \neg P z$
(*proof*)

lemma *rightmost-element-property*:

assumes $\exists x \in \text{set } xs. P x$

obtains $ys \ x' \ zs$ **where** $xs = ys @ x' \# zs$ **and** $P x'$ **and** $\forall z \in \text{set } zs. \neg P z$
(*proof*)

Lemma concerning maps and @

lemma *map-append-append-maps*:
 assumes *map*: $\text{map } f \text{ } xs = ys @ zs$
 obtains $xs' \ xs''$ **where** $\text{map } f \text{ } xs' = ys$ **and** $\text{map } f \text{ } xs'' = zs$ **and** $xs = xs' @ xs''$
 $\langle \text{proof} \rangle$

Lemma concerning splitting of *lists*

lemma *path-split-general*:
 assumes *all*: $\forall zs. xs \neq ys @ zs$
 obtains $j \ zs$ **where** $xs = (\text{take } j \ ys) @ zs$ **and** $j < \text{length } ys$
 and $\forall k > j. \forall zs'. xs \neq (\text{take } k \ ys) @ zs'$
 $\langle \text{proof} \rangle$

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-kind = Update 'state ⇒ 'state          (↑-)
  | Predicate 'state ⇒ bool    (('(-)✓)
```

1.1.2 Transfer and predicate functions

```
fun transfer :: 'state edge-kind ⇒ 'state ⇒ 'state
where transfer (↑f) s = f s
  | transfer (P)✓ s = s
```

```
fun transfers :: 'state edge-kind list ⇒ 'state ⇒ 'state
where transfers [] s = s
  | transfers (e#es) s = transfers es (transfer e s)
```

```
fun pred :: 'state edge-kind ⇒ 'state ⇒ bool
where pred (↑f) s = True
  | pred (P)✓ s = (P s)
```

```
fun preds :: 'state edge-kind list ⇒ 'state ⇒ bool
```

where $\text{preds } [] \ s = \text{True}$
 $| \text{preds } (e\#es) \ s = (\text{pred } e \ s \wedge \text{preds } es \ (\text{transfer } e \ s))$

lemma *transfers-split*:
 $(\text{transfers } (ets@ets') \ s) = (\text{transfers } ets' \ (\text{transfers } ets \ s))$
 $\langle \text{proof} \rangle$

lemma *preds-split*:
 $(\text{preds } (ets@ets') \ s) = (\text{preds } ets \ s \wedge \text{preds } ets' \ (\text{transfers } ets \ s))$
 $\langle \text{proof} \rangle$

lemma *transfers-id-no-influence*:
 $\text{transfers } [et \leftarrow ets. et \neq \uparrow id] \ s = \text{transfers } ets \ s$
 $\langle \text{proof} \rangle$

lemma *preds-True-no-influence*:
 $\text{preds } [et \leftarrow ets. et \neq (\lambda s. \text{True})_{\surd}] \ s = \text{preds } ets \ s$
 $\langle \text{proof} \rangle$

end

1.2 CFG

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

1.2.1 The abstract CFG

locale *CFG* =
fixes *sourcenode* :: 'edge \Rightarrow 'node
fixes *targetnode* :: 'edge \Rightarrow 'node
fixes *kind* :: 'edge \Rightarrow 'state edge-kind
fixes *valid-edge* :: 'edge \Rightarrow bool
fixes *Entry*::'node ('('Entry'-'))
assumes *Entry-target* [*dest*]: $\llbracket \text{valid-edge } a; \text{targetnode } a = (\text{-Entry-}) \rrbracket \Longrightarrow \text{False}$
and *edge-det*:
 $\llbracket \text{valid-edge } a; \text{valid-edge } a'; \text{sourcenode } a = \text{sourcenode } a';$
 $\text{targetnode } a = \text{targetnode } a' \rrbracket \Longrightarrow a = a'$

begin

definition *valid-node* :: 'node \Rightarrow bool
where *valid-node* $n \equiv$
 $(\exists a. \text{valid-edge } a \wedge (n = \text{sourcenode } a \vee n = \text{targetnode } a))$

lemma [*simp*]: $\text{valid-edge } a \Longrightarrow \text{valid-node } (\text{sourcenode } a)$

$\langle proof \rangle$

lemma *[simp]: valid-edge a \implies valid-node (targetnode a)*
 $\langle proof \rangle$

1.2.2 CFG paths and lemmas

inductive *path :: 'node \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool*
(- \dashrightarrow^ - [51,0,0] 80)*

where

empty-path: valid-node n \implies n \dashrightarrow^ n*

| *Cons-path:*

$\llbracket n'' - as \rightarrow^* n'; \text{valid-edge } a; \text{sourcenode } a = n; \text{targetnode } a = n' \rrbracket$
 $\implies n - a \# as \rightarrow^* n'$

lemma *path-valid-node:*

assumes *n \dashrightarrow^* n'* **shows** *valid-node n and valid-node n'*
 $\langle proof \rangle$

lemma *empty-path-nodes [dest]: n \dashrightarrow^* n' \implies n = n'*
 $\langle proof \rangle$

lemma *path-valid-edges: n \dashrightarrow^* n' \implies $\forall a \in \text{set } as. \text{valid-edge } a$*
 $\langle proof \rangle$

lemma *path-edge: valid-edge a \implies sourcenode a \dashrightarrow^* targetnode a*
 $\langle proof \rangle$

lemma *path-Entry-target [dest]:*

assumes *n \dashrightarrow^* (-Entry-)*
shows *n = (-Entry-) and as = []*
 $\langle proof \rangle$

lemma *path-Append: $\llbracket n - as \rightarrow^* n''; n'' - as' \rightarrow^* n' \rrbracket$*
 $\implies n - as @ as' \rightarrow^* n'$
 $\langle proof \rangle$

lemma *path-split:*

assumes *n - as @ a # as' \rightarrow^* n'*
shows *n - as \rightarrow^* sourcenode a and valid-edge a and targetnode a \dashrightarrow^* n'*
 $\langle proof \rangle$

lemma *path-split-Cons*:

assumes $n - as \rightarrow^* n'$ **and** $as \neq []$

obtains $a' as'$ **where** $as = a' \# as'$ **and** $n = \text{sourcenode } a'$

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** $n - as' \rightarrow^* \text{sourcenode } a'$

and *valid-edge* a' **and** $n' = \text{targetnode } a'$

<proof>

lemma *path-split-second*:

assumes $n - as @ a \# as' \rightarrow^* n'$ **shows** $\text{sourcenode } a - a \# as' \rightarrow^* n'$

<proof>

lemma *path-Entry-Cons*:

assumes *(-Entry-)* $- as \rightarrow^* n'$ **and** $n' \neq \text{(-Entry-)}$

obtains $n a$ **where** $\text{sourcenode } a = \text{(-Entry-)}$ **and** $\text{targetnode } a = n$

and $n - tl as \rightarrow^* n'$ **and** *valid-edge* a **and** $a = \text{hd } as$

<proof>

lemma *path-det*:

$\llbracket n - as \rightarrow^* n'; n - as \rightarrow^* n'' \rrbracket \implies n' = n''$

<proof>

definition

sourcenodes :: 'edge list \Rightarrow 'node list

where *sourcenodes* $xs \equiv \text{map } \text{sourcenode } xs$

definition

kinds :: 'edge list \Rightarrow 'state edge-kind list

where *kinds* $xs \equiv \text{map } \text{kind } xs$

definition

targetnodes :: 'edge list \Rightarrow 'node list

where *targetnodes* $xs \equiv \text{map } \text{targetnode } xs$

lemma *path-sourcenode*:

$\llbracket n - as \rightarrow^* n'; as \neq [] \rrbracket \implies \text{hd } (\text{sourcenodes } as) = n$

<proof>

lemma *path-targetnode*:
 $\llbracket n - as \rightarrow * n'; as \neq [] \rrbracket \implies last (targetnodes\ as) = n'$
 $\langle proof \rangle$

lemma *sourcenodes-is-n-Cons-butlast-targetnodes*:
 $\llbracket n - as \rightarrow * n'; as \neq [] \rrbracket \implies$
 $sourcenodes\ as = n \# (butlast (targetnodes\ as))$
 $\langle proof \rangle$

lemma *targetnodes-is-tl-sourcenodes-App-n'*:
 $\llbracket n - as \rightarrow * n'; as \neq [] \rrbracket \implies$
 $targetnodes\ as = (tl (sourcenodes\ as))@[n']$
 $\langle proof \rangle$

lemma *Entry-sourcenode-hd*:
assumes $n - as \rightarrow * n'$ **and** $(-Entry-) \in set (sourcenodes\ as)$
shows $n = (-Entry-)$ **and** $(-Entry-) \notin set (sourcenodes (tl\ as))$
 $\langle proof \rangle$

end

end

1.3 CFG well-formedness

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

1.3.1 Well-formedness of the abstract CFG

locale *CFG-wf* = *CFG* *sourcenode* *targetnode* *kind* *valid-edge* *Entry*
for *sourcenode* :: 'edge \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow 'state *edge-kind* **and** *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node (('(-Entry'-')) +
fixes *Def*::'node \Rightarrow 'var set
fixes *Use*::'node \Rightarrow 'var set
fixes *state-val*::'state \Rightarrow 'var \Rightarrow 'val
assumes *Entry-empty*:*Def* (-Entry-) = {} \wedge *Use* (-Entry-) = {}
and *CFG-edge-no-Def-equal*:
 $\llbracket valid-edge\ a; V \notin Def (sourcenode\ a); pred (kind\ a)\ s \rrbracket$
 $\implies state-val (transfer (kind\ a)\ s)\ V = state-val\ s\ V$
and *CFG-edge-transfer-uses-only-Use*:
 $\llbracket valid-edge\ a; \forall V \in Use (sourcenode\ a). state-val\ s\ V = state-val\ s'\ V; \rrbracket$

$\text{pred } (\text{kind } a) \ s; \text{pred } (\text{kind } a) \ s'$
 $\implies \forall V \in \text{Def } (\text{sourcenode } a). \text{state-val } (\text{transfer } (\text{kind } a) \ s) \ V =$
 $\text{state-val } (\text{transfer } (\text{kind } a) \ s') \ V$
and *CFG-edge-Uses-pred-equal*:
 $\llbracket \text{valid-edge } a; \text{pred } (\text{kind } a) \ s;$
 $\forall V \in \text{Use } (\text{sourcenode } a). \text{state-val } s \ V = \text{state-val } s' \ V \rrbracket$
 $\implies \text{pred } (\text{kind } a) \ s'$
and *deterministic*: $\llbracket \text{valid-edge } a; \text{valid-edge } a'; \text{sourcenode } a = \text{sourcenode } a';$
 $\text{targetnode } a \neq \text{targetnode } a' \rrbracket$
 $\implies \exists Q \ Q'. \text{kind } a = (Q)_{\surd} \wedge \text{kind } a' = (Q')_{\surd} \wedge$
 $(\forall s. (Q \ s \longrightarrow \neg Q' \ s) \wedge (Q' \ s \longrightarrow \neg Q \ s))$

begin

lemma $[\text{dest!}]$: $V \in \text{Use } (-\text{Entry-}) \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma $[\text{dest!}]$: $V \in \text{Def } (-\text{Entry-}) \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *CFG-path-no-Def-equal*:
 $\llbracket n - \text{as} \rightarrow * \ n'; \forall n \in \text{set } (\text{sourcenodes } \text{as}). \ V \notin \text{Def } n; \text{preds } (\text{kinds } \text{as}) \ s \rrbracket$
 $\implies \text{state-val } (\text{transfers } (\text{kinds } \text{as}) \ s) \ V = \text{state-val } s \ V$
 $\langle \text{proof} \rangle$

end

end

1.4 Dynamic and static data dependence

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

1.4.1 Dynamic data dependence

context *CFG-wf* **begin**

definition *dyn-data-dependence* ::
 $\text{'node} \Rightarrow \text{'var} \Rightarrow \text{'node} \Rightarrow \text{'edge list} \Rightarrow \text{bool } (- \text{ influences } - \text{ in } - \text{ via } - [51,0,0])$
where $n \text{ influences } V \text{ in } n' \text{ via } \text{as} \equiv$
 $((V \in \text{Def } n) \wedge (V \in \text{Use } n') \wedge (n - \text{as} \rightarrow * \ n') \wedge$
 $(\exists a' \ \text{as}'. (\text{as} = a' \# \text{as}') \wedge (\forall n'' \in \text{set } (\text{sourcenodes } \text{as}'). \ V \notin \text{Def } n''))$

lemma *dyn-influence-Cons-source*:
 $n \text{ influences } V \text{ in } n' \text{ via } a \# \text{as} \implies \text{sourcenode } a = n$

<proof>

lemma *dyn-influence-source-notin-tl-edges:*

assumes n influences V in n' via $a\#as$

shows $n \notin \text{set}(\text{sourcenodes } as)$

<proof>

lemma *dyn-influence-only-first-edge:*

assumes n influences V in n' via $a\#as$ **and** $\text{preds}(\text{kinds}(a\#as))\ s$

shows $\text{state-val}(\text{transfers}(\text{kinds}(a\#as))\ s)\ V =$

$\text{state-val}(\text{transfer}(\text{kind } a)\ s)\ V$

<proof>

1.4.2 Static data dependence

definition *data-dependence* :: $'node \Rightarrow 'var \Rightarrow 'node \Rightarrow \text{bool}$

$(- \text{influences } - \text{ in } - [51, 0])$

where *data-dependences-eq:n influences V in n'* $\equiv \exists as. n \text{ influences } V \text{ in } n' \text{ via } as$

lemma *data-dependence-def: n influences V in n' =*

$(\exists a' as'. (V \in \text{Def } n) \wedge (V \in \text{Use } n') \wedge$

$(n - a'\#as' \rightarrow^* n') \wedge (\forall n'' \in \text{set}(\text{sourcenodes } as'). V \notin \text{Def } n''))$

<proof>

end

end

theory *CFGExit* imports *CFG* begin

1.4.3 Adds an exit node to the abstract CFG

locale *CFGExit* = *CFG* *sourcenode targetnode kind valid-edge Entry*

for *sourcenode* :: $'edge \Rightarrow 'node$ **and** *targetnode* :: $'edge \Rightarrow 'node$

and *kind* :: $'edge \Rightarrow 'state \text{ edge-kind}$ **and** *valid-edge* :: $'edge \Rightarrow \text{bool}$

and *Entry* :: $'node \text{ ('(-Entry-'))} +$

fixes *Exit*:: $'node \text{ ('(-Exit-'))}$

assumes *Exit-source* [*dest*]: $\llbracket \text{valid-edge } a; \text{sourcenode } a = (-\text{Exit-}) \rrbracket \implies \text{False}$

and *Entry-Exit-edge*: $\exists a. \text{valid-edge } a \wedge \text{sourcenode } a = (-\text{Entry-}) \wedge$

$\text{targetnode } a = (-\text{Exit-}) \wedge \text{kind } a = (\lambda s. \text{False})_{\checkmark}$

begin

lemma *Entry-noteq-Exit* [*dest*]:

assumes $\text{eq}:(-\text{Entry-}) = (-\text{Exit-})$ **shows** *False*

<proof>

lemma *Exit-noteq-Entry* [*dest*]: $(-Exit-) = (-Entry-) \implies False$
 ⟨*proof*⟩

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

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

definition *inner-node* :: *'node* $\implies bool$
where *inner-node-def*:
inner-node $n \equiv \text{valid-node } n \wedge n \neq (-Entry-) \wedge n \neq (-Exit-)$

lemma *inner-is-valid*:
inner-node $n \implies \text{valid-node } n$
 ⟨*proof*⟩

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

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

lemma [*simp*]: $[[\text{valid-edge } a; \text{targetnode } a \neq (-Exit-)]$
 $\implies \text{inner-node } (\text{targetnode } a)$
 ⟨*proof*⟩

lemma [*simp*]: $[[\text{valid-edge } a; \text{sourcenode } a \neq (-Entry-)]$
 $\implies \text{inner-node } (\text{sourcenode } a)$
 ⟨*proof*⟩

lemma *valid-node-cases* [*consumes 1, case-names Entry Exit inner*]:
 $[[\text{valid-node } n; n = (-Entry-) \implies Q; n = (-Exit-) \implies Q;$
 $\text{inner-node } n \implies Q]] \implies Q$
 ⟨*proof*⟩

lemma *path-Exit-source* [*dest*]:
assumes $(-Exit-) -as \rightarrow^* n'$ **shows** $n' = (-Exit-)$ **and** $as = []$
 ⟨*proof*⟩

lemma *Exit-no-sourcenode*[*dest*]:
assumes $isin:(-Exit-) \in \text{set } (\text{sourcenodes } as)$ **and** $\text{path}:n -as \rightarrow^* n'$
shows *False*
 ⟨*proof*⟩

end

end

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

1.4.4 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 \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow 'state edge-kind **and** *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node ('('Entry'-')) **and** *Def* :: 'node \Rightarrow 'var set
and *Use* :: 'node \Rightarrow 'var set **and** *state-val* :: 'state \Rightarrow 'var \Rightarrow 'val
and *Exit* :: 'node ('('Exit'-')) +
assumes *Exit-empty:Def* $(-Exit-) = \{\}$ \wedge *Use* $(-Exit-) = \{\}$

begin

lemma *Exit-Use-empty* [*dest!*]: $V \in Use (-Exit-) \Longrightarrow False$
(*proof*)

lemma *Exit-Def-empty* [*dest!*]: $V \in Def (-Exit-) \Longrightarrow False$
(*proof*)

end

end

1.5 PDG

theory *PDG* imports *DataDependence CFGExit-wf* begin

1.5.1 The dynamic PDG

locale *DynPDG* =

CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: 'edge \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow 'state edge-kind **and** *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node ('('Entry'-')) **and** *Def* :: 'node \Rightarrow 'var set
and *Use* :: 'node \Rightarrow 'var set **and** *state-val* :: 'state \Rightarrow 'var \Rightarrow 'val
and *Exit* :: 'node ('('Exit'-')) +
fixes *dyn-control-dependence* :: 'node \Rightarrow 'node \Rightarrow 'edge list \Rightarrow bool
(- *controls* - via - [51,0,0])
assumes *Exit-not-dyn-control-dependent:n controls n' via as* $\Longrightarrow n' \neq (-Exit-)$
assumes *dyn-control-dependence-path:*
 $n \text{ controls } n' \text{ via } as \Longrightarrow n - as \rightarrow^* n' \wedge as \neq []$

begin

inductive *cdep-edge* :: 'node \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool
(- \dashrightarrow_{cd} - [51,0,0] 80)
and *ddep-edge* :: 'node \Rightarrow 'var \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool
(- -'{'-}' \dashrightarrow_{dd} - [51,0,0,0] 80)
and *DynPDG-edge* :: 'node \Rightarrow 'var option \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool

where

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

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

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

inductive *DynPDG-path* :: 'node \Rightarrow 'edge list \Rightarrow 'node \Rightarrow bool
(- \dashrightarrow_{d*} - [51,0,0] 80)

where *DynPDG-path-Nil*:

$valid-node\ n \implies n -[] \rightarrow_{d*} n$

$| DynPDG-path-Append-cdep:$
 $\llbracket n -as \rightarrow_{d*} n''; n'' -as' \rightarrow_{cd} n' \rrbracket \implies n -as@as' \rightarrow_{d*} n'$

$| DynPDG-path-Append-ddep:$
 $\llbracket n -as \rightarrow_{d*} n''; n'' -\{V\}as' \rightarrow_{dd} n' \rrbracket \implies n -as@as' \rightarrow_{d*} n'$

lemma *DynPDG-empty-path-eq-nodes*: $n -[] \rightarrow_{d*} n' \implies n = n'$
(*proof*)

lemma *DynPDG-path-cdep*: $n -as \rightarrow_{cd} n' \implies n -as \rightarrow_{d*} n'$
(*proof*)

lemma *DynPDG-path-ddep*: $n -\{V\}as \rightarrow_{dd} n' \implies n -as \rightarrow_{d*} n'$
(*proof*)

lemma *DynPDG-path-Append*:

$\llbracket n'' -as' \rightarrow_{d*} n'; n -as \rightarrow_{d*} n'' \rrbracket \implies n -as@as' \rightarrow_{d*} n'$
(*proof*)

lemma *DynPDG-path-Exit*: $\llbracket n - as \rightarrow_{d^*} n'; n' = (-Exit) \rrbracket \implies n = (-Exit)$
 ⟨proof⟩

lemma *DynPDG-path-not-inner*:
 $\llbracket n - as \rightarrow_{d^*} n'; \neg \text{inner-node } n' \rrbracket \implies n = n'$
 ⟨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*:
 $n - as \rightarrow_{d^*} n' \implies n - as \rightarrow^* n'$
 ⟨proof⟩

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

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

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

$\llbracket n -\{V\}a\#as \rightarrow_{dd}\ n';\ preds\ (kinds\ (a\#as))\ s \rrbracket \implies$
 $state-val\ (transfers\ (kinds\ (a\#as))\ s)\ V = state-val\ (transfer\ (kind\ a)\ s)\ V$
<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** $state-val\ s\ V = state-val\ s'\ V$
and $state-val\ (transfers\ (kinds\ as)\ s)\ V \neq$
 $state-val\ (transfers\ (kinds\ as)\ s')\ V$
obtains $as'\ a\ as''$ **where** $as = as'@a\#as''$
and $sourcenode\ a -\{V\}a\#as'' \rightarrow_{dd}\ n'$
and $state-val\ (transfers\ (kinds\ as)\ s)\ V =$
 $state-val\ (transfers\ (kinds\ (as'@[a]))\ s)\ V$
and $state-val\ (transfers\ (kinds\ as)\ s')\ V =$
 $state-val\ (transfers\ (kinds\ (as'@[a]))\ s')\ V$
<proof>

end

1.5.2 The static PDG

locale *PDG* =

CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: 'edge \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow 'state edge-kind **and** *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node ('('Entry'-')) **and** *Def* :: 'node \Rightarrow 'var set
and *Use* :: 'node \Rightarrow 'var set **and** *state-val* :: 'state \Rightarrow 'var \Rightarrow 'val
and *Exit* :: 'node ('('Exit'-')) +
fixes *control-dependence* :: 'node \Rightarrow 'node \Rightarrow bool
(- *controls* - [51,0])
assumes *Exit-not-control-dependent*: $n\ controls\ n' \implies n' \neq (-Exit-)$
assumes *control-dependence-path*:
 $n\ controls\ n'$
 $\implies \exists as.\ CFG.path\ sourcenode\ targetnode\ valid-edge\ n\ as\ n' \wedge as \neq []$

begin

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

where

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

| *PDG-cdep-edge*:
 $n \text{ controls } n' \implies n \rightarrow_{cd} n'$

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

inductive *PDG-path* :: 'node ⇒ 'node ⇒ bool
(- →_{d*} - [51,0] 80)

where *PDG-path-Nil*:

$valid\text{-node } n \implies n \rightarrow_{d*} n$

| *PDG-path-Append-cdep*:
 $\llbracket n \rightarrow_{d*} n''; n'' \rightarrow_{cd} n' \rrbracket \implies n \rightarrow_{d*} n'$

| *PDG-path-Append-ddep*:
 $\llbracket n \rightarrow_{d*} n''; n'' -V \rightarrow_{dd} n' \rrbracket \implies n \rightarrow_{d*} n'$

lemma *PDG-path-cdep*: $n \rightarrow_{cd} n' \implies n \rightarrow_{d*} n'$
<proof>

lemma *PDG-path-ddep*: $n -V \rightarrow_{dd} n' \implies n \rightarrow_{d*} n'$
<proof>

lemma *PDG-path-Append*:
 $\llbracket n'' \rightarrow_{d*} n'; n \rightarrow_{d*} n'' \rrbracket \implies n \rightarrow_{d*} n'$
<proof>

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

assumes $n \rightarrow_{cd} n'$ **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** $as \neq []$
 $\langle proof \rangle$

lemma *PDG-path-CFG-path*:
assumes $n \rightarrow_{d^*} n'$ **obtains** *as* **where** $n - as \rightarrow^* n'$
 $\langle proof \rangle$

lemma *PDG-path-Exit*: $\llbracket n \rightarrow_{d^*} n'; n' = (-Exit-) \rrbracket \implies n = (-Exit-)$
 $\langle proof \rangle$

lemma *PDG-path-not-inner*:
 $\llbracket n \rightarrow_{d^*} n'; \neg inner\text{-node } n \rrbracket \implies n = n'$
 $\langle proof \rangle$

Definition of the static backward slice

definition *PDG-BS* :: $'node \Rightarrow 'node\ set$
where *PDG-BS* $n \equiv (if\ valid\text{-node } n\ then\ \{n'.\ n' \rightarrow_{d^*} n\}\ else\ \{\})$

lemma *PDG-BS-valid-node*: $n \in PDG\text{-BS } n_c \implies valid\text{-node } n$
 $\langle proof \rangle$

lemma *Exit-PDG-BS*: $n \in PDG\text{-BS } (-Exit-) \implies n = (-Exit-)$
 $\langle proof \rangle$

end

end

1.6 Postdomination

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

1.6.1 Standard Postdomination

locale *Postdomination* = *CFGExit* *sourcenode* *targetnode* *kind* *valid-edge* *Entry* *Exit*
for *sourcenode* :: $'edge \Rightarrow 'node$ **and** *targetnode* :: $'edge \Rightarrow 'node$
and *kind* :: $'edge \Rightarrow 'state\ edge\text{-kind}$ **and** *valid-edge* :: $'edge \Rightarrow bool$
and *Entry* :: $'node\ ('(-Entry\text{-}'))$ **and** *Exit* :: $'node\ ('(-Exit\text{-}'))$ +
assumes *Entry-path*: $valid\text{-node } n \implies \exists as.\ (-Entry-) - as \rightarrow^* n$

and *Exit-path:valid-node* $n \implies \exists as. n -as \rightarrow^* (-Exit-)$

begin

definition *postdominate* :: 'node \Rightarrow 'node \Rightarrow bool (- *postdominates* - [51,0])

where *postdominate-def*: n' *postdominates* $n \equiv$

(*valid-node* $n \wedge$ *valid-node* $n' \wedge$
(($\forall as. n -as \rightarrow^* (-Exit-) \longrightarrow n' \in set (sourcenodes\ as)$)))

lemma *postdominate-implies-path*:

assumes n' *postdominates* n **obtains** *as* **where** $n -as \rightarrow^* n'$
<proof>

lemma *postdominate-refl*:

assumes *valid:valid-node* n **and** *notExit:n* $\neq (-Exit-)$
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* $n \implies False$
<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

1.6.2 Strong Postdomination

locale *StrongPostdomination* =

Postdomination sourcenode targetnode kind *valid-edge* *Entry* *Exit*
for sourcenode :: 'edge \Rightarrow 'node **and** targetnode :: 'edge \Rightarrow 'node
and kind :: 'edge \Rightarrow 'state *edge-kind* **and** *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node (('(-Entry'-)) **and** *Exit* :: 'node (('(-Exit'-)) +
assumes *successor-set-finite*: *valid-node* *n* \implies
finite {*n'*. $\exists a'$. *valid-edge* *a'* \wedge sourcenode *a'* = *n* \wedge targetnode *a'* = *n'*}

begin

definition *strong-postdominate* :: 'node \Rightarrow 'node \Rightarrow bool

(- *strongly-postdominates* - [51,0])

where *strong-postdominate-def*:*n'* *strongly-postdominates* *n* \equiv

(*n'* *postdominates* *n* \wedge
($\exists k \geq 1$. $\forall as\ nx$. *n* -*as* \rightarrow^* *nx* \wedge
length *as* $\geq k \implies n' \in \text{set}(\text{sourcenodes } as)$))

lemma *strong-postdominate-prop-smaller-path:*

assumes $all:\forall as\ nx. n -as\rightarrow^* nx \wedge length\ as \geq k \longrightarrow n' \in set(sourcenodes\ as)$
and $n -as\rightarrow^* n''$ **and** $length\ as \geq k$
obtains $as'\ as''$ **where** $n -as'\rightarrow^* n'$ **and** $length\ as' < k$ **and** $n' -as''\rightarrow^* n''$
and $as = as'@as''$
 <proof>

lemma *strong-postdominate-refl*:
assumes *valid-node* n **and** $n \neq (-Exit-)$
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*:
 $\llbracket n' \text{ strongly-postdominates } n; n \text{ strongly-postdominates } n' \rrbracket \implies n = n'$
 <proof>

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

lemma *Exit-no-strong-postdominator*:
 $\llbracket (-Exit-) \text{ strongly-postdominates } n; n -as\rightarrow^* (-Exit-) \rrbracket \implies False$
 <proof>

lemma *strong-postdominate-path-targetnode*:
assumes n' *strongly-postdominates* n **and** $n -as\rightarrow^* n''$
and $n' \notin set(sourcenodes\ as)$
shows n' *strongly-postdominates* n''
 <proof>

lemma *not-strong-postdominate-successor-set*:
assumes $\neg n$ *strongly-postdominates* (*sourcenode* a) **and** *valid-node* n
and *valid-edge* a

and $\text{all}:\forall nx \in N. \exists a'. \text{valid-edge } a' \wedge \text{sourcenode } a' = \text{sourcenode } a \wedge$
 $\text{targetnode } a' = nx \wedge n \text{ strongly-postdominates } nx$
obtains a' **where** $\text{valid-edge } a'$ **and** $\text{sourcenode } a' = \text{sourcenode } a$
and $\text{targetnode } a' \notin N$
 ⟨*proof*⟩

lemma *not-strong-postdominate-predecessor-successor*:
assumes $\neg n \text{ strongly-postdominates } (\text{sourcenode } a)$
and $\text{valid-node } n$ **and** $\text{valid-edge } a$
obtains a' **where** $\text{valid-edge } a'$ **and** $\text{sourcenode } a' = \text{sourcenode } a$
and $\neg n \text{ strongly-postdominates } (\text{targetnode } a')$
 ⟨*proof*⟩

end

end

1.7 Standard control dependence

theory *StandardControlDependence* **imports** *PDG Postdomination* **begin**

1.7.1 Dynamic standard control dependence

Definition and some lemmas

context *Postdomination* **begin**

definition

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

lemma

Exit-not-dyn-standard-control-dependent:

assumes $\text{control}:n \text{ controls}_s (-\text{Exit-}) \text{ via } as$ **shows** *False*
 ⟨*proof*⟩

lemma

dyn-standard-control-dependence-def-variant:

$n \text{ controls}_s n' \text{ via } as = ((n -as \rightarrow^* n') \wedge (n \neq n') \wedge$

$(\neg n' \text{ postdominates } n) \wedge (n' \notin \text{set}(\text{sourcenodes } as)) \wedge$
 $(\forall n'' \in \text{set}(\text{targetnodes } as). n' \text{ postdominates } n'')$
 <proof>

lemma *which-node-dyn-standard-control-dependence-source:*

assumes *path*:(-Entry-) -as@a#a's'→* n
and *Exit-path*:n -as''→* (-Exit-) **and** *source*:sourcenode a = n'
and *source'*:sourcenode a' = n'
and *no-source*:n ∉ set(sourcenodes (a#a's'^)) **and** *valid-edge'*:valid-edge a'
and *inner-node*:inner-node n **and** *not-pd*:¬ n postdominates (targetnode a')
and *last*:∀ ax ax'. ax ∈ set as' ∧ sourcenode ax = sourcenode ax' ∧
 valid-edge ax' → n postdominates targetnode ax'
shows n' controls_s n via a#a's'
 <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

Instantiate dynamic PDG

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

1.7.2 Static standard control dependence

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' ≡ ∃ as. n controls_s n' via as

lemma *standard-control-dependence-def*: n controls_s n' =
(∃ a a' as. (n' ∉ set(sourcenodes (a#as))) ∧ (n -a#as→* n') ∧
(n' postdominates (targetnode a)) ∧
(valid-edge a') ∧ (sourcenode a' = n) ∧
(¬ n' postdominates (targetnode a')))

⟨proof⟩

lemma *Exit-not-standard-control-dependent*:

n controls_s (-Exit-) ⇒ False

⟨proof⟩

lemma *standard-control-dependence-def-variant*:

n controls_s n' = (∃ as. (n -as→* n') ∧ (n ≠ n') ∧
(¬ n' postdominates n) ∧ (n' ∉ set(sourcenodes as)) ∧
(∀ n'' ∈ set(targetnodes as). n' postdominates n''))

⟨proof⟩

lemma *inner-node-standard-control-dependence-predecessor*:

assumes inner-node n (-Entry-) -as→* n n -as'→* (-Exit-)

obtains n' where n' controls_s n

⟨proof⟩

end

Instantiate static PDG

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

end

1.8 Weak control dependence

theory *WeakControlDependence* **imports** *PDG Postdomination* **begin**

1.8.1 Dynamic standard control dependence

Definition and some lemmas

context *StrongPostdomination* **begin**

definition

dyn-weak-control-dependence :: 'node \Rightarrow 'node \Rightarrow 'edge list \Rightarrow bool
 (- weakly controls - via - [51,0,0])

where *dyn-weak-control-dependence-def*:n weakly controls n' via as \equiv
 $(\exists a\ a'\ as'. (as = a\#\ as') \wedge (n' \notin \text{set}(\text{sourcenodes}\ as)) \wedge (n -as \rightarrow^* n') \wedge$
 $(n' \text{strongly-postdominates}(\text{targetnode}\ a)) \wedge$
 $(\text{valid-edge}\ a') \wedge (\text{sourcenode}\ a' = n) \wedge$
 $(\neg n' \text{strongly-postdominates}(\text{targetnode}\ a'))))$

lemma *Exit-not-dyn-weak-control-dependent*:

assumes *control*:n weakly controls (-Exit-) via as **shows** *False*

<proof>

end

Instantiate dynamic PDG

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 \Rightarrow 'node **and** *targetnode* :: 'edge \Rightarrow 'node
and *kind* :: 'edge \Rightarrow 'state edge-kind **and** *valid-edge* :: 'edge \Rightarrow bool
and *Entry* :: 'node ('('Entry'-')) **and** *Def* :: 'node \Rightarrow 'var set
and *Use* :: 'node \Rightarrow 'var set **and** *state-val* :: 'state \Rightarrow 'var \Rightarrow '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

1.8.2 Static weak control dependence

Definition and some lemmas

context *StrongPostdomination* **begin**

definition

weak-control-dependence :: 'node ⇒ 'node ⇒ bool
(- *weakly controls* - [51,0])

where *weak-control-dependences-eq*:

n *weakly controls* n' ≡ ∃ *as*. n *weakly controls* n' *via as*

lemma

weak-control-dependence-def: n *weakly controls* n' =
(∃ a a' *as*. ($n' \notin \text{set}(\text{sourcenodes } (a\#as))$) ∧ ($n - a\#as \rightarrow^* n'$) ∧
(n' *strongly-postdominates* (*targetnode* a)) ∧
(*valid-edge* a') ∧ (*sourcenode* $a' = n$) ∧
(¬ n' *strongly-postdominates* (*targetnode* a')))

⟨*proof*⟩

lemma *Exit-not-weak-control-dependent*:

n *weakly controls* (-*Exit*-) ⇒ *False*

⟨*proof*⟩

end

Instantiate static PDG

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

1.9 Weak order dependence

theory *WeakOrderDependence* **imports** *CFG DataDependence* **begin**

Weak order dependence is just defined as a static control dependence

1.9.1 Definition and some lemmas

definition (in *CFG*) *weak-order-dependence* :: 'node \Rightarrow 'node \Rightarrow 'node \Rightarrow bool

($\cdot \longrightarrow_{\text{wod}} \cdot, \cdot$)

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

inductive-set (in *CFG-wf*) *wod-backward-slice* :: 'node \Rightarrow 'node set

for $n :: \text{'node}$

where *refl*: $\text{valid-node } n \implies n \in \text{wod-backward-slice } n$

| *cd-closed*:

$\llbracket n' \longrightarrow_{\text{wod}} n_1, n_2; n_1 \in \text{wod-backward-slice } n; n_2 \in \text{wod-backward-slice } n \rrbracket$
 $\implies n' \in \text{wod-backward-slice } n$

| *dd-closed*: $\llbracket n' \text{ influences } V \text{ in } n''; n'' \in \text{wod-backward-slice } n \rrbracket$

$\implies n' \in \text{wod-backward-slice } n$

lemma (in *CFG-wf*)

wod-backward-slice-valid-node: $n \in \text{wod-backward-slice } n_c \implies \text{valid-node } n$
 $\langle \text{proof} \rangle$

end

1.10 Relations between control dependences

theory *ControlDependenceRelations*

imports *WeakOrderDependence StandardControlDependence*

begin

context *StrongPostdomination* **begin**

```

lemma standard-control-implies-weak-order:
  assumes  $n \text{ controls}_s n'$  shows  $n \longrightarrow_{\text{wod}} n', (-\text{Exit-})$ 
   $\langle \text{proof} \rangle$ 

end

end

```

1.11 CFG and semantics conform

```

theory SemanticsCFG imports CFG begin

```

```

locale CFG-semantics-wf = CFG sourcenode targetnode kind valid-edge Entry
  for sourcenode :: 'edge  $\Rightarrow$  'node and targetnode :: 'edge  $\Rightarrow$  'node
  and kind :: 'edge  $\Rightarrow$  'state edge-kind and valid-edge :: 'edge  $\Rightarrow$  bool
  and Entry :: 'node ('('Entry'-')) +
  fixes sem::'com  $\Rightarrow$  'state  $\Rightarrow$  'com  $\Rightarrow$  'state  $\Rightarrow$  bool
    (((1<-,->)  $\Rightarrow$  / (1<-,->)) [0,0,0,0] 81)
  fixes identifies::'node  $\Rightarrow$  'com  $\Rightarrow$  bool (-  $\triangleq$  - [51,0] 80)
  assumes fundamental-property:
     $\llbracket n \triangleq c; \langle c, s \rangle \Rightarrow \langle c', s' \rangle \rrbracket \Longrightarrow$ 
     $\exists n' \text{ as. } n - \text{as} \rightarrow^* n' \wedge \text{transfers } (\text{kinds as}) s = s' \wedge \text{preds } (\text{kinds as}) s \wedge$ 
     $n' \triangleq c'$ 

```

```

end

```

Chapter 2

Dynamic Slicing

2.1 Dependent live variables

theory *DependentLiveVariables* **imports** *../Basic/PDG* **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 \Rightarrow ('var \times 'edge list \times 'edge list) set

for *n'* :: 'node

where *dep-vars-Use*:

$V \in \text{Use } n' \implies (V, [], []) \in \text{dependent-live-vars } n'$

| *dep-vars-Cons-cdep*:

$\llbracket V \in \text{Use } (\text{sourcenode } a); \text{sourcenode } a \xrightarrow{a\#as'}_{cd} n''; n'' \xrightarrow{as''}_{d*} n' \rrbracket$
 $\implies (V, [], a\#as'@as'') \in \text{dependent-live-vars } n'$

| *dep-vars-Cons-ddep*:

$\llbracket (V, as', as) \in \text{dependent-live-vars } n'; V' \in \text{Use } (\text{sourcenode } a);$
 $n' = \text{last}(\text{targetnodes } (a\#as));$
 $\text{sourcenode } a \xrightarrow{\{V\}a\#as'}_{dd} \text{last}(\text{targetnodes } (a\#as')) \rrbracket$
 $\implies (V', [], a\#as) \in \text{dependent-live-vars } n'$

| *dep-vars-Cons-keep*:

$\llbracket (V, as', as) \in \text{dependent-live-vars } n'; n' = \text{last}(\text{targetnodes } (a\#as));$
 $\neg \text{sourcenode } a \xrightarrow{\{V\}a\#as'}_{dd} \text{last}(\text{targetnodes } (a\#as')) \rrbracket$
 $\implies (V, a\#as', a\#as) \in \text{dependent-live-vars } n'$

lemma *dependent-live-vars-fst-prefix-snd*:

$(V, as', as) \in \text{dependent-live-vars } n' \implies \exists as''. as'@as'' = as$
(*proof*)

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

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

lemma *dependent-live-vars-Use-cases*:
 $\llbracket (V, as', as) \in \text{dependent-live-vars } n'; n - as \rightarrow^* n' \rrbracket$
 $\implies \exists nx as''. as = as' @ as'' \wedge n - as' \rightarrow^* nx \wedge nx - as'' \rightarrow_{d^*} n' \wedge V \in \text{Use } nx$
 \wedge
 $(\forall n'' \in \text{set } (\text{sourcenodes } as'). V \notin \text{Def } n'')$
 ⟨*proof*⟩

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

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

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

lemma *dependent-live-vars-same-paths*:
 $as \neq [] \longrightarrow n' = \text{last}(\text{targetnodes } as) \implies$
 $(V, as, as) \in \text{dependent-live-vars } n' =$

$(V \in Use\ n' \wedge (\forall as'\ a\ as''. as = as'@a\#as'' \longrightarrow$
 $\neg sourcenode\ a - \{V\}a\#as'' \rightarrow_{dd}\ n'))$
 <proof>

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

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

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

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

lemma *dependent-live-vars-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*:
 $\llbracket n'' - as'' \rightarrow_{d*}\ n'; (V', as', as) \in dependent-live-vars\ n'' \rrbracket$
 $\implies (V', as', as@as'') \in dependent-live-vars\ n'$
 <proof>

end

end

2.2 Formalisation of bit vectors

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

types *bit-vector* = *bool list*

fun *bv-leqs* :: *bit-vector* \Rightarrow *bit-vector* \Rightarrow *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*

2.2.1 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* = (($\forall i < \text{length } xs. xs ! i \longrightarrow ys ! i$) \wedge *length xs* = *length ys*)
<proof>

2.2.2 \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*: $\llbracket xs \preceq_b ys; ys \preceq_b zs \rrbracket \implies xs \preceq_b zs$
 $\langle proof \rangle$

lemma *bv-leqs-antisym*: $\llbracket xs \preceq_b ys; ys \preceq_b xs \rrbracket \implies xs = ys$
 $\langle proof \rangle$

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

interpretation *order bv-leqs bv-less*
 $\langle proof \rangle$

end

2.3 Dynamic backward slice

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

2.3.1 Backward slice of paths

context *DynPDG* **begin**

fun *slice-path* :: *'edge list* \Rightarrow *bit-vector*
where *slice-path* [] = []
| *slice-path* ($a\#as$) = ($let\ n' = last(targetnodes\ (a\#as))\ in$
 $(sourcnode\ a - a\#as \rightarrow_d^* n')\#slice-path\ as$)

lemma *slice-path-length*:
 $length(slice-path\ as) = length\ as$
 $\langle proof \rangle$

lemma *slice-path-right-Cons*:
assumes $slice: slice-path\ as = x\#xs$
obtains $a'\ as'$ **where** $as = a'\#as'$ **and** $slice-path\ as' = xs$
 $\langle proof \rangle$

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

fun *select-edge-kinds* :: *'edge list* \Rightarrow *bit-vector* \Rightarrow *'state edge-kind list*
where *select-edge-kinds* [] [] = []

| $select-edge-kinds (a\#as) (b\#bs) = (if\ b\ then\ kind\ a$
 $\quad\quad\quad else\ (case\ kind\ a\ of\ \uparrow f \Rightarrow \uparrow id \mid (Q)_{\surd} \Rightarrow (\lambda s. True)_{\surd}))\#select-edge-kinds\ as$
 bs

definition $slice-kinds :: 'edge\ list \Rightarrow 'state\ edge-kind\ list$
where $slice-kinds\ as = select-edge-kinds\ as\ (slice-path\ as)$

lemma $select-edge-kinds-max-bv:$
 $select-edge-kinds\ as\ (replicate\ (length\ as)\ True) = kinds\ as$
 $\langle proof \rangle$

lemma $slice-path-legs-information-same-Uses:$
 $\llbracket n - as \rightarrow^* n'; bs \preceq_b bs'; slice-path\ as = bs;$
 $select-edge-kinds\ as\ bs = es; select-edge-kinds\ as\ bs' = es';$
 $\forall V\ xs. (V, xs, as) \in dependent-live-vars\ n' \longrightarrow state-val\ s\ V = state-val\ s'\ V;$
 $\quad\quad\quad \llbracket preds\ es'\ s' \rrbracket$
 $\implies (\forall V \in Use\ n'. state-val\ (transfers\ es\ s)\ V =$
 $\quad\quad\quad state-val\ (transfers\ es'\ s')\ V) \wedge preds\ es\ s$
 $\langle proof \rangle$

theorem $fundamental-property-of-path-slicing:$
assumes $n - as \rightarrow^* n'$ **and** $preds\ (kinds\ as)\ s$
shows $(\forall V \in Use\ n'. state-val\ (transfers\ (slice-kinds\ as)\ s)\ V =$
 $\quad\quad\quad state-val\ (transfers\ (kinds\ as)\ s)\ V)$
and $preds\ (slice-kinds\ as)\ s$
 $\langle proof \rangle$

end

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

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

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

begin

theorem *fundamental-property-of-path-slicing-semantically*:
assumes $n \triangleq c$ **and** $\langle c, s \rangle \Rightarrow \langle c', s' \rangle$
obtains n' **as where** $n - as \rightarrow^* n'$ **and** *preds (slice-kinds as) s*
and $n' \triangleq c'$
and $\forall V \in Use\ n'.\ state-val\ (transfers\ (slice-kinds\ as)\ s)\ V =$
state-val s' V

$\langle proof \rangle$

end

end

2.4 Observable Sets of Nodes

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

context CFG **begin**

inductive-set *obs* :: 'node \Rightarrow 'node set \Rightarrow 'node set

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

where *obs-elem*:

$\llbracket n - as \rightarrow^* n'; \forall nx \in set(sourcenodes\ as). nx \notin S; n' \in S \rrbracket \implies n' \in obs\ n\ S$

lemma *obsE*:

assumes $n' \in obs\ n\ S$

obtains *as* **where** $n - as \rightarrow^* n'$ **and** $\forall nx \in set(sourcenodes\ as). nx \notin S$

and $n' \in S$

$\langle proof \rangle$

lemma *n-in-obs*:

assumes *valid-node n* **and** $n \in S$ **shows** $obs\ n\ S = \{n\}$

$\langle proof \rangle$

lemma *in-obs-valid*:

assumes $n' \in obs\ n\ S$ **shows** *valid-node n* **and** *valid-node n'*

$\langle proof \rangle$

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*:

$\llbracket n -as \rightarrow^* n'; \forall n' \in set(sourcenodes\ as). n' \notin S \rrbracket$
 $\implies obs\ n'\ S \subseteq obs\ n\ S$
<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 has to fulfill.

3.1 Distance of Paths

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

```
context CFG begin
```

```
inductive distance :: 'node  $\Rightarrow$  'node  $\Rightarrow$  nat  $\Rightarrow$  bool
```

```
where distanceI:
```

```
   $\llbracket n - as \rightarrow^* n'; \text{length } as = x; \forall as'. n - as' \rightarrow^* n' \longrightarrow x \leq \text{length } as \rrbracket$   
   $\implies \text{distance } n \ n' \ x$ 
```

```
lemma every-path-distance:
```

```
  assumes  $n - as \rightarrow^* n'$ 
```

obtains x **where** $\text{distance } n \ n' \ x$ **and** $x \leq \text{length } a$
 ⟨proof⟩

lemma *distance-det*:
 $\llbracket \text{distance } n \ n' \ x; \text{distance } n \ n' \ x' \rrbracket \implies x = x'$
 ⟨proof⟩

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

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

end

end

3.2 Static backward slice

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

begin

locale *BackwardSlice* =
 $\text{CFG-wf } \text{sourcenode } \text{targetnode } \text{kind } \text{valid-edge } \text{Entry } \text{Def } \text{Use } \text{state-val}$
for $\text{sourcenode} :: 'edge \Rightarrow 'node$ **and** $\text{targetnode} :: 'edge \Rightarrow 'node$
and $\text{kind} :: 'edge \Rightarrow 'state \text{ edge-kind}$ **and** $\text{valid-edge} :: 'edge \Rightarrow \text{bool}$
and $\text{Entry} :: 'node \Rightarrow ('Entry')$ **and** $\text{Def} :: 'node \Rightarrow 'var \text{ set}$
and $\text{Use} :: 'node \Rightarrow 'var \text{ set}$ **and** $\text{state-val} :: 'state \Rightarrow 'var \Rightarrow 'val +$
fixes $\text{backward-slice} :: 'node \Rightarrow 'node \text{ set}$

assumes *valid-nodes*: $n \in \text{backward-slice } n_c \implies \text{valid-node } n$
and *refl*: *valid-node* $n \implies n \in \text{backward-slice } n$
and *dd-closed*: $\llbracket n' \in \text{backward-slice } n_c; n \text{ influences } V \text{ in } n' \rrbracket$
 $\implies n \in \text{backward-slice } n_c$
and *obs-finite*: *valid-node* $n \implies \text{finite } (\text{obs } n (\text{backward-slice } n_c))$
and *obs-singleton*: *valid-node* n
 $\implies \text{card } (\text{obs } n (\text{backward-slice } n_c)) \leq 1$

begin

lemma *slice-n-in-obs*:

$n \in \text{backward-slice } n_c \implies \text{obs } n (\text{backward-slice } n_c) = \{n\}$
 $\langle \text{proof} \rangle$

lemma *assumes valid-node n*

shows *obs-singleton-disj*:

$(\exists m. \text{obs } n (\text{backward-slice } n_c) = \{m\}) \vee \text{obs } n (\text{backward-slice } n_c) = \{\}$
 $\langle \text{proof} \rangle$

lemma *obs-singleton-element*:

$m \in \text{obs } n (\text{backward-slice } n_c) \implies \text{obs } n (\text{backward-slice } n_c) = \{m\}$
 $\langle \text{proof} \rangle$

lemma *obs-the-element*:

$m \in \text{obs } n (\text{backward-slice } n_c) \implies (\text{THE } m. m \in \text{obs } n (\text{backward-slice } n_c)) =$
 m
 $\langle \text{proof} \rangle$

3.2.1 Traversing the sliced graph

slice-kind n' a conforms to *kind* a in the sliced graph

definition *slice-kind* :: 'node \Rightarrow 'edge \Rightarrow 'state edge-kind

where *slice-kind* n' $a =$ (let $S = \text{backward-slice } n'$; $n = \text{sourcenode } a$ in
 (if $\text{sourcenode } a \in S$ then *kind* a
 else (case *kind* a of $\uparrow f \Rightarrow \uparrow \text{id} \mid (Q)_{\surd} \Rightarrow$
 (if $\text{obs } (\text{sourcenode } a) S = \{\}$ then
 (let $nx = (\text{SOME } n'. \exists a'. n = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge \text{targetnode } a' = n')$
 in (if $(\text{targetnode } a = nx)$ then $(\lambda s. \text{True})_{\surd}$ else $(\lambda s. \text{False})_{\surd}$))
 else (let $m = \text{THE } m. m \in \text{obs } n S$ in
 (if $(\exists x. \text{distance } (\text{targetnode } a) m x \wedge \text{distance } n m (x + 1) \wedge$
 $(\text{targetnode } a = (\text{SOME } nx'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{distance } (\text{targetnode } a') m x \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = nx'))$
 then $(\lambda s. \text{True})_{\surd}$ else $(\lambda s. \text{False})_{\surd}$
))
))

))

definition

$\text{slice-kinds} :: 'node \Rightarrow 'edge \text{ list} \Rightarrow 'state \text{ edge-kind list}$
where $\text{slice-kinds } n' \text{ as} \equiv \text{map } (\text{slice-kind } n') \text{ as}$

lemma slice-kind-in-slice:

$\text{sourcenode } a \in \text{backward-slice } n' \implies \text{slice-kind } n' a = \text{kind } a$
(proof)

lemma slice-kind-Upd:

$\llbracket \text{sourcenode } a \notin \text{backward-slice } n'; \text{kind } a = \uparrow f \rrbracket \implies \text{slice-kind } n' a = \uparrow id$
(proof)

lemma slice-kind-Pred-empty-obs-SOME:

$\llbracket \text{sourcenode } a \notin \text{backward-slice } n'; \text{kind } a = (Q)_{\checkmark};$
 $\text{obs } (\text{sourcenode } a) (\text{backward-slice } n') = \{\};$
 $\text{targetnode } a = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a'$
 \wedge
 $\text{targetnode } a' = n') \rrbracket$
 $\implies \text{slice-kind } n' a = (\lambda s. \text{True})_{\checkmark}$
(proof)

lemma slice-kind-Pred-empty-obs-not-SOME:

$\llbracket \text{sourcenode } a \notin \text{backward-slice } n'; \text{kind } a = (Q)_{\checkmark};$
 $\text{obs } (\text{sourcenode } a) (\text{backward-slice } n') = \{\};$
 $\text{targetnode } a \neq (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a'$
 \wedge
 $\text{targetnode } a' = n') \rrbracket$
 $\implies \text{slice-kind } n' a = (\lambda s. \text{False})_{\checkmark}$
(proof)

lemma slice-kind-Pred-obs-nearer-SOME:

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

lemma slice-kind-Pred-obs-nearer-not-SOME:

assumes $\text{sourcenode } a \notin \text{backward-slice } n'$ **and** $\text{kind } a = (Q)_{\checkmark}$
and $m \in \text{obs } (\text{sourcenode } a) (\text{backward-slice } n')$
and $\text{distance } (\text{targetnode } a) m x \text{ distance } (\text{sourcenode } a) m (x + 1)$
and $\text{targetnode } a \neq (\text{SOME } nx'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{distance } (\text{targetnode } a') m x \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = nx')$
shows $\text{slice-kind } n' a = (\lambda s. \text{False})_{\checkmark}$
 $\langle \text{proof} \rangle$

lemma *slice-kind-Pred-obs-not-nearer:*

assumes $\text{sourcenode } a \notin \text{backward-slice } n'$ **and** $\text{kind } a = (Q)_{\checkmark}$
and $\text{in-obs}: m \in \text{obs } (\text{sourcenode } a) (\text{backward-slice } n')$
and $\text{dist}: \text{distance } (\text{sourcenode } a) m (x + 1)$
 $\quad \neg \text{distance } (\text{targetnode } a) m x$
shows $\text{slice-kind } n' a = (\lambda s. \text{False})_{\checkmark}$
 $\langle \text{proof} \rangle$

lemma *kind-Predicate-notin-slice-slice-kind-Predicate:*

assumes $\text{kind } a = (Q)_{\checkmark}$ **and** $\text{sourcenode } a \notin \text{backward-slice } n'$
obtains Q' **where** $\text{slice-kind } n' a = (Q')_{\checkmark}$ **and** $Q' = (\lambda s. \text{False}) \vee Q' = (\lambda s.$
 $\text{True})$
 $\langle \text{proof} \rangle$

lemma *only-one-SOME-edge:*

assumes $\text{valid-edge } a$
shows $\exists! a'. \text{sourcenode } a = \text{sourcenode } a' \wedge \text{valid-edge } a' \wedge$
 $\text{targetnode } a' = (\text{SOME } n'. \exists a'. \text{sourcenode } a = \text{sourcenode } a' \wedge$
 $\text{valid-edge } a' \wedge \text{targetnode } a' = n')$
 $\langle \text{proof} \rangle$

lemma *slice-kind-only-one-True-edge:*

assumes $\text{sourcenode } a = \text{sourcenode } a'$ **and** $\text{targetnode } a \neq \text{targetnode } a'$
and $\text{valid-edge } a$ **and** $\text{valid-edge } a'$ **and** $\text{slice-kind } n' a = (\lambda s. \text{True})_{\checkmark}$
shows $\text{slice-kind } n' a' = (\lambda s. \text{False})_{\checkmark}$
 $\langle \text{proof} \rangle$

lemma *slice-deterministic:*

assumes $\text{valid-edge } a$ **and** $\text{valid-edge } a'$
and $\text{sourcenode } a = \text{sourcenode } a'$ **and** $\text{targetnode } a \neq \text{targetnode } a'$
obtains $Q Q'$ **where** $\text{slice-kind } n' a = (Q)_{\checkmark}$ **and** $\text{slice-kind } n' a' = (Q')_{\checkmark}$
and $\forall s. (Q s \longrightarrow \neg Q' s) \wedge (Q' s \longrightarrow \neg Q s)$
 $\langle \text{proof} \rangle$

3.2.2 Observable and silent moves

inductive *silent-move* ::

'node \Rightarrow ('edge \Rightarrow 'state edge-kind) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge \Rightarrow 'node \Rightarrow 'state
 \Rightarrow bool
 ($\cdot, \cdot \vdash '(\cdot, \cdot) \dashrightarrow_{\tau} '(\cdot, \cdot)$ [51,50,0,0,50,0,0] 51)

where *silent-moveI*:

[[pred (f a) s; transfer (f a) s = s'; sourcenode a \notin backward-slice n_c ;
 valid-edge a]]
 $\Rightarrow n_{c,f} \vdash (\text{sourcenode } a, s) -a \rightarrow_{\tau} (\text{targetnode } a, s')$

inductive *silent-moves* ::

'node \Rightarrow ('edge \Rightarrow 'state edge-kind) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge list \Rightarrow 'node \Rightarrow
 'state \Rightarrow bool
 ($\cdot, \cdot \vdash '(\cdot, \cdot) \dashrightarrow_{\tau} '(\cdot, \cdot)$ [51,50,0,0,50,0,0] 51)

where *silent-moves-Nil*: $n_{c,f} \vdash (n, s) = [] \Rightarrow_{\tau} (n, s)$

| *silent-moves-Cons*:

[[$n_{c,f} \vdash (n, s) -a \rightarrow_{\tau} (n', s')$; $n_{c,f} \vdash (n', s') = as \Rightarrow_{\tau} (n'', s'')$]]
 $\Rightarrow n_{c,f} \vdash (n, s) = a \# as \Rightarrow_{\tau} (n'', s'')$

lemma *silent-moves-obs-slice*:

[[$n_{c,f} \vdash (n, s) = as \Rightarrow_{\tau} (n', s')$; $nx \in \text{obs } n'$ (backward-slice n_c)]]
 $\Rightarrow nx \in \text{obs } n$ (backward-slice n_c)
 <proof>

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

[[$n_{c,f} \vdash (n, s) = as \Rightarrow_{\tau} (n', s')$; valid-node n]]
 $\Rightarrow \text{preds } (\text{map } f \text{ as}) s \wedge \text{transfers } (\text{map } f \text{ as}) s = s' \wedge n -as \rightarrow^* n'$
 <proof>

lemma *obs-silent-moves*:

assumes $\text{obs } n$ (backward-slice n_c) = { n' }
obtains as **where** $n_{c,\text{slice-kind } n_c} \vdash (n, s) = as \Rightarrow_{\tau} (n', s)$
 <proof>

inductive *observable-move* ::

'node \Rightarrow ('edge \Rightarrow 'state edge-kind) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge \Rightarrow 'node \Rightarrow 'state
 \Rightarrow bool
 ($\cdot, \cdot \vdash '(\cdot, \cdot) \dashrightarrow_{\tau} '(\cdot, \cdot)$ [51,50,0,0,50,0,0] 51)

where *observable-moveI*:

[[pred (f a) s; transfer (f a) s = s'; sourcenode a \in backward-slice n_c ;

$\text{valid-edge } a \llbracket$
 $\implies n_c, f \vdash (\text{sourcenode } a, s) -a \rightarrow (\text{targetnode } a, s')$

inductive *observable-moves* ::

$'node \Rightarrow ('edge \Rightarrow 'state \text{ edge-kind}) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge \text{ list} \Rightarrow 'node \Rightarrow$
 $'state \Rightarrow \text{bool}$
 $(-, - \vdash '(-, -) \implies '(-, -) [51, 50, 0, 0, 50, 0, 0] 51)$

where *observable-moves-snoc*:

$\llbracket n_c, f \vdash (n, s) = as \Rightarrow_\tau (n', s'); n_c, f \vdash (n', s') -a \rightarrow (n'', s'') \rrbracket$
 $\implies n_c, f \vdash (n, s) = as @ [a] \Rightarrow (n'', s'')$

lemma *observable-move-notempty*:

$\llbracket n_c, f \vdash (n, s) = as \Rightarrow (n', s'); as = [] \rrbracket \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *silent-move-observable-moves*:

$\llbracket n_c, f \vdash (n'', s'') = as \Rightarrow (n', s'); n_c, f \vdash (n, s) -a \rightarrow_\tau (n'', s'') \rrbracket$
 $\implies n_c, f \vdash (n, s) = a \# as \Rightarrow (n', s')$
 $\langle \text{proof} \rangle$

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

$n_c, f \vdash (n, s) = as \Rightarrow (n', s')$
 $\implies \text{preds } (\text{map } f \text{ as}) \text{ } s \wedge \text{transfers } (\text{map } f \text{ as}) \text{ } s = s' \wedge n -as \rightarrow^* n'$
 $\langle \text{proof} \rangle$

3.2.3 Relevant variables

inductive-set *relevant-vars* :: $'node \Rightarrow 'node \Rightarrow 'var \text{ set } (rv -)$

for $n_c :: 'node$ **and** $n :: 'node$

where *rvI*:

$\llbracket n -as \rightarrow^* n'; n' \in \text{backward-slice } n_c; V \in \text{Use } n';$
 $\forall nx \in \text{set}(\text{sourcenodes } as). V \notin \text{Def } nx \rrbracket$
 $\implies V \in rv \text{ } n_c \text{ } n$

lemma *rvE*:

assumes *rv*: $V \in rv \text{ } n_c \text{ } n$

obtains *as* n' **where** $n -as \rightarrow^* n'$ **and** $n' \in \text{backward-slice } n_c$ **and** $V \in \text{Use } n'$

and $\forall nx \in \text{set}(\text{sourcenodes } as). V \notin \text{Def } nx$

$\langle \text{proof} \rangle$

lemma *eq-obs-in-rv*:

assumes *obs-eq:obs* n (*backward-slice* n_c) = *obs* n' (*backward-slice* n_c)

and $x \in rv\ n_c\ n$ **shows** $x \in rv\ n_c\ n'$

<proof>

lemma *closed-eq-obs-eq-rvs*:

fixes $n_c :: 'node$

assumes *valid-node* n **and** *valid-node* n'

and *obs-eq:obs* n (*backward-slice* n_c) = *obs* n' (*backward-slice* n_c)

shows $rv\ n_c\ n = rv\ n_c\ n'$

<proof>

lemma *rv-edge-slice-kinds*:

assumes *valid-edge* a **and** *sourcenode* $a = n$ **and** *targetnode* $a = n''$

and $\forall V \in rv\ n'\ n. state-val\ s\ V = state-val\ s'\ V$

and *preds* (*slice-kinds* n' ($a\#\#as$)) s **and** *preds* (*slice-kinds* n' ($a\#\#asx$)) s'

shows $\forall V \in rv\ n'\ n''. state-val\ (transfer\ (slice-kind\ n'\ a)\ s)\ V =$

$state-val\ (transfer\ (slice-kind\ n'\ a)\ s')\ V$

<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-kinds* n' ($a\#\#as$)) s **and** *preds* (*slice-kinds* n' ($ax\#\#asx$)) s'

and $\forall V \in rv\ n'\ n. state-val\ s\ V = state-val\ s'\ V$

shows *False*

<proof>

3.2.4 The set WS

inductive-set $WS :: 'node \Rightarrow (('node \times 'state) \times ('node \times 'state))\ set$

for $n_c :: 'node$

where $WSI: \llbracket obs\ n\ (backward-slice\ n_c) = obs\ n'\ (backward-slice\ n_c);$

$\forall V \in rv\ n_c\ n. state-val\ s\ V = state-val\ s'\ V;$

$valid-node\ n; valid-node\ n' \rrbracket$

$\implies ((n,s),(n',s')) \in WS\ n_c$

lemma *WSD*:

$((n,s),(n',s')) \in WS\ n_c$

$\implies obs\ n\ (backward-slice\ n_c) = obs\ n'\ (backward-slice\ n_c) \wedge$

$(\forall V \in rv\ n_c\ n. state-val\ s\ V = state-val\ s'\ V) \wedge$

$valid-node\ n \wedge valid-node\ n'$

<proof>

lemma *WS-silent-move*:

assumes $((n_1, s_1), (n_2, s_2)) \in WS\ n_c$ **and** $n_c, kind \vdash (n_1, s_1) - a \rightarrow_\tau (n_1', s_1')$
and $obs\ n_1'$ (*backward-slice* n_c) $\neq \{\}$ **shows** $((n_1', s_1'), (n_2, s_2)) \in WS\ n_c$
 $\langle proof \rangle$

lemma *WS-silent-moves*:

$\llbracket n_c, f \vdash (n_1, s_1) = as \Rightarrow_\tau (n_1', s_1'); ((n_1, s_1), (n_2, s_2)) \in WS\ n_c; f = kind;$
 $obs\ n_1'$ (*backward-slice* n_c) $\neq \{\}$ \rrbracket
 $\Rightarrow ((n_1', s_1'), (n_2, s_2)) \in WS\ n_c$
 $\langle proof \rangle$

lemma *WS-observable-move*:

assumes $((n_1, s_1), (n_2, s_2)) \in WS\ n_c$ **and** $n_c, kind \vdash (n_1, s_1) - a \rightarrow (n_1', s_1')$
obtains *as* **where** $((n_1', s_1'), (n_1', transfer\ (slice-kind\ n_c\ a)\ s_2)) \in WS\ n_c$
and $n_c, slice-kind\ n_c \vdash (n_2, s_2) = as@[a] \Rightarrow (n_1', transfer\ (slice-kind\ n_c\ a)\ s_2)$
 $\langle proof \rangle$

definition *is-weak-sim* :: $((node \times state) \times (node \times state))\ set \Rightarrow node \Rightarrow bool$
where *is-weak-sim* $R\ n_c \equiv$
 $\forall n_1\ s_1\ n_2\ s_2\ n_1'\ s_1'\ as. ((n_1, s_1), (n_2, s_2)) \in R \wedge n_c, kind \vdash (n_1, s_1) = as \Rightarrow (n_1', s_1')$
 $\longrightarrow (\exists n_2'\ s_2'\ as'. ((n_1', s_1'), (n_2', s_2')) \in R \wedge$
 $n_c, slice-kind\ n_c \vdash (n_2, s_2) = as' \Rightarrow (n_2', s_2'))$

lemma *WS-weak-sim*:

assumes $((n_1, s_1), (n_2, s_2)) \in WS\ n_c$
and $n_c, kind \vdash (n_1, s_1) = as \Rightarrow (n_1', s_1')$
shows $((n_1', s_1'), (n_1', transfer\ (slice-kind\ n_c\ (last\ as))\ s_2)) \in WS\ n_c \wedge$
 $(\exists as'. n_c, slice-kind\ n_c \vdash (n_2, s_2) = as@[last\ as] \Rightarrow$
 $(n_1', transfer\ (slice-kind\ n_c\ (last\ as))\ s_2))$
 $\langle proof \rangle$

The following lemma states the correctness of static intraprocedural slicing:
the simulation $WS\ n_c$ is a desired weak simulation

theorem *WS-is-weak-sim:is-weak-sim* $(WS\ n_c)\ n_c$
 $\langle proof \rangle$

3.2.5 $n - as \rightarrow^* n'$ **and transitive closure of** $n_c, f \vdash (n, s) = as \Rightarrow_\tau$
 (n', s')

inductive *trans-observable-moves* ::

$'node \Rightarrow ('edge \Rightarrow 'state\ edge-kind) \Rightarrow 'node \Rightarrow 'state \Rightarrow 'edge\ list \Rightarrow 'node \Rightarrow$

'state \Rightarrow bool
 $(-, - \vdash '(-, -) \Rightarrow^* '(-, -) [51, 50, 0, 0, 50, 0, 0] 51)$

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

| tom-Cons:
 $\llbracket n_c, f \vdash (n, s) = as \Rightarrow (n', s'); n_c, f \vdash (n', s') = as' \Rightarrow^* (n'', s'') \rrbracket$
 $\Rightarrow n_c, f \vdash (n, s) = (last\ as) \# as' \Rightarrow^* (n'', s'')$

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

lemma silent-moves-no-slice-edges:
 $n_c, f \vdash (n, s) = as \Rightarrow_\tau (n', s') \Rightarrow slice\ edges\ n_c\ as = []$
 <proof>

lemma observable-moves-last-slice-edges:
 $n_c, f \vdash (n, s) = as \Rightarrow (n', s') \Rightarrow slice\ edges\ n_c\ as = [last\ as]$
 <proof>

lemma slice-edges-no-nodes-in-slice:
 $slice\ edges\ n_c\ as = []$
 $\Rightarrow \forall nx \in set(sourcenodes\ as). nx \notin (backward\ slice\ n_c)$
 <proof>

lemma sliced-path-determ:
 $\llbracket n - as \rightarrow^* n'; n - as' \rightarrow^* n'; slice\ edges\ n'\ as = slice\ edges\ n'\ as';$
 $preds\ (slice\ kinds\ n'\ as)\ s; preds\ (slice\ kinds\ n'\ as')\ s';$
 $\forall V \in rv\ n'\ n.\ state\ val\ s\ V = state\ val\ s'\ V \rrbracket \Rightarrow as = as'$
 <proof>

lemma path-trans-observable-moves:
assumes $n - as \rightarrow^* n'$ **and** $preds\ (kinds\ as)\ s$ **and** $transfers\ (kinds\ as)\ s = s'$
obtains $n''\ s''\ as'\ as''$ **where** $n_c, kind \vdash (n, s) = slice\ edges\ n_c\ as \Rightarrow^* (n'', s'')$
and $n_c, kind \vdash (n'', s'') = as' \Rightarrow_\tau (n', s')$
and $slice\ edges\ n_c\ as = slice\ edges\ n_c\ as''$ **and** $n - as'' @ as' \rightarrow^* n'$
 <proof>

lemma WS-weak-sim-trans:

assumes $((n_1, s_1), (n_2, s_2)) \in WS\ n_c$
and $n_c, kind \vdash (n_1, s_1) = as \Rightarrow^* (n_1', s_1')$ **and** $as \neq []$
shows $((n_1', s_1'), (n_1', transfers\ (slice-kinds\ n_c\ as)\ s_2)) \in WS\ n_c \wedge$
 $n_c, slice-kind\ n_c \vdash (n_2, s_2) = as \Rightarrow^* (n_1', transfers\ (slice-kinds\ n_c\ as)\ s_2)$
 <proof>

lemma *transfers-slice-kinds-slice-edges*:
 $transfers\ (slice-kinds\ n_c\ (slice-edges\ n_c\ as))\ s =$
 $transfers\ (slice-kinds\ n_c\ as)\ s$
 <proof>

lemma *trans-observable-moves-preds*:
assumes $n_c, f \vdash (n, s) = as \Rightarrow^* (n', s')$ **and** *valid-node* n
obtains as' **where** $preds\ (map\ f\ as')\ s$ **and** $slice-edges\ n_c\ as' = as$
and $n - as' \rightarrow^* n'$
 <proof>

lemma *exists-sliced-path-preds*:
assumes $n - as \rightarrow^* n'$ **and** $slice-edges\ n_c\ as = []$ **and** $n' \in backward-slice\ n_c$
obtains as' **where** $n - as' \rightarrow^* n'$ **and** $preds\ (slice-kinds\ n_c\ as')\ s$
and $slice-edges\ n_c\ as' = []$
 <proof>

theorem *fundamental-property-of-static-slicing*:
assumes $path: n - as \rightarrow^* n'$ **and** $preds: preds\ (kinds\ as)\ s$
obtains as' **where** $preds\ (slice-kinds\ n'\ as')\ s$
and $(\forall V \in Use\ n'.\ state-val\ (transfers\ (slice-kinds\ n'\ as')\ s)\ V =$
 $state-val\ (transfers\ (kinds\ as)\ s)\ V)$
and $slice-edges\ n'\ as = slice-edges\ n'\ as'$ **and** $n - as' \rightarrow^* n'$
 <proof>

end

3.2.6 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 $sourcenode :: 'edge \Rightarrow 'node$ **and** $targetnode :: 'edge \Rightarrow 'node$
and $kind :: 'edge \Rightarrow 'state\ edge-kind$ **and** $valid-edge :: 'edge \Rightarrow bool$
and $Entry :: 'node\ (('Entry')$ **and** $Def :: 'node \Rightarrow 'var\ set$

lemma *obs-singleton*:
assumes *valid:valid-node n*
shows $(\exists m. \text{obs } n \text{ (PDG-BS } n_c) = \{m\}) \vee \text{obs } n \text{ (PDG-BS } n_c) = \{\}$
 $\langle \text{proof} \rangle$

lemma *PDGBackwardSliceCorrect*:
BackwardSlice sourcenode targetnode kind valid-edge
(-Entry-) Def Use state-val PDG-BS
 $\langle \text{proof} \rangle$

end

3.3.2 Weak control dependence

context *WeakControlDependencePDG* **begin**

lemma *Exit-in-obs-slice-node:(-Exit-) ∈ obs n' (PDG-BS n_c) ⇒ n_c = (-Exit-)*
 $\langle \text{proof} \rangle$

lemma *cd-closed*:
assumes *isin:n' ∈ PDG-BS n_c and controls:n weakly controls n'*
shows *n ∈ PDG-BS n_c*
 $\langle \text{proof} \rangle$

lemma *obs-strong-postdominate*:
assumes *isin:n ∈ obs n' (PDG-BS n_c) and sliceNotExit:n_c ≠ (-Exit-)*
shows *n strongly-postdominates n'*
 $\langle \text{proof} \rangle$

lemma *obs-singleton*:
assumes *valid:valid-node n*
shows $(\exists m. \text{obs } n \text{ (PDG-BS } n_c) = \{m\}) \vee \text{obs } n \text{ (PDG-BS } n_c) = \{\}$
 $\langle \text{proof} \rangle$

lemma *WeakPDGBackwardSliceCorrect*:
BackwardSlice sourcenode targetnode kind valid-edge
(-Entry-) Def Use state-val PDG-BS
 $\langle \text{proof} \rangle$

end

3.3.3 Weak order dependence

context *CFG-wf* **begin**

lemma *obs-singleton*:
 assumes *valid:valid-node n*
 shows $(\exists m. \text{obs } n \text{ (wod-backward-slice } n_c) = \{m\}) \vee$
 $\text{obs } n \text{ (wod-backward-slice } n_c) = \{\}$
<proof>

lemma *WOD-valid-backward-slice*:
 BackwardSlice sourcenode targetnode kind valid-edge
 (-Entry-) Def Use state-val wod-backward-slice
<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

```
types 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 v1 v2 = Some(Bool(v1 = v2))  
  | binop And (Bool b1) (Bool b2) = Some(Bool(b1 & b2))  
  | binop Less (Intg i1) (Intg i2) = Some(Bool(i1 < i2))  
  | binop Add (Intg i1) (Intg i2) = Some(Intg(i1 + i2))
```

```

| 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

types

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* (*e*₁⋖*bop*⋗*e*₂) *s* =

(*case interpret e*₁ *s of None* ⇒ *None*

| *Some v*₁ ⇒ (*case interpret e*₂ *s of None* ⇒ *None*

| *Some v*₂ ⇒ (

*case binop bop v*₁ *v*₂ *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* \Rightarrow *nat* \Rightarrow *w-node* (- \oplus - 60)

where (- *l* -) \oplus *i* = (- *l* + *i* -)
 | (-*Entry*-) \oplus *i* = (-*Entry*-)
 | (-*Exit*-) \oplus *i* = (-*Exit*-)

lemma *Exit-label-incr* [*dest*]: (-*Exit*-) = $n \oplus i \Longrightarrow n = (-\text{Exit-})$
 \langle *proof* \rangle

lemma *label-incr-Exit* [*dest*]: $n \oplus i = (-\text{Exit-}) \Longrightarrow n = (-\text{Exit-})$
 \langle *proof* \rangle

lemma *Entry-label-incr* [*dest*]: (-*Entry*-) = $n \oplus i \Longrightarrow n = (-\text{Entry-})$
 \langle *proof* \rangle

lemma *label-incr-Entry* [*dest*]: $n \oplus i = (-\text{Entry-}) \Longrightarrow n = (-\text{Entry-})$
 \langle *proof* \rangle

lemma *label-incr-inj*:
 $n \oplus c = n' \oplus c \Longrightarrow n = n'$
 \langle *proof* \rangle

lemma *label-incr-simp*: $n \oplus i = m \oplus (i + j) \Longrightarrow n = m \oplus j$
 \langle *proof* \rangle

lemma *label-incr-simp-rev*: $m \oplus (j + i) = n \oplus i \Longrightarrow m \oplus j = n$
 \langle *proof* \rangle

lemma *label-incr-start-Node-smaller*:
 (- *l* -) = $n \oplus i \Longrightarrow n = (-\text{(l - i)-})$
 \langle *proof* \rangle

lemma *label-incr-ge*: (- *l* -) = $n \oplus i \Longrightarrow l \geq i$
 \langle *proof* \rangle

lemma *label-incr-0* [*dest*]:
 $\llbracket (-0-) = n \oplus i; i > 0 \rrbracket \Longrightarrow \text{False}$
 \langle *proof* \rangle

lemma *label-incr-0-rev* [*dest*]:
 $\llbracket n \oplus i = (-0-); i > 0 \rrbracket \Longrightarrow \text{False}$
 \langle *proof* \rangle

4.2.2 CFG edges

types $w\text{-edge} = (w\text{-node} \times \text{state edge-kind} \times w\text{-node})$

inductive $\text{While-CFG} :: \text{cmd} \Rightarrow w\text{-node} \Rightarrow \text{state edge-kind} \Rightarrow w\text{-node} \Rightarrow \text{bool}$
 $(- \vdash - \dashrightarrow -)$

where

WCFG-Entry-Exit:

$\text{prog} \vdash (-\text{Entry-}) -(\lambda s. \text{False})_{\surd} \rightarrow (-\text{Exit-})$

| *WCFG-Entry:*

$\text{prog} \vdash (-\text{Entry-}) -(\lambda s. \text{True})_{\surd} \rightarrow (-0-)$

| *WCFG-Skip:*

$\text{Skip} \vdash (-0-) -\uparrow \text{id} \rightarrow (-\text{Exit-})$

| *WCFG-LAss:*

$V := e \vdash (-0-) -\uparrow (\lambda s. s(V := (\text{interpret } e \ s))) \rightarrow (-1-)$

| *WCFG-LAssSkip:*

$V := e \vdash (-1-) -\uparrow \text{id} \rightarrow (-\text{Exit-})$

| *WCFG-SeqFirst:*

$\llbracket c_1 \vdash n -et \rightarrow n'; n' \neq (-\text{Exit-}) \rrbracket \Longrightarrow c_1;;c_2 \vdash n -et \rightarrow n'$

| *WCFG-SeqConnect:*

$\llbracket c_1 \vdash n -et \rightarrow (-\text{Exit-}); n \neq (-\text{Entry-}) \rrbracket \Longrightarrow c_1;;c_2 \vdash n -et \rightarrow (-0-) \oplus \#:c_1$

| *WCFG-SeqSecond:*

$\llbracket c_2 \vdash n -et \rightarrow n'; n \neq (-\text{Entry-}) \rrbracket \Longrightarrow c_1;;c_2 \vdash n \oplus \#:c_1 -et \rightarrow n' \oplus \#:c_1$

| *WCFG-CondTrue:*

$\text{if } (b) \ c_1 \ \text{else } c_2 \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some true})_{\surd} \rightarrow (-0-) \oplus 1$

| *WCFG-CondFalse:*

$\text{if } (b) \ c_1 \ \text{else } c_2 \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some false})_{\surd} \rightarrow (-0-) \oplus (\#:c_1 + 1)$

| *WCFG-CondThen:*

$\llbracket c_1 \vdash n -et \rightarrow n'; n \neq (-\text{Entry-}) \rrbracket \Longrightarrow \text{if } (b) \ c_1 \ \text{else } c_2 \vdash n \oplus 1 -et \rightarrow n' \oplus 1$

| *WCFG-CondElse:*

$\llbracket c_2 \vdash n -et \rightarrow n'; n \neq (-\text{Entry-}) \rrbracket$
 $\Longrightarrow \text{if } (b) \ c_1 \ \text{else } c_2 \vdash n \oplus (\#:c_1 + 1) -et \rightarrow n' \oplus (\#:c_1 + 1)$

| *WCFG-WhileTrue:*

$\text{while } (b) \ c' \vdash (-0-) -(\lambda s. \text{interpret } b \ s = \text{Some true})_{\surd} \rightarrow (-0-) \oplus 2$

| *WCFG-WhileFalse:*

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

| *WCFG-WhileFalseSkip*:

$while (b) c' \vdash (-1-) \text{ --}\uparrow id \rightarrow (-Exit-)$

| *WCFG-WhileBody*:

$\llbracket c' \vdash n \text{ --}et \rightarrow n'; n \neq (-Entry-); n' \neq (-Exit-) \rrbracket$

$\implies while (b) c' \vdash n \oplus 2 \text{ --}et \rightarrow n' \oplus 2$

| *WCFG-WhileBodyExit*:

$\llbracket c' \vdash n \text{ --}et \rightarrow (-Exit-); n \neq (-Entry-) \rrbracket \implies while (b) c' \vdash n \oplus 2 \text{ --}et \rightarrow (-0-)$

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

lemmas *WCFG-elim*s = *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*]:

$prog \vdash (-Exit-) \text{ --}et \rightarrow n' \implies False$

<proof>

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

$prog \vdash n \text{ --}et \rightarrow (-Entry-) \implies False$

<proof>

lemma *WCFG-sourcelabel-less-num-nodes*:

$prog \vdash (-l-) \text{ --}et \rightarrow n' \implies l < \#:prog$

<proof>

lemma *WCFG-targetlabel-less-num-nodes*:

$prog \vdash n \text{ --}et \rightarrow (-l-) \implies l < \#:prog$

<proof>

lemma *WCFG-EntryD*:

$prog \vdash (-Entry-) \text{ --}et \rightarrow n'$

$\implies (n' = (-Exit-) \wedge et = (\lambda s. False)_ \surd) \vee (n' = (-0-) \wedge et = (\lambda s. True)_ \surd)$

<proof>

lemma *WCFG-edge-det*:

$\llbracket prog \vdash n \text{ --}et \rightarrow n'; prog \vdash n \text{ --}et' \rightarrow n \rrbracket \implies et = et'$

<proof>

lemma *less-num-nodes-edge-Exit*:

obtains *l et* **where** $l < \#:prog$ **and** $prog \vdash (-l-) \text{ --}et \rightarrow (-Exit-)$

<proof>

lemma *less-num-nodes-edge*:

$l < \# : \text{prog} \implies \exists n \text{ et. } \text{prog} \vdash n - \text{et} \rightarrow (- l -) \vee \text{prog} \vdash (- l -) - \text{et} \rightarrow n$
(proof)

lemma *WCFG-deterministic*:

$\llbracket \text{prog} \vdash n_1 - \text{et}_1 \rightarrow n_1'; \text{prog} \vdash n_2 - \text{et}_2 \rightarrow n_2'; n_1 = n_2; n_1' \neq n_2' \rrbracket$
 $\implies \exists Q Q'. \text{et}_1 = (Q)_{\checkmark} \wedge \text{et}_2 = (Q')_{\checkmark} \wedge (\forall s. (Q s \longrightarrow \neg Q' s) \wedge (Q' s \longrightarrow \neg Q s))$
(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* \Rightarrow *w-node*
where *sourcenode* $e \equiv \text{fst } e$

abbreviation *targetnode* :: *w-edge* \Rightarrow *w-node*
where *targetnode* $e \equiv \text{snd}(\text{snd } e)$

abbreviation *kind* :: *w-edge* \Rightarrow *state edge-kind*
where *kind* $e \equiv \text{fst}(\text{snd } e)$

definition *valid-edge* :: *cmd* \Rightarrow *w-edge* \Rightarrow *bool*
where *valid-edge* $\text{prog } a \equiv \text{prog} \vdash \text{sourcenode } a - \text{kind } a \rightarrow \text{targetnode } a$

definition *valid-node* :: *cmd* \Rightarrow *w-node* \Rightarrow *bool*
where *valid-node* $\text{prog } n \equiv$
 $(\exists a. \text{valid-edge } \text{prog } a \wedge (n = \text{sourcenode } a \vee n = \text{targetnode } a))$

interpretation *While-CFG*:

CFG sourcenode targetnode kind valid-edge prog Entry
(proof)

interpretation *While-CFGExit:CFGExit* *sourcenode targetnode kind*
valid-edge prog Entry Exit

(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* \Rightarrow *nat* \Rightarrow *cmd* \Rightarrow *bool*
where

Labels-Base:
labels *c* 0 *c*

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

| *Labels-Seq1*:
labels *c*₁ *l* *c* \Longrightarrow *labels* (*c*₁;;*c*₂) *l* (*c*;;*c*₂)

| *Labels-Seq2*:
labels *c*₂ *l* *c* \Longrightarrow *labels* (*c*₁;;*c*₂) (*l* + #:*c*₁) *c*

| *Labels-CondTrue*:
labels *c*₁ *l* *c* \Longrightarrow *labels* (*if* (*b*) *c*₁ *else* *c*₂) (*l* + 1) *c*

| *Labels-CondFalse*:
labels *c*₂ *l* *c* \Longrightarrow *labels* (*if* (*b*) *c*₁ *else* *c*₂) (*l* + #:*c*₁ + 1) *c*

| *Labels-WhileBody*:
labels *c'* *l* *c* \Longrightarrow *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'* \Longrightarrow *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
begin
```

4.5.1 Definition of some functions

```
fun lhs :: cmd ⇒ vname set
```

```
where
```

```
  lhs Skip                = {}
| lhs (V:=e)              = {V}
| lhs (c1;;c2)            = lhs c1
| lhs (if (b) c1 else c2) = {}
| lhs (while (b) c)      = {}
```

```
fun rhs-aux :: expr ⇒ vname set
```

```
where
```

```
  rhs-aux (Val v)        = {}
| rhs-aux (Var V)       = {V}
| rhs-aux (e1 <<bop>> e2) = (rhs-aux e1 ∪ rhs-aux e2)
```

```
fun rhs :: cmd ⇒ vname set
```

```
where
```

```
  rhs Skip                = {}
| rhs (V:=e)              = rhs-aux e
| rhs (c1;;c2)            = rhs c1
| rhs (if (b) c1 else c2) = rhs-aux b
| rhs (while (b) c)      = rhs-aux b
```

```
lemma rhs-interpret-eq:
```

```
  [[interpret b s = Some v'; ∀ V ∈ rhs-aux b. s V = s' V]]
  ⇒ interpret b s' = Some v'
```

```
⟨proof⟩
```

```
fun Defs :: cmd ⇒ w-node ⇒ vname set
```

```
where Defs prog n = {V. ∃ l c. n = (- l -) ∧ labels prog l c ∧ V ∈ lhs c}
```

```
fun Uses :: cmd ⇒ w-node ⇒ vname set
```

```
where Uses prog n = {V. ∃ l c. n = (- l -) ∧ labels prog l c ∧ V ∈ rhs c}
```

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

lemma *WCFG-edge-no-Defs-equal*:

$\llbracket prog \vdash n -et \rightarrow n'; V \notin Defs\ prog\ n \rrbracket \implies (transfer\ et\ s)\ V = s\ V$
 $\langle proof \rangle$

lemma *WCFG-edge-transfer-uses-only-Uses*:

$\llbracket prog \vdash n -et \rightarrow n'; \forall V \in Uses\ prog\ n.\ s\ V = s'\ V \rrbracket$
 $\implies \forall V \in Defs\ prog\ n.\ (transfer\ et\ s)\ V = (transfer\ et\ s')\ V$
 $\langle proof \rangle$

lemma *WCFG-edge-Uses-pred-eq*:

$\llbracket prog \vdash n -et \rightarrow n'; \forall V \in Uses\ prog\ n.\ s\ V = s'\ V; pred\ et\ s \rrbracket$
 $\implies pred\ et\ s'$
 $\langle proof \rangle$

interpretation *While-CFG-wf*: *CFG-wf sourcenode targetnode kind*
valid-edge prog Entry Defs prog Uses prog id

$\langle proof \rangle$

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

$\langle proof \rangle$

end

4.6 Semantics

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

4.6.1 Small Step Semantics

inductive *red* :: *cmd * state* \Rightarrow *cmd * state* \Rightarrow *bool*

and *red'* :: *cmd* \Rightarrow *state* \Rightarrow *cmd* \Rightarrow *state* \Rightarrow *bool*

$((((1\langle -,/- \rangle) \rightarrow / (1\langle -,/- \rangle)) [0,0,0,0] 81)$

where

$\langle c, s \rangle \rightarrow \langle c', s' \rangle == red\ (c, s)\ (c', s')$

| *RedLAss*:

$\langle V := e, s \rangle \rightarrow \langle Skip, s(V := (interpret\ e\ s)) \rangle$

| *SeqRed*:

$\langle c_1, s \rangle \rightarrow \langle c_1', s' \rangle \implies \langle c_1 ;; c_2, s \rangle \rightarrow \langle c_1' ;; c_2, s' \rangle$

| *RedSeq*:

$\langle Skip ;; c_2, s \rangle \rightarrow \langle c_2, s \rangle$

| *RedCondTrue*:
 $\text{interpret } b \ s = \text{Some true} \implies \langle \text{if } (b) \ c_1 \ \text{else } c_2, s \rangle \rightarrow \langle c_1, s \rangle$

| *RedCondFalse*:
 $\text{interpret } b \ s = \text{Some false} \implies \langle \text{if } (b) \ c_1 \ \text{else } c_2, s \rangle \rightarrow \langle c_2, s \rangle$

| *RedWhileTrue*:
 $\text{interpret } b \ s = \text{Some true} \implies \langle \text{while } (b) \ c, s \rangle \rightarrow \langle c;; \text{while } (b) \ c, s \rangle$

| *RedWhileFalse*:
 $\text{interpret } b \ s = \text{Some false} \implies \langle \text{while } (b) \ c, s \rangle \rightarrow \langle \text{Skip}, s \rangle$

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

abbreviation *reds* :: *cmd* \Rightarrow *state* \Rightarrow *cmd* \Rightarrow *state* \Rightarrow *bool*
 $((1 \langle -, / - \rangle) \rightarrow * / (1 \langle -, / - \rangle)) [0, 0, 0, 0] \ 81$ **where**
 $\langle c, s \rangle \rightarrow * \langle c', s' \rangle == \text{red}^{**} (c, s) (c', s')$

4.6.2 Label Semantics

inductive *step* :: *cmd* \Rightarrow *cmd* \Rightarrow *state* \Rightarrow *nat* \Rightarrow *cmd* \Rightarrow *state* \Rightarrow *nat* \Rightarrow *bool*
 $((- \vdash (1 \langle -, / - \rangle) \rightsquigarrow / (1 \langle -, / - \rangle)) [51, 0, 0, 0, 0, 0, 0] \ 81)$
where

StepLAss:
 $V := e \vdash \langle V := e, s, 0 \rangle \rightsquigarrow \langle \text{Skip}, s (V := (\text{interpret } e \ s)), 1 \rangle$

| *StepSeq*:
 $\llbracket \text{labels } (c_1;;c_2) \ l \ (\text{Skip};;c_2); \ \text{labels } (c_1;;c_2) \ \# : c_1 \ c_2; \ l < \# : c_1 \rrbracket$
 $\implies c_1;;c_2 \vdash \langle \text{Skip};;c_2, s, l \rangle \rightsquigarrow \langle c_2, s, \# : c_1 \rangle$

| *StepSeqWhile*:
 $\text{labels } (\text{while } (b) \ c') \ l \ (\text{Skip};; \text{while } (b) \ c')$
 $\implies \text{while } (b) \ c' \vdash \langle \text{Skip};; \text{while } (b) \ c', s, l \rangle \rightsquigarrow \langle \text{while } (b) \ c', s, 0 \rangle$

| *StepCondTrue*:
 $\text{interpret } b \ s = \text{Some true}$
 $\implies \text{if } (b) \ c_1 \ \text{else } c_2 \vdash \langle \text{if } (b) \ c_1 \ \text{else } c_2, s, 0 \rangle \rightsquigarrow \langle c_1, s, 1 \rangle$

| *StepCondFalse*:
 $\text{interpret } b \ s = \text{Some false}$
 $\implies \text{if } (b) \ c_1 \ \text{else } c_2 \vdash \langle \text{if } (b) \ c_1 \ \text{else } c_2, s, 0 \rangle \rightsquigarrow \langle c_2, s, \# : c_1 + 1 \rangle$

| *StepWhileTrue*:
 $\text{interpret } b \ s = \text{Some true}$
 $\implies \text{while } (b) \ c \vdash \langle \text{while } (b) \ c, s, 0 \rangle \rightsquigarrow \langle c;; \text{while } (b) \ c, s, 2 \rangle$

| *StepWhileFalse*:
 $\text{interpret } b \ s = \text{Some false} \implies \text{while } (b) \ c \vdash \langle \text{while } (b) \ c, s, 0 \rangle \rightsquigarrow \langle \text{Skip}, s, 1 \rangle$

| *StepRecSeq1*:
 $prog \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$
 $\implies prog;; c_2 \vdash \langle c;; c_2, s, l \rangle \rightsquigarrow \langle c';; c_2, s', l' \rangle$

| *StepRecSeq2*:
 $prog \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$
 $\implies c_1;; prog \vdash \langle c, s, l + \#:c_1 \rangle \rightsquigarrow \langle c', s', l' + \#:c_1 \rangle$

| *StepRecCond1*:
 $prog \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$
 $\implies \text{if } (b) \text{ prog else } c_2 \vdash \langle c, s, l + 1 \rangle \rightsquigarrow \langle c', s', l' + 1 \rangle$

| *StepRecCond2*:
 $prog \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$
 $\implies \text{if } (b) \text{ } c_1 \text{ else } prog \vdash \langle c, s, l + \#:c_1 + 1 \rangle \rightsquigarrow \langle c', s', l' + \#:c_1 + 1 \rangle$

| *StepRecWhile*:
 $cx \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle$
 $\implies \text{while } (b) \text{ } cx \vdash \langle c;; \text{while } (b) \text{ } cx, s, l + 2 \rangle \rightsquigarrow \langle c';; \text{while } (b) \text{ } cx, s', l' + 2 \rangle$

lemma *step-label-less*:

$prog \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \implies l < \#:prog \wedge l' < \#:prog$
 $\langle \text{proof} \rangle$

abbreviation *steps* :: $cmd \Rightarrow cmd \Rightarrow state \Rightarrow nat \Rightarrow cmd \Rightarrow state \Rightarrow nat \Rightarrow bool$

$((- \vdash (1 \langle -, /-, /- \rangle) \rightsquigarrow^* / (1 \langle -, /-, /- \rangle))) [51, 0, 0, 0, 0, 0, 0] \text{ 81} \text{ where}$
 $prog \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle ==$
 $(\lambda(c, s, l) (c', s', l'). prog \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle)^{**} (c, s, l) (c', s', l')$

4.6.3 Proof of bisimulation of $\langle c, s \rangle \rightarrow \langle c', s' \rangle$ 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** $\langle c, s \rangle \rightarrow \langle c', s' \rangle$

lemma *step-red*:

$prog \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \implies \langle c, s \rangle \rightarrow \langle c', s' \rangle$
 $\langle \text{proof} \rangle$

lemma *steps-reds*:

$prog \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle \implies \langle c, s \rangle \rightarrow^* \langle c', s' \rangle$
 $\langle \text{proof} \rangle$

From $\langle c, s \rangle \rightarrow \langle c', s' \rangle$ **and labels to** $\text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle$

lemma *red-step*:

$\llbracket \text{labels prog } l \ c; \langle c, s \rangle \rightarrow \langle c', s' \rangle \rrbracket$
 $\implies \exists l'. \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \wedge \text{labels prog } l' \ c'$
 $\langle \text{proof} \rangle$

lemma *reds-steps*:

$\llbracket \langle c, s \rangle \rightarrow^* \langle c', s' \rangle; \text{labels prog } l \ c \rrbracket$
 $\implies \exists l'. \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle \wedge \text{labels prog } l' \ c'$
 $\langle \text{proof} \rangle$

The bisimulation theorem

theorem *reds-steps-bisimulation*:

$\text{labels prog } l \ c \implies (\langle c, s \rangle \rightarrow^* \langle c', s' \rangle) =$
 $(\exists l'. \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow^* \langle c', s', l' \rangle \wedge \text{labels prog } l' \ c')$
 $\langle \text{proof} \rangle$

end

4.7 Equivalence

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

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

lemma *Skip-WCFG-edge-Exit*:

$\llbracket \text{labels prog } l \ \text{Skip} \rrbracket \implies \text{prog} \vdash (- \ l \ -) \text{-}\uparrow\text{id} \rightarrow (- \ \text{Exit} \ -)$
 $\langle \text{proof} \rangle$

lemma *step-WCFG-edge*:

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

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

lemma *WCFG-edge-Exit-Skip*:

$\llbracket \text{prog} \vdash n \text{-et} \rightarrow (- \ \text{Exit} \ -); n \neq (- \ \text{Entry} \ -) \rrbracket$
 $\implies \exists l. n = (- \ l \ -) \wedge \text{labels prog } l \ \text{Skip} \wedge \text{et} = \uparrow\text{id}$
 $\langle \text{proof} \rangle$

lemma *WCFG-edge-step*:
 $\llbracket \text{prog} \vdash (- l -) -et \rightarrow (- l' -); \text{transfer et } s = s'; \text{pred et } s \rrbracket$
 $\implies \exists c c'. \text{prog} \vdash \langle c, s, l \rangle \rightsquigarrow \langle c', s', l' \rangle \wedge \text{labels prog } l c \wedge \text{labels prog } l' c'$
 $\langle \text{proof} \rangle$

end

4.8 Semantic well-formedness of While CFG

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

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

fun *labels-nodes* :: *cmd* \Rightarrow *w-node* \Rightarrow *cmd* \Rightarrow *bool* **where**
 $\text{labels-nodes prog } (- l -) c = \text{labels prog } l c$
 $|\ \text{labels-nodes prog } (-\text{Entry-}) c = \text{False}$
 $|\ \text{labels-nodes prog } (-\text{Exit-}) c = \text{False}$

interpretation *While-semantics-CFG-wf*: *CFG-semantics-wf*
 $\text{sourcenode targetnode kind valid-edge prog Entry reds labels-nodes prog}$
 $\langle \text{proof} \rangle$

end

4.9 Lemmas for the control dependences

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

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

abbreviation *path* :: *cmd* \Rightarrow *w-node* \Rightarrow *w-edge list* \Rightarrow *w-node* \Rightarrow *bool*
 $(- \vdash - \dashrightarrow^* -)$
where $\text{prog} \vdash n -as \rightarrow^* n' \equiv \text{CFG.path sourcenode targetnode (valid-edge prog)}$

$n \text{ as } n'$

definition *label-incrs* :: *w-edge list* \Rightarrow *nat* \Rightarrow *w-edge list* $(- \oplus s - 60)$
where $as \oplus s i \equiv \text{map } (\lambda(n, et, n'). (n \oplus i, et, n' \oplus i)) as$

lemma *path-SeqFirst*:

$prog \vdash n -as \rightarrow^* (- l -) \implies prog;;c_2 \vdash n -as \rightarrow^* (- l -)$
<proof>

lemma *path-SeqSecond*:

$\llbracket prog \vdash n -as \rightarrow^* n'; n \neq (-Entry-); as \neq [] \rrbracket$
 $\implies c_1;;prog \vdash n \oplus \#:c_1 -as \oplus s \#:c_1 \rightarrow^* n' \oplus \#:c_1$
<proof>

lemma *path-CondTrue*:

$prog \vdash (- l -) -as \rightarrow^* n'$
 $\implies if (b) prog else c_2 \vdash (- l -) \oplus 1 -as \oplus s 1 \rightarrow^* n' \oplus 1$
<proof>

lemma *path-CondFalse*:

$prog \vdash (- l -) -as \rightarrow^* n'$
 $\implies if (b) c_1 else prog \vdash (- l -) \oplus (\#:c_1 + 1) -as \oplus s (\#:c_1 + 1) \rightarrow^* n' \oplus (\#:c_1 + 1)$
<proof>

lemma *path-While*:

$prog \vdash (- l -) -as \rightarrow^* (- l' -)$
 $\implies while (b) prog \vdash (- l -) \oplus 2 -as \oplus s 2 \rightarrow^* (- l' -) \oplus 2$
<proof>

lemma *inner-node-Entry-Exit-path*:

$l < \#:prog \implies (\exists as. prog \vdash (- l -) -as \rightarrow^* (-Exit-)) \wedge$
 $(\exists as. prog \vdash (-Entry-) -as \rightarrow^* (- l -))$
<proof>

lemma *valid-node-Exit-path*:

assumes *valid-node* *prog n shows* $\exists as. prog \vdash n -as \rightarrow^* (-Exit-)$
<proof>

lemma *valid-node-Entry-path*:

assumes *valid-node* *prog n shows* $\exists as. prog \vdash (-Entry-) -as \rightarrow^* n$
<proof>

4.9.2 Some finiteness considerations

lemma *finite-labels:finite* $\{l. \exists c. labels prog l c\}$

<proof>

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

lemma *finite-successors*:
finite {*n'. ∃ a'. valid-edge prog a' ∧ sourcenode a' = n ∧*
targetnode a' = n'}
⟨*proof*⟩

end

4.10 Interpretations of the various dynamic control dependences

theory *DynamicControlDependences* **imports** *AdditionalLemmas*
../Basic/StandardControlDependence
../Basic/WeakControlDependence
begin

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

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

end

4.11 Interpretations of the various static control dependences

theory *StaticControlDependences* **imports** *AdditionalLemmas*
../Basic/StandardControlDependence
../Basic/WeakControlDependence
begin

interpretation *WStandardControlDependence*:
StandardControlDependencePDG 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>

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

```
typedef wf-jvmprog = {(P, Phi). wf-jvm-prog_Phi P}  
<proof>
```

```
hide const Phi  
hide const E
```

```
abbreviation rep-jvmprog-jvm-prog :: wf-jvmprog  $\Rightarrow$  jvm-prog  
(-wf)  
  where  $P_{wf} \equiv fst(Rep-wf-jvmprog(P))$ 
```

```
abbreviation rep-jvmprog-phi :: wf-jvmprog  $\Rightarrow$  ty_P  
(-Phi)  
  where  $P_{\Phi} \equiv snd(Rep-wf-jvmprog(P))$ 
```

```
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

types $jvmprog = wf-jvmprog \times cname \times mname$
types $callstack = (cname \times mname \times pc) list$

The state is modeled as $heap \times stack\text{-}variables \times local\text{-}variables$

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.

types $state = heap \times ((nat \times nat) \Rightarrow val) \times ((nat \times nat) \Rightarrow val)$

abbreviation $heap\text{-}of :: state \Rightarrow heap$

where

$heap\text{-}of\ s \equiv fst(s)$

abbreviation $stk\text{-}of :: state \Rightarrow ((nat \times nat) \Rightarrow val)$

where

$stk\text{-}of\ s \equiv fst(snd(s))$

abbreviation $loc\text{-}of :: state \Rightarrow ((nat \times nat) \Rightarrow val)$

where

$loc\text{-}of\ s \equiv snd(snd(s))$

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 $exec\text{-}instr :: instr \Rightarrow wf\text{-}jvmprog \Rightarrow state \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow state$

where

$exec\text{-}instr\text{-}Load:$

$exec\text{-}instr\ (Load\ n)\ P\ s\ calldepth\ stk\text{-}length\ rs\ ill =$

$(let\ (h,stk,loc) = s$

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

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

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

$| exec-instr-New:$
 $exec-instr (New C) P s calldepth stk-length rs ill =$
 $(let (h, stk, loc) = s;$
 $a = the(new-Addr h)$
 $in (h(a \mapsto (blank (P_{wf} C)), stk((calldepth, stk-length) := Addr a), loc))$

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

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

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

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

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

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

| *exec-instr-CmpEq*:
exec-instr (CmpEq) P s calldepth stk-length rs ill =
(let (h,stk,loc) = s;
v₁ = stk (calldepth, stk-length - 1);
v₂ = stk (calldepth, stk-length - 2)
in (h, stk((calldepth, stk-length - 2) := Bool (v₁ = v₂)), 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_Φ) C M)!pc)))*

abbreviation *locLength* :: *wf-jvmprog* ⇒ *cname* ⇒ *mname* ⇒ *pc* ⇒ *nat*

where

locLength P C M pc ≡ *length (snd(the(((P_Φ) 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 $locs :: nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a \text{ list}$
where $locs \ n \ loc \equiv map \ loc \ [0..<n]$

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

This is used for extracting the stack as a list

abbreviation $stks :: nat \Rightarrow (nat \Rightarrow 'a) \Rightarrow 'a \text{ list}$
where $stks \ n \ stk \equiv map \ stk \ (rev \ [0..<n])$

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

fun $locss :: wf-jvmprog \Rightarrow callstack \Rightarrow ((nat \times nat) \Rightarrow 'a) \Rightarrow 'a \text{ list list}$
where
 $locss \ P \ [] \ loc = []$
 $| \ locss \ P \ ((C,M,pc)\#cs) \ loc =$
 $(locs \ (locLength \ P \ C \ M \ pc) \ (\lambda a. \ loc \ (length \ cs, \ a)))\#(locss \ P \ cs \ loc)$

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

fun $stkss :: wf-jvmprog \Rightarrow callstack \Rightarrow ((nat \times nat) \Rightarrow 'a) \Rightarrow 'a \text{ list list}$
where
 $stkss \ P \ [] \ stk = []$
 $| \ stkss \ P \ ((C,M,pc)\#cs) \ stk =$
 $(stks \ (stkLength \ P \ C \ M \ pc) \ (\lambda a. \ stk \ (length \ cs, \ a)))\#(stkss \ P \ cs \ stk)$

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

abbreviation $state-to-jvm-state :: wf-jvmprog \Rightarrow callstack \Rightarrow state \Rightarrow jvm-state$
where $state-to-jvm-state \ P \ cs \ s \equiv$
 $(None, \ heap-of \ s, \ zip \ (stkss \ P \ cs \ (stk-of \ s)) \ (zip \ (locss \ P \ cs \ (loc-of \ s)) \ cs))$

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

definition $framestack-to-callstack :: frame \ list \Rightarrow callstack$
where $framestack-to-callstack \ frs \equiv map \ snd \ (map \ snd \ frs)$

State Conformance

Now we lift byte code verifier conformance to our state representation

definition $bv-conform :: wf-jvmprog \Rightarrow callstack \Rightarrow state \Rightarrow bool$
 $(-, \vdash_{BV} - \checkmark)$
where $P, cs \vdash_{BV} s \checkmark \equiv correct-state \ (P_{wf}) \ (P_{\Phi}) \ (state-to-jvm-state \ P \ cs \ s)$

Statically determine catch-block

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

fun *find-handler-for* :: *wf-jvmprog* \Rightarrow *cname* \Rightarrow *callstack* \Rightarrow *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 \Rightarrow *find-handler-for P C cs*
| *Some pc-d* \Rightarrow (*C', M', fst pc-d*)#*cs*))

5.1.3 Simplification lemmas

lemma *find-handler-decr* [*simp*]: *find-handler-for P Exc cs* \neq *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))*]]
 \Rightarrow *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))*]]
 \Rightarrow *locss P cs loc ! (length cs - Suc a) ! b* = *loc (a,b)*
<*proof*>

lemma *hd-stks* [*simp*]: *n* \neq 0 \Rightarrow *hd (stks n stk)* = *stk(n - 1)*
<*proof*>

lemma *hd-tl-stks*: *n* > 1 \Rightarrow *hd (tl (stks n stk))* = *stk(n - 2)*
<*proof*>

lemma *stkss-purge*:
length cs \leq *a* \Rightarrow *stkss P cs (stk((a,b) := c))* = *stkss P cs stk*
<*proof*>

lemma *stkss-purge'*:

$length\ cs \leq a \implies stkss\ P\ cs\ (\lambda s. \text{if } s = (a, b) \text{ then } c \text{ else } stk\ s) = stkss\ P\ cs\ stk$
 ⟨proof⟩

lemma *locss-purge*:

$length\ cs \leq a \implies locss\ P\ cs\ (loc((a,b) := c)) = locss\ P\ cs\ loc$
 ⟨proof⟩

lemma *locss-purge'*:

$length\ cs \leq a \implies locss\ P\ cs\ (\lambda s. \text{if } s = (a, b) \text{ then } c \text{ else } loc\ s) = locss\ P\ cs\ loc$
 ⟨proof⟩

lemma *locs-pullout* [simp]:

$locs\ b\ (loc(n := e)) = locs\ b\ loc\ [n := e]$
 ⟨proof⟩

lemma *locs-pullout'* [simp]:

$locs\ b\ (\lambda a. \text{if } a = n \text{ then } e \text{ else } loc\ (c, a)) = locs\ b\ (\lambda a. loc\ (c, a))\ [n := e]$
 ⟨proof⟩

lemma *stks-pullout*:

$n < b \implies stks\ b\ (stk(n := e)) = stks\ b\ stk\ [b - Suc\ n := e]$
 ⟨proof⟩

lemma *nth-tl* : $xs \neq [] \implies tl\ xs\ !\ n = xs\ !\ (Suc\ n)$

⟨proof⟩

lemma *f2c-Nil* [simp]: $framestack\ \text{-to-callstack}\ [] = []$

⟨proof⟩

lemma *f2c-Cons* [simp]:

$framestack\ \text{-to-callstack}\ ((stk, loc, C, M, pc) \# frs) = (C, M, pc) \# (framestack\ \text{-to-callstack}\ frs)$

⟨proof⟩

lemma *f2c-length* [simp]:

$length\ (framestack\ \text{-to-callstack}\ frs) = length\ frs$

⟨proof⟩

lemma *f2c-s2jvm-id* [simp]:

$framestack\ \text{-to-callstack}\ (snd(snd(state\ \text{-to-jvm-state}\ P\ cs\ s))) =$

cs

⟨proof⟩

lemma *f2c-s2jvm-id'* [simp]:

$framestack\ \text{-to-callstack}\$

$(zip\ (stkss\ P\ cs\ stk)\ (zip\ (locss\ P\ cs\ loc)\ cs)) = cs$

$\langle proof \rangle$

lemma *f2c-append* [*simp*]:
 $framestack\text{-}to\text{-}callstack (frs @ frs') =$
 $(framestack\text{-}to\text{-}callstack frs) @ (framestack\text{-}to\text{-}callstack frs')$
 $\langle proof \rangle$

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* =
 $Entry (('(-Entry'-))$
 $| Node callstack (callstack \times bool) option (('(- -, - '-))$

The empty callstack indicates the exit node

abbreviation *j-node-Exit* :: *j-node* (('(-Exit'-))
where *j-node-Exit* $\equiv (- [], None -)$

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

types *j-edge* = (*j-node* \times *state edge-kind* \times *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* \Rightarrow *j-node* \Rightarrow *state edge-kind* \Rightarrow *j-node* \Rightarrow *bool*
 $(- \vdash - \dashrightarrow -)$
where
JCFG-EntryExit:
 $prog \vdash (-Entry-) \text{ --}(\lambda s. False)\checkmark\text{ --} (-Exit-)$

JCFG-EntryStart:
 $prog = (P, C0, Main) \implies prog \vdash (-Entry-) \text{ --}(\lambda s. True)\checkmark\text{ --} (- [(C0, Main, 0)], None -)$

| *JCFG-ReturnExit*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) C M) ! pc = \text{Return} \rrbracket \\ & \implies \text{prog} \vdash (- [(C, M, pc)], \text{None} -) \dashv\!\! \dashv id \rightarrow (-\text{Exit}-) \end{aligned}$$

| *JCFG-Straight-NoExc*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad \text{instrs-of } (P_{wf}) C M ! pc \in \{\text{Load idx}, \text{Store idx}, \text{Push val}, \text{Pop}, \text{IAdd}, \text{CmpEq}\}; \\ & \quad ek = \uparrow(\lambda s. \text{exec-instr } ((\text{instrs-of } (P_{wf}) C M) ! pc) P s \\ & \quad \quad (\text{length } cs) (\text{stkLength } P C M pc) \text{ arbitrary arbitrary}) \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None} -) \dashv\!\! \dashv ek \rightarrow (- (C, M, \text{Suc } pc) \# cs, \text{None} -) \end{aligned}$$

| *JCFG-New-Normal-Pred*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) C M) ! pc = (\text{New Cl}); \\ & \quad ek = (\lambda(h, \text{stk}, \text{loc}). \text{new-Addr } h \neq \text{None}) \surd \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None} -) \dashv\!\! \dashv ek \rightarrow (- (C, M, pc) \# cs, [(C, M, \text{Suc } pc) \# cs, \text{False}]) -) \end{aligned}$$

| *JCFG-New-Normal-Update*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) C M) ! pc = (\text{New Cl}); \\ & \quad ek = \uparrow(\lambda s. \text{exec-instr } (\text{New Cl}) P s (\text{length } cs) (\text{stkLength } P C M pc) \text{ arbitrary arbitrary}) \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, pc) \# cs, [(C, M, \text{Suc } pc) \# cs, \text{False}]) -) \dashv\!\! \dashv ek \rightarrow (- (C, M, \text{Suc } pc) \# cs, \text{None} -) \end{aligned}$$

| *JCFG-New-Exc-Pred*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) C M) ! pc = (\text{New Cl}); \\ & \quad \text{find-handler-for } P \text{ OutOfMemory } ((C, M, pc) \# cs) = cs'; \\ & \quad ek = (\lambda(h, \text{stk}, \text{loc}). \text{new-Addr } h = \text{None}) \surd \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None} -) \dashv\!\! \dashv ek \rightarrow (- (C, M, pc) \# cs, [(cs', \text{True}]) -) \end{aligned}$$

| *JCFG-New-Exc-Update*:

$$\begin{aligned} & \llbracket \text{prog} = (P, C0, \text{Main}); \\ & \quad (\text{instrs-of } (P_{wf}) C M) ! pc = (\text{New Cl}); \\ & \quad \text{find-handler-for } P \text{ OutOfMemory } ((C, M, pc) \# cs) = (C', M', pc') \# cs'; \\ & \quad ek = \uparrow(\lambda(h, \text{stk}, \text{loc}). \\ & \quad \quad (h, \\ & \quad \quad \quad \text{stk}((\text{length } cs', (\text{stkLength } P C' M' pc') - 1) := \text{Addr } (\text{addr-of-sys-xcpt} \\ & \quad \quad \quad \text{OutOfMemory})), \\ & \quad \quad \quad \text{loc}) \\ & \quad \quad \quad) \rrbracket \\ & \implies \text{prog} \vdash (- (C, M, pc) \# cs, [(C', M', pc') \# cs', \text{True}]) -) \dashv\!\! \dashv ek \rightarrow (- (C', M', pc') \# cs', \text{None} -) \end{aligned}$$

| *JCFG-New-Exc-Exit*:

$$\llbracket \text{prog} = (P, C0, \text{Main});$$

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

| *JCFG-Getfield-Normal-Pred:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = (Getfield Fd Cl);$
 $ek = (\lambda(h, stk, loc). stk(length\ cs, stkLength\ P\ C\ M\ pc - 1) \neq Null) \checkmark]$
 $\implies prog \vdash (- (C, M, pc)\#cs, None -) \dashv\vdash ek \rightarrow (- (C, M, pc)\#cs, [(C, M, Suc\ pc)\#cs, False]) -)$

| *JCFG-Getfield-Normal-Update:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = (Getfield Fd Cl);$
 $ek = \uparrow(\lambda s. exec\text{-instr } (Getfield Fd Cl) P s (length\ cs) (stkLength\ P\ C\ M\ pc)$
 $arbitrary\ arbitrary)]$
 $\implies prog \vdash (- (C, M, pc)\#cs, [(C, M, Suc\ pc)\#cs, False]) -) \dashv\vdash ek \rightarrow (- (C,$
 $M, Suc\ pc)\#cs, None -)$

| *JCFG-Getfield-Exc-Pred:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = (Getfield Fd Cl);$
 $find\text{-handler-for } P NullPointer ((C, M, pc)\#cs) = cs';$
 $ek = (\lambda(h, stk, loc). stk(length\ cs, stkLength\ P\ C\ M\ pc - 1) = Null) \checkmark]$
 $\implies prog \vdash (- (C, M, pc)\#cs, None -) \dashv\vdash ek \rightarrow (- (C, M, pc)\#cs, [(cs', True)] -)$

| *JCFG-Getfield-Exc-Update:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = (Getfield Fd Cl);$
 $find\text{-handler-for } P NullPointer ((C, M, pc)\#cs) = (C', M', pc')\#cs';$
 $ek = \uparrow(\lambda(h, stk, loc).$
 $(h,$
 $stk((length\ cs', (stkLength\ P\ C'\ M'\ pc') - 1) := Addr (addr\text{-of-sys-xcpt}$
 $NullPointer)),$
 $loc)$
 $)]$
 $\implies prog \vdash (- (C, M, pc)\#cs, [(C', M', pc')\#cs', True]) -) \dashv\vdash ek \rightarrow (- (C', M',$
 $pc')\#cs', None -)$

| *JCFG-Getfield-Exc-Exit:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = (Getfield Fd Cl);$
 $find\text{-handler-for } P NullPointer ((C, M, pc)\#cs) = []]$
 $\implies prog \vdash (- (C, M, pc)\#cs, [([], True)] -) \dashv\vdash id \rightarrow (-Exit-)$

| *JCFG-Putfield-Normal-Pred:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = (Putfield Fd Cl);$
 $ek = (\lambda(h, stk, loc). stk(length\ cs, stkLength\ P\ C\ M\ pc - 2) \neq Null) \checkmark]$

$\implies \text{prog} \vdash (- (C, M, pc)\#cs, \text{None} -) -ek \rightarrow (- (C, M, pc)\#cs, [(C, M, \text{Suc } pc)\#cs, \text{False}]) -)$

| *JCFG-Putfield-Normal-Update:*

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{\text{wf}}) C M) ! pc = (\text{Putfield Fd Cl});$
 $ek = \uparrow(\lambda s. \text{exec-instr } (\text{Putfield Fd Cl}) P s (\text{length } cs) (\text{stkLength } P C M pc)$
 $\text{arbitrary arbitrary}) \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc)\#cs, [(C, M, \text{Suc } pc)\#cs, \text{False}]) -) -ek \rightarrow (- (C,$
 $M, \text{Suc } pc)\#cs, \text{None} -)$

| *JCFG-Putfield-Exc-Pred:*

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{\text{wf}}) C M) ! pc = (\text{Putfield Fd Cl});$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc)\#cs) = cs';$
 $ek = (\lambda(h, \text{stk}, \text{loc}). \text{stk}(\text{length } cs, \text{stkLength } P C M pc - 2) = \text{Null})_{\checkmark} \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc)\#cs, \text{None} -) -ek \rightarrow (- (C, M, pc)\#cs, [(cs', \text{True}]) -)$

| *JCFG-Putfield-Exc-Update:*

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{\text{wf}}) C M) ! pc = (\text{Putfield Fd Cl});$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc)\#cs) = (C', M', pc')\#cs';$
 $ek = \uparrow(\lambda(h, \text{stk}, \text{loc}).$
 $(h,$
 $\text{stk}((\text{length } cs', (\text{stkLength } P C' M' pc') - 1) := \text{Addr } (\text{addr-of-sys-xcpt}$
 $\text{NullPointer})),$
 $\text{loc})$
 $) \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc)\#cs, [(C', M', pc')\#cs', \text{True}]) -) -ek \rightarrow (- (C', M',$
 $pc')\#cs', \text{None} -)$

| *JCFG-Putfield-Exc-Exit:*

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{\text{wf}}) C M) ! pc = (\text{Putfield Fd Cl});$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc)\#cs) = [] \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc)\#cs, [([], \text{True}]) -) -\uparrow id \rightarrow (-\text{Exit}-)$

| *JCFG-Checkcast-Normal-Pred:*

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{\text{wf}}) C M) ! pc = (\text{Checkcast Cl});$
 $ek = (\lambda(h, \text{stk}, \text{loc}). \text{cast-ok } (P_{\text{wf}}) Cl h (\text{stk}(\text{length } cs, \text{stkLength } P C M pc -$
 $\text{Suc } 0)))_{\checkmark} \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc)\#cs, \text{None} -) -ek \rightarrow (- (C, M, \text{Suc } pc)\#cs, \text{None} -)$

| *JCFG-Checkcast-Exc-Pred:*

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{\text{wf}}) C M) ! pc = (\text{Checkcast Cl});$
 $\text{find-handler-for } P \text{ ClassCast } ((C, M, pc)\#cs) = cs';$
 $ek = (\lambda(h, \text{stk}, \text{loc}). \neg \text{cast-ok } (P_{\text{wf}}) Cl h (\text{stk}(\text{length } cs, \text{stkLength } P C M pc -$

$Suc\ 0))\checkmark]$
 $\implies prog \vdash (- (C, M, pc)\#cs, None -) -ek \rightarrow (- (C, M, pc)\#cs, [(cs', True)] -)$

| *JCFG-Checkcast-Exc-Update:*

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

| *JCFG-Checkcast-Exc-Exit:*

$\llbracket prog = (P, C0, Main);$
 $(instrs\text{-of}\ (P_{wf})\ C\ M) ! pc = (Checkcast\ Cl);$
 $find\text{-handler}\text{-for}\ P\ ClassCast\ ((C, M, pc)\#cs) = []]$
 $\implies prog \vdash (- (C, M, pc)\#cs, [([], True)] -) -\uparrow id \rightarrow (-Exit-)$

| *JCFG-Invoke-Normal-Pred:*

$\llbracket prog = (P, C0, Main);$
 $(instrs\text{-of}\ (P_{wf})\ C\ M) ! pc = (Invoke\ M2\ n);$
 $cd = length\ cs;$
 $stk\text{-length} = stkLength\ P\ C\ M\ pc;$
 $ek = (\lambda(h, stk, loc).$
 $stk(cd, stk\text{-length} - Suc\ n) \neq Null \wedge$
 $fst(method\ (P_{wf})\ (cname\text{-of}\ h\ (the\ Addr(stk(cd, stk\text{-length} - Suc\ n))))\ M2)$
 $= D$
 \rrbracket
 \implies
 $prog \vdash (- (C, M, pc)\#cs, None -) -ek \rightarrow (- (C, M, pc)\#cs, [(D, M2, 0)\#(C,$
 $M, pc)\#cs, False]) -)$

| *JCFG-Invoke-Normal-Update:*

$\llbracket prog = (P, C0, Main);$
 $(instrs\text{-of}\ (P_{wf})\ C\ M) ! pc = (Invoke\ M2\ n);$
 $stk\text{-length} = stkLength\ P\ C\ M\ pc;$
 $loc\text{-length} = locLength\ P\ D\ M2\ 0;$
 $ek = \uparrow(\lambda s. exec\text{-instr}\ (Invoke\ M2\ n)\ P\ s\ (length\ cs)\ stk\text{-length}\ arbitrary$
 $loc\text{-length})$
 \rrbracket
 $\implies prog \vdash (- (C, M, pc)\#cs, [(D, M2, 0)\#(C, M, pc)\#cs, False]) -) -ek \rightarrow$
 $(- (D, M2, 0)\#(C, M, pc)\#cs, None -)$

| *JCFG-Invoke-Exc-Pred:*

$\llbracket prog = (P, C0, Main);$

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

| *JCFG-Invoke-Exc-Update:*

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

| *JCFG-Invoke-Exc-Exit:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = (Invoke\ M2\ n);$
 $find\text{-handler}\text{-for } P\ \text{NullPointer } ((C, M, pc)\#cs) = []]$
 $\implies prog \vdash (- (C, M, pc)\#cs, [([], True)]) -) -\uparrow id \rightarrow (-Exit-)$

| *JCFG-Return-Update:*

$[prog = (P, C0, Main);$
 $(instrs\text{-of } (P_{wf}) C M) ! pc = Return;$
 $stk\text{-length} = stkLength\ P\ C\ M\ pc;$
 $r\text{-stk-length} = stkLength\ P\ C'\ M'\ (Suc\ pc');$
 $ek = \uparrow(\lambda s. exec\text{-instr } Return\ P\ s\ (Suc\ (length\ cs))\ stk\text{-length}\ r\text{-stk-length}\ arbi\text{-}$
 $trary)]$
 $\implies prog \vdash (- (C, M, pc)\#(C', M', pc')\#cs, None -) -ek \rightarrow (- (C', M', Suc$
 $pc')\#cs, None -)$

| *JCFG-Goto-Update:*

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

| *JCFG-IfFalse-False:*

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

| *JCFG-IfFalse-Next*:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = (\text{IfFalse } b);$
 $ek = (\lambda(h, \text{stk}, \text{loc}). \text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) \neq \text{Bool False} \vee b$
 $= 1) \checkmark \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, \text{Suc } pc) \# cs, \text{None } -)$

| *JCFG-Throw-Pred*:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = \text{Throw};$
 $cd = \text{length } cs;$
 $\text{stk-length} = \text{stkLength } P C M pc;$
 $\exists \text{Exc. find-handler-for } P \text{ Exc } ((C, M, pc) \# cs) = cs';$
 $ek = (\lambda(h, \text{stk}, \text{loc}).$
 $(\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) = \text{Null} \wedge$
 $\text{find-handler-for } P \text{ NullPointer } ((C, M, pc) \# cs) = cs') \vee$
 $(\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) \neq \text{Null} \wedge$
 $\text{find-handler-for } P (\text{cname-of } h (\text{the-Addr}(\text{stk}(cd, \text{stk-length} - 1)))) ((C,$
 $M, pc) \# cs) = cs')$
 $\checkmark \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, \text{None } -) -ek \rightarrow (- (C, M, pc) \# cs, [(cs', \text{True})] -)$

| *JCFG-Throw-Update*:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = \text{Throw};$
 $ek = \uparrow(\lambda(h, \text{stk}, \text{loc}).$
 $(h,$
 $\text{stk}((\text{length } cs', (\text{stkLength } P C' M' pc') - 1) :=$
 $\text{if } (\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1) = \text{Null}) \text{ then}$
 $\text{Addr } (\text{addr-of-sys-xcpt } \text{NullPointer})$
 $\text{else } (\text{stk}(\text{length } cs, \text{stkLength } P C M pc - 1))),$
 $\text{loc})$
 \rrbracket
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, [(C', M', pc') \# cs', \text{True}] -) -ek \rightarrow (- (C', M',$
 $pc') \# cs', \text{None } -)$

| *JCFG-Throw-Exit*:
 $\llbracket \text{prog} = (P, C0, \text{Main});$
 $(\text{instrs-of } (P_{wf}) C M) ! pc = \text{Throw} \rrbracket$
 $\implies \text{prog} \vdash (- (C, M, pc) \# cs, [([], \text{True})] -) -\uparrow id \rightarrow (-\text{Exit})$

5.1.7 CFG properties

lemma *JVMCFG-Exit-no-sourcenode* [*dest*]:
assumes $\text{edge:prog} \vdash (-\text{Exit}) -et \rightarrow n'$
shows *False*
 $\langle \text{proof} \rangle$

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

assumes $edge:prog \vdash n -et \rightarrow (-Entry-)$
shows $False$
 $\langle proof \rangle$

lemma $JVMCFG-EntryD$:
 $\llbracket (P, C, M) \vdash n -et \rightarrow n'; n = (-Entry-) \rrbracket$
 $\implies (n' = (-Exit-) \wedge et = (\lambda s. False)_{\surd}) \vee (n' = (- [(C, M, \theta)], None -) \wedge et = (\lambda s. True)_{\surd})$
 $\langle proof \rangle$

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

lemma $JVMCFG-edge-det$:
 $\llbracket prog \vdash n -et \rightarrow n'; prog \vdash n -et' \rightarrow n' \rrbracket \implies et = et'$
 $\langle proof \rangle$

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 $sourcenode$ $:: j-edge \Rightarrow j-node$
where $sourcenode\ e \equiv fst\ e$

abbreviation $targetnode$ $:: j-edge \Rightarrow j-node$
where $targetnode\ e \equiv snd(snd\ e)$

abbreviation $kind$ $:: j-edge \Rightarrow state\ edge-kind$
where $kind\ e \equiv fst(snd\ e)$

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

fun $valid-callstack$ $:: jvmprog \Rightarrow callstack \Rightarrow bool$
where
 $valid-callstack\ prog\ [] = True$
 $| valid-callstack\ (P, C0, Main)\ [(C, M, pc)] \longleftrightarrow$
 $C = C0 \wedge M = Main \wedge$
 $(P_{\Phi})\ C\ M\ !\ pc \neq None \wedge$
 $(\exists T\ Ts\ mxs\ mxl\ is\ xt. (P_{wf}) \vdash C\ sees\ M:Ts \rightarrow T = (mxs, mxl, is, xt)\ in\ C \wedge pc < length\ is)$

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

fun *valid-node* :: *jvmprog* \Rightarrow *j-node* \Rightarrow *bool*

where

valid-node prog (*-Entry-*) = *True*

$| \text{valid-node prog } (- cs, \text{None } -) \longleftrightarrow \text{valid-callstack prog cs}$
 $| \text{valid-node prog } (- cs, [(cs', xf)] -) \longleftrightarrow$
 $\text{valid-callstack prog cs} \wedge \text{valid-callstack prog cs}' \wedge$
 $(\exists Q. \text{prog} \vdash (- cs, \text{None } -) \text{ --}(Q)\checkmark\rightarrow (- cs, [(cs', xf)] -)) \wedge$
 $(\exists f. \text{prog} \vdash (- cs, [(cs', xf)] -) \text{ --}\uparrow f\rightarrow (- cs', \text{None } -))$

fun *valid-edge* :: *jvmprog* \Rightarrow *j-edge* \Rightarrow *bool*

where

$\text{valid-edge prog } a \longleftrightarrow$
 $(\text{prog} \vdash (\text{sourcenode } a) \text{ --}(kind\ a)\rightarrow (\text{targetnode } a))$
 $\wedge (\text{valid-node prog } (\text{sourcenode } a))$
 $\wedge (\text{valid-node prog } (\text{targetnode } a))$

interpretation *JVM-CFG-Interpret*:

CFG sourcenode targetnode kind valid-edge prog Entry
 $\langle \text{proof} \rangle$

interpretation *JVM-CFGExit-Interpret*:

CFGExit sourcenode targetnode kind valid-edge prog Entry (-Exit-)
 $\langle \text{proof} \rangle$

end

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

5.3 Instantiation of the *CFG-wf* locale

5.3.1 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* \Rightarrow *jinja-var* \Rightarrow *jinja-val*

where

$state\text{-}val (h, stk, loc) (HeapVar a) = Object (h a)$
 $| state\text{-}val (h, stk, loc) (Stk cd idx) = Primitive (stk (cd, idx))$
 $| state\text{-}val (h, stk, loc) (Loc cd idx) = Primitive (loc (cd, idx))$

5.3.2 The *Def* and *Use* sets

inductive-set *Def* :: *wf-jvmprog* \Rightarrow *j-node* \Rightarrow *jinja-var set*

for *P* :: *wf-jvmprog*

and *n* :: *j-node*

where

Def-Load:

$\llbracket n = (- (C, M, pc) \# cs, None -);$
 $instrs\text{-}of (P_{wf}) C M ! pc = Load\ idx;$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc \rrbracket$
 $\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

Def-Store:

$\llbracket n = (- (C, M, pc) \# cs, None -);$
 $instrs\text{-}of (P_{wf}) C M ! pc = Store\ idx;$
 $cd = length\ cs \rrbracket$
 $\Longrightarrow Loc\ cd\ idx \in Def\ P\ n$

Def-Push:

$\llbracket n = (- (C, M, pc) \# cs, None -);$
 $instrs\text{-}of (P_{wf}) C M ! pc = Push\ v;$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc \rrbracket$
 $\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

Def-New-Normal-Stk:

$\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
 $instrs\text{-}of (P_{wf}) C M ! pc = New\ Cl;$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc \rrbracket$
 $\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

Def-New-Normal-Heap:

$\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
 $instrs\text{-}of (P_{wf}) C M ! pc = New\ Cl \rrbracket$
 $\Longrightarrow HeapVar\ a \in Def\ P\ n$

Def-Exc-Stk:

$\llbracket n = (- (C, M, pc) \# cs, [(cs', True)] -);$
 $cs' \neq [];$
 $cd = length\ cs' - 1;$
 $(C', M', pc') = hd\ cs';$
 $i = stkLength\ P\ C'\ M'\ pc' - 1 \rrbracket$
 $\Longrightarrow Stk\ cd\ i \in Def\ P\ n$

| *Def-Getfield-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
instrs-of (P_{wf}) $C M ! pc = \text{Getfield } Fd Cl$;
 $cd = \text{length } cs$;
 $i = \text{stkLength } P C M pc - 1 \rrbracket$
 $\implies \text{Stk } cd i \in \text{Def } P n$

| *Def-Putfield-Heap*:
 $\llbracket n = (- (C, M, pc) \# cs, [(cs', False)] -);$
instrs-of (P_{wf}) $C M ! pc = \text{Putfield } Fd Cl \rrbracket$
 $\implies \text{HeapVar } a \in \text{Def } P n$

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

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

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

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

inductive-set *Use* :: *wf-jvmprog* \Rightarrow *j-node* \Rightarrow *jinja-var set*
for P :: *wf-jvmprog*
and n :: *j-node*
where
Use-Load:
 $\llbracket n = (- (C, M, pc) \# cs, None -);$

$instrs\text{-}of (P_{wf}) C M ! pc = Load\ idx;$
 $cd = length\ cs \]$
 $\implies (Loc\ cd\ idx) \in Use\ P\ n$

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

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

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

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

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

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

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

| *Use-Checkcast-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, x -);$
 $x = \text{None} \vee x = \llbracket (cs', \text{False}) \rrbracket;$
instrs-of $(P_{wf}) C M ! pc = \text{Checkcast } Cl;$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P C M pc - \text{Suc } 0 \rrbracket$
 $\implies \text{Stk } cd i \in \text{Use } P n$

| *Use-Checkcast-Heap*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$
instrs-of $(P_{wf}) C M ! pc = \text{Checkcast } Cl \rrbracket$
 $\implies \text{HeapVar } a \in \text{Use } P n$

| *Use-Invoke-Stk-Pred*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$
instrs-of $(P_{wf}) C M ! pc = \text{Invoke } M' n';$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P C M pc - \text{Suc } n' \rrbracket$
 $\implies \text{Stk } cd i \in \text{Use } P n$

| *Use-Invoke-Heap-Pred*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$
instrs-of $(P_{wf}) C M ! pc = \text{Invoke } M' n' \rrbracket$
 $\implies \text{HeapVar } a \in \text{Use } P n$

| *Use-Invoke-Stk-Update*:
 $\llbracket n = (- (C, M, pc) \# cs, \llbracket (cs', \text{False}) \rrbracket -);$
instrs-of $(P_{wf}) C M ! pc = \text{Invoke } M' n';$
 $cd = \text{length } cs;$
 $i < \text{stkLength } P C M pc;$
 $i \geq \text{stkLength } P C M pc - \text{Suc } n' \rrbracket$
 $\implies \text{Stk } cd i \in \text{Use } P n$

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

| *Use-IAdd-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$
instrs-of $(P_{wf}) C M ! pc = \text{IAdd};$
 $cd = \text{length } cs;$
 $i = \text{stkLength } P C M pc - 1 \vee i = \text{stkLength } P C M pc - 2 \rrbracket$
 $\implies \text{Stk } cd i \in \text{Use } P n$

| *Use-IfFalse-Stk*:
 $\llbracket n = (- (C, M, pc) \# cs, \text{None} -);$

$instrs\text{-}of (P_{wf}) C M ! pc = (IfFalse b);$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc - 1$]
 $\implies Stk\ cd\ i \in Use\ P\ n$

| *Use-CmpEq-Stk*:
[[$n = (- (C, M, pc)\#cs, None -);$
 $instrs\text{-}of (P_{wf}) C M ! pc = CmpEq;$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc - 1 \vee i = stkLength\ P\ C\ M\ pc - 2$]
 $\implies Stk\ cd\ i \in Use\ P\ n$

| *Use-Throw-Stk*:
[[$n = (- (C, M, pc)\#cs, x -);$
 $x = None \vee x = [(cs', True)];$
 $instrs\text{-}of (P_{wf}) C M ! pc = Throw;$
 $cd = length\ cs;$
 $i = stkLength\ P\ C\ M\ pc - 1$]
 $\implies Stk\ cd\ i \in Use\ P\ n$

| *Use-Throw-Heap*:
[[$n = (- (C, M, pc)\#cs, x -);$
 $x = None \vee x = [(cs', True)];$
 $instrs\text{-}of (P_{wf}) C M ! pc = Throw$]
 $\implies HeapVar\ a \in Use\ P\ n$

declare *correct-state-def* [simp del]

lemma *edge-transfer-uses-only-Use*:

[[$valid\text{-}edge (P, C0, Main)\ a;$ $\forall V \in Use\ P (sourcenode\ a).$ $state\text{-}val\ s\ V = state\text{-}val\ s'\ V$]
 $\implies \forall V \in Def\ P (sourcenode\ a).$ $state\text{-}val (BasicDefs.transfer (kind\ a)\ s)\ V =$
 $state\text{-}val (BasicDefs.transfer (kind\ a)\ s')\ V$

<proof>

lemma *CFG-edge-Uses-pred-equal*:

[[$valid\text{-}edge (P, C0, Main)\ a;$
 $pred (kind\ a)\ s;$
 $\forall V \in Use\ P (sourcenode\ a).$ $state\text{-}val\ s\ V = state\text{-}val\ s'\ V$]
 $\implies pred (kind\ a)\ s'$

<proof>

lemma *edge-no-Def-equal*:

[[$valid\text{-}edge (P, C0, Main)\ a;$
 $V \notin Def\ P (sourcenode\ a)$]
 $\implies state\text{-}val (transfer (kind\ a)\ s)\ V = state\text{-}val\ s\ V$

<proof>

interpretation *JVM-CFG-wf: CFG-wf*
sourcenode targetnode kind valid-edge prog (-Entry-)
 Def (fst(prog)) Use (fst(prog)) state-val
 <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* :: *typ*

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" ≠ OutOfMemory
 "C" ≠ ClassCast
 ⟨proof⟩

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

declare *distinct-classes* [simp add]

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

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

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

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

lemma *method-in-EP-is-M*:
 $EP \vdash C \text{ sees } M: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{is}, \text{xt}) \text{ in } D$
 $\implies C = \text{"C"} \wedge$
 $M = \text{"M"} \wedge$
 $Ts = [] \wedge$
 $T = \text{Void} \wedge$
 $m\text{xs} = 1 \wedge$
 $m\text{x}l = 0 \wedge$
 $\text{is} = [\text{Push Unit}, \text{Return}] \wedge$
 $\text{xt} = [] \wedge$
 $D = \text{"C"}$
 ⟨proof⟩

lemma [simp]:
 $\exists T Ts m\text{xs} m\text{x}l \text{ is}. (\exists \text{xt}. EP \vdash \text{"C"} \text{ sees } \text{"M"}: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{is}, \text{xt}) \text{ in } \text{"C"}) \wedge \text{is} \neq []$
 ⟨proof⟩

lemma [simp]:
 $\exists T Ts m\text{xs} m\text{x}l \text{ is}. (\exists \text{xt}. EP \vdash \text{"C"} \text{ sees } \text{"M"}: Ts \rightarrow T = (m\text{xs}, m\text{x}l, \text{is}, \text{xt}) \text{ in } \text{"C"}) \wedge$
 $\text{Suc } 0 < \text{length is}$
 ⟨proof⟩

lemma *C-sees-M-in-EP* [simp]:
 $EP \vdash \text{"C"} \text{ sees } \text{"M"}: [] \rightarrow \text{Void} = (1, 0, [\text{Push Unit}, \text{Return}], []) \text{ in } \text{"C"}$

$\langle \text{proof} \rangle$

lemma *instrs-of-EP-C-M* [simp]:
 instrs-of EP "C" "M" = [Push Unit, Return]
 $\langle \text{proof} \rangle$

lemma *valid-node-in-EP-D*:
 valid-node (Abs-wf-jvmprog (EP, Phi-EP), "C", "M") n
 $\implies n \in \{(-\text{Entry-}), (- [(\text{"C"}, \text{"M"}, 0)], \text{None } -), (- [(\text{"C"}, \text{"M"}, 1)], \text{None } -),$
 $(-\text{Exit-})\}$
 $\langle \text{proof} \rangle$

lemma *EP-C-M-0-valid* [simp]:
 JVM-CFG-Interpret.valid-node (Abs-wf-jvmprog (EP, Phi-EP), "C", "M")
 $(- [(\text{"C"}, \text{"M"}, 0)], \text{None } -)$
 $\langle \text{proof} \rangle$

lemma *EP-C-M-Suc-0-valid* [simp]:
 JVM-CFG-Interpret.valid-node (Abs-wf-jvmprog (EP, Phi-EP), "C", "M")
 $(- [(\text{"C"}, \text{"M"}, \text{Suc } 0)], \text{None } -)$
 $\langle \text{proof} \rangle$

typedef *cfg-wf-prog* =
 $\{P. (\forall n. \text{valid-node } P \ n \longrightarrow$
 $(\exists \text{ as. JVM-CFG-Interpret.path } P \ (-\text{Entry-}) \ \text{as } n) \wedge$
 $(\exists \text{ as. JVM-CFG-Interpret.path } P \ n \ \text{as } (-\text{Exit-}))\}$
 $\langle \text{proof} \rangle$

abbreviation *lift-to-cfg-wf-prog* :: $(\text{jvmprog} \Rightarrow 'a) \Rightarrow (\text{cfg-wf-prog} \Rightarrow 'a)$
 $(-\text{CFG})$
 where $f_{\text{CFG}} \equiv (\lambda P. f \ (\text{Rep-cfg-wf-prog } P))$

6.2 Interpretation of the *Postdomination* locale

interpretation *JVM-CFG-Postdomination*:
 Postdomination sourcenode targetnode kind valid-edge CFG prog Entry (-Exit-)
 $\langle \text{proof} \rangle$

6.3 Interpretation of the *StrongPostdomination* locale

6.3.1 Some helpfull lemmas

lemma *find-handler-for-tl-eq*:

$find_handler_for\ P\ Exc\ cs = (C, M, pc) \# cs' \implies \exists cs''\ pc. cs = cs'' @ [(C, M, pc)] @ cs'$
 $\langle proof \rangle$

lemma *valid-callstack-tl*:

$valid_callstack\ prog\ ((C, M, pc) \# cs) \implies valid_callstack\ prog\ cs$
 $\langle proof \rangle$

lemma *find-handler-Throw-Invoke-pc-in-range*:

$\llbracket cs = (C', M', pc') \# cs';\ valid_callstack\ (P, C0, Main)\ cs;$
 $instrs_of\ (P_{wf})\ C'\ M'!\ pc' = Throw \vee (\exists M''\ n''.\ instrs_of\ (P_{wf})\ C'\ M'!\ pc'$
 $= Invoke\ M''\ n'');$
 $find_handler_for\ P\ Exc\ cs = (C, M, pc) \# cs'' \rrbracket$
 $\implies pc < length\ (instrs_of\ (P_{wf})\ C\ M)$
 $\langle proof \rangle$

6.3.2 Every node has only finitely many successors

lemma *successor-set-finite*:

$JVM_CFG_Interpret.valid_node\ prog\ n$
 $\implies finite\ \{n'.\ \exists a'.\ valid_edge\ prog\ a' \wedge sourcenode\ a' = n \wedge targetnode\ a' = n'\}$
 $\langle proof \rangle$

6.3.3 Interpretation of the locale

interpretation *JVM-CFG-StrongPostdomination*:

$StrongPostdomination\ sourcenode\ targetnode\ kind\ valid_edge_{CFG}\ prog\ Entry\ (-Exit)$
 $\langle proof \rangle$

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* :: $((nat \times nat) \Rightarrow val) \Rightarrow (frame\ list) \Rightarrow ((nat \times nat) \Rightarrow val)$

where

```
update-stk stk frs  $\equiv (\lambda(a, b).$   
  if length frs  $\leq a$  then stk (a, b)  
  else let xs = fst (frs ! (length frs - Suc a))  
    in if length xs  $\leq b$  then stk (a, b) else xs ! (length xs - Suc b)
```

abbreviation *update-loc* :: $((nat \times nat) \Rightarrow val) \Rightarrow (frame\ list) \Rightarrow ((nat \times nat) \Rightarrow val)$

where

```
update-loc loc frs  $\equiv (\lambda(a, b).$   
  if length frs  $\leq a$  then loc (a, b)  
  else let xs = fst (snd (frs ! (length frs - Suc a)))  
    in if length xs  $\leq b$  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*:
 $find_handler\ (P_{wf})\ a\ h\ frs = (xp',h',frs')$
 $\implies find_handler_for\ P\ (cname_of\ h\ a)\ (framestack_to_callstack\ frs) =$
 $framestack_to_callstack\ frs'$
 ⟨proof⟩

lemma *find-handler-nonempty-frs* [simp]:
 $(find_handler\ P\ a\ h\ frs \neq (None, h', []))$
 ⟨proof⟩

lemma *find-handler-heap-egD*:
 $find_handler\ P\ a\ h\ frs = (xp, h', frs') \implies h' = h$
 ⟨proof⟩

lemma *find-handler-frs-decrD*:
 $find_handler\ P\ a\ h\ frs = (xp, h', frs') \implies length\ frs' \leq length\ frs$
 ⟨proof⟩

lemma *find-handler-decrD* [dest]:
 $find_handler\ P\ a\ h\ frs = (xp, h', f\#\frs) \implies False$
 ⟨proof⟩

lemma *find-handler-decrD'* [dest]:
 $\llbracket find_handler\ P\ a\ h\ frs = (xp,h',f\#\frs'); length\ frs = length\ frs' \rrbracket \implies False$
 ⟨proof⟩

lemma *Suc-minus-Suc-Suc* [simp]:
 $b < n - 1 \implies Suc\ (n - Suc\ (Suc\ b)) = n - Suc\ b$
 ⟨proof⟩

lemma *find-handler-loc-fun-eg'*:
 $find_handler\ (P_{wf})\ a\ h$
 $(zip\ (stkss\ P\ cs\ stk)\ (zip\ (locss\ P\ cs\ loc)\ cs)) =$
 (xf, h', frs)
 $\implies update_loc\ loc\ frs = loc$
 ⟨proof⟩

lemma *find-handler-loc-fun-eq*:

$find_handler (P_{wf}) a h (snd(snd(state-to-jvm-state P cs (h,stk,loc)))) = (xf,h',frs)$
 $\implies update_loc loc frs = loc$
 ⟨proof⟩

lemma *find-handler-stk-fun-eq'*:

$\llbracket find_handler (P_{wf}) a h$
 $(zip (stkss P cs stk) (zip (locss P cs loc) cs)) =$
 $(None, h', frs);$
 $cd = length frs - 1;$
 $i = length (fst(hd(frs))) - 1 \rrbracket$
 $\implies update_stk stk frs = stk((cd, i) := Addr a)$
 ⟨proof⟩

lemma *find-handler-stk-fun-eq*:

$find_handler (P_{wf}) a h (snd(snd(state-to-jvm-state P cs (h,stk,loc)))) = (None,h',frs)$
 $\implies update_stk stk frs = stk((length frs - 1, length (fst(hd(frs))) - 1) := Addr$
 $a)$
 ⟨proof⟩

lemma *f2c-emptyD [dest]*:

$framestack-to-callstack frs = [] \implies frs = []$
 ⟨proof⟩

lemma *f2c-emptyD' [dest]*:

$[] = framestack-to-callstack frs \implies frs = []$
 ⟨proof⟩

lemma *correct-state-imp-valid-callstack*:

$\llbracket P, cs \vdash_{BV} s \checkmark; fst (last cs) = C0; fst(snd (last cs)) = Main \rrbracket$
 $\implies valid_callstack (P,C0,Main) cs$
 ⟨proof⟩

declare *correct-state-def [simp del]*

lemma *bool-sym*: $Bool (a = b) = Bool (b = a)$

⟨proof⟩

lemma *find-handler-exec-correct*:

$\llbracket (P_{wf}), (P_{\Phi}) \vdash state-to-jvm-state P cs (h,stk,loc) \checkmark;$
 $(P_{wf}), (P_{\Phi}) \vdash find_handler (P_{wf}) a h$
 $(zip (stkss P cs stk) (zip (locss P cs loc) cs)) \checkmark;$
 $find_handler_for P (cname-of h a) cs = (C', M', pc') \# cs'$
 $\rrbracket \implies$
 $(P_{wf}), (P_{\Phi}) \vdash (None, h,$
 $(stkss (stkLength P C' M' pc')$
 $(\lambda a'. (stk((length cs', stkLength P C' M' pc' - Suc 0) := Addr a)) (length$
 $cs', a'))),$

$locs (locLength P C' M' pc') (\lambda a. loc (length cs', a)), C', M', pc') \#$
 $zip (stkss P cs' stk) (zip (locss P cs' loc) cs') \checkmark$
 <proof>

lemma *locs-rev-stks*:

$x \geq z \implies$
 $locs z$
 $(\lambda b.$
 $if z < b then loc (Suc y, b)$
 $else if b \leq z$
 $then stk (y, x + b - Suc z)$
 $else arbitrary)$
 $@ [stk (y, x - Suc 0)]$
 $=$
 $stk (y, x - Suc (z))$
 $\# rev (take z (stks x (\lambda a. stk(y, a))))$
 <proof>

lemma *locs-invoke-purge*:

$(z::nat) > c \implies$
 $locs l$
 $(\lambda b. if z = c \longrightarrow Q b then loc (c, b) else u b) =$
 $locs l (\lambda a. loc (c, a))$
 <proof>

lemma *nth-rev-equalityI*:

$\llbracket (\lambda b. if z = c \longrightarrow Q b then loc (c, b) else u b) =$
 $locs l (\lambda a. loc (c, a)) \rrbracket$
 $\implies xs = ys$
 <proof>

lemma *length-locss*:

$i < length cs$
 $\implies length (locss P cs loc ! (length cs - Suc i)) =$
 $locLength P (fst(cs ! (length cs - Suc i)))$
 $(fst(snd(cs ! (length cs - Suc i))))$
 $(snd(snd(cs ! (length cs - Suc i))))$
 <proof>

lemma *locss-invoke-purge*:

$z > length cs \implies$
 $locss P cs$
 $(\lambda(a, b). if (a = z \longrightarrow Q b)$
 $then loc (a, b)$
 $else u b)$
 $= locss P cs loc$
 <proof>

lemma *stks-purge'*:

$d \geq b \implies \text{stks } b \ (\lambda x. \text{ if } x = d \text{ then } e \text{ else } \text{stk } x) = \text{stks } b \ \text{stk}$
 $\langle \text{proof} \rangle$

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*:

$\llbracket \text{valid-edge } (P, C0, \text{Main}) \ a;$
 $\text{sourcenode } a = (- (C, M, pc) \# cs, x -);$
 $\text{targetnode } a = (- (C', M', pc') \# cs', x' -);$
 $\text{pred } (\text{kind } a) \ s;$
 $x \neq \text{None} \implies (\exists a\text{-pred.}$
 $\text{sourcenode } a\text{-pred} = (- (C, M, pc) \# cs, \text{None} -) \wedge$
 $\text{targetnode } a\text{-pred} = \text{sourcenode } a \wedge$
 $\text{valid-edge } (P, C0, \text{Main}) \ a\text{-pred} \wedge$
 $\text{pred } (\text{kind } a\text{-pred}) \ s$
 $\rangle;$
 $P, ((C, M, pc) \# cs) \vdash_{BV} s \ \checkmark \rrbracket$
 $\implies P, ((C', M', pc') \# cs') \vdash_{BV} \text{transfer } (\text{kind } a) \ s \ \checkmark$
 $\langle \text{proof} \rangle$

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* \Rightarrow *callstack* \Rightarrow *state* \Rightarrow *callstack* \Rightarrow *state* \Rightarrow *bool*

$(- \vdash \langle -, - \rangle \Rightarrow \langle -, - \rangle)$

where *Step*:

$\llbracket \text{prog} = (P, C0, \text{Main});$
 $P, cs \vdash_{BV} s \ \checkmark;$
 $\text{valid-callstack } \text{prog } cs;$
 $\text{JVMEexec.exec } ((P_{wf}), \text{state-to-jvm-state } P \ cs \ s) = \llbracket (\text{None}, h', \text{frs}') \rrbracket;$
 $cs' = \text{framestack-to-callstack } \text{frs}';$
 $s = (h, \text{stk}, \text{loc});$
 $s' = (h', \text{update-stk } \text{stk } \text{frs}', \text{update-loc } \text{loc } \text{frs}') \rrbracket$
 $\implies \text{prog} \vdash \langle cs, s \rangle \Rightarrow \langle cs', s' \rangle$

abbreviation $identifies :: j\text{-node} \Rightarrow callstack \Rightarrow bool$
where $identifies\ n\ cs \equiv (n = (-\ cs, None -))$

7.2.2 Some more simplification lemmas

lemma $valid\text{-}callstack\text{-}tl$:

$valid\text{-}callstack\ prog\ ((C, M, pc) \# cs) \Longrightarrow valid\text{-}callstack\ prog\ cs$
 $\langle proof \rangle$

lemma $stkss\text{-}cong\ [cong]$:

$\llbracket P = P';$
 $cs = cs';$
 $\bigwedge a\ b. \llbracket a < length\ cs;$
 $b < stkLength\ P\ (fst(cs\ !\ (length\ cs - Suc\ a)))$
 $\quad (fst(snd(cs\ !\ (length\ cs - Suc\ a))))$
 $\quad (snd(snd(cs\ !\ (length\ cs - Suc\ a)))) \rrbracket$
 $\Longrightarrow stk\ (a, b) = stk'\ (a, b) \rrbracket$
 $\Longrightarrow stkss\ P\ cs\ stk = stkss\ P'\ cs'\ stk'$
 $\langle proof \rangle$

lemma $locss\text{-}cong\ [cong]$:

$\llbracket P = P';$
 $cs = cs';$
 $\bigwedge a\ b. \llbracket a < length\ cs;$
 $b < locLength\ P\ (fst(cs\ !\ (length\ cs - Suc\ a)))$
 $\quad (fst(snd(cs\ !\ (length\ cs - Suc\ a))))$
 $\quad (snd(snd(cs\ !\ (length\ cs - Suc\ a)))) \rrbracket$
 $\Longrightarrow loc\ (a, b) = loc'\ (a, b) \rrbracket$
 $\Longrightarrow locss\ P\ cs\ loc = locss\ P'\ cs'\ loc'$
 $\langle proof \rangle$

lemma $hd\text{-}tl\text{-}equalityI$:

$\llbracket length\ xs = length\ ys; hd\ xs = hd\ ys; tl\ xs = tl\ ys \rrbracket \Longrightarrow xs = ys$
 $\langle proof \rangle$

lemma $stkLength\text{-}is\text{-}length\text{-}stk$:

$P_{wf}, P_{\Phi} \vdash (None, h, (stk, loc, C, M, pc) \# frs') \checkmark \Longrightarrow stkLength\ P\ C\ M\ pc =$
 $length\ stk$
 $\langle proof \rangle$

lemma $locLength\text{-}is\text{-}length\text{-}loc$:

$P_{wf}, P_{\Phi} \vdash (None, h, (stk, loc, C, M, pc) \# frs') \checkmark \Longrightarrow locLength\ P\ C\ M\ pc =$
 $length\ loc$
 $\langle proof \rangle$

lemma $correct\text{-}state\text{-}frs\text{-}tlD$:

$(P_{wf}), (P_{\Phi}) \vdash (None, h, a \# frs') \checkmark \Longrightarrow (P_{wf}), (P_{\Phi}) \vdash (None, h, frs') \checkmark$
 $\langle proof \rangle$

lemma *update-stk-Cons* [simp]:

$$\begin{aligned} & \text{stkss } P \text{ (framestack-to-callstack frs')} \text{ (update-stk stk ((stk', loc', C', M', pc') \#} \\ & \text{frs'))} = \\ & \text{stkss } P \text{ (framestack-to-callstack frs')} \text{ (update-stk stk frs')} \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *update-loc-Cons* [simp]:

$$\begin{aligned} & \text{locss } P \text{ (framestack-to-callstack frs')} \text{ (update-loc loc ((stk', loc', C', M', pc') \#} \\ & \text{frs'))} = \\ & \text{locss } P \text{ (framestack-to-callstack frs')} \text{ (update-loc loc frs')} \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *s2j-id*:

$$\begin{aligned} & (P_{wf}), (P_{\Phi}) \vdash (\text{None}, h', \text{frs}') \checkmark \\ \implies & \text{state-to-jvm-state } P \text{ (framestack-to-callstack frs')} \\ & (h, \text{update-stk stk frs}', \text{update-loc loc frs}') = (\text{None}, h, \text{frs}') \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *find-handler-last-cs-eqD*:

$$\begin{aligned} & \llbracket \text{find-handler } P_{wf} \text{ a h frs} = (\text{None}, h', \text{frs}') \rrbracket; \\ & \text{last frs} = (\text{stk}, \text{loc}, C, M, \text{pc}); \\ & \text{last frs}' = (\text{stk}', \text{loc}', C', M', \text{pc}') \rrbracket \\ \implies & C = C' \wedge M = M' \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *exec-last-frs-eq-class*:

$$\begin{aligned} & \llbracket \text{JVMExec.exec } (P_{wf} \text{ None}, h, \text{frs}) = \llbracket (\text{None}, h', \text{frs}') \rrbracket \rrbracket; \\ & \text{last frs} = (\text{stk}, \text{loc}, C, M, \text{pc}); \\ & \text{last frs}' = (\text{stk}', \text{loc}', C', M', \text{pc}'); \\ & \text{frs} \neq \llbracket \rrbracket; \\ & \text{frs}' \neq \llbracket \rrbracket \rrbracket \\ \implies & C = C' \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *exec-last-frs-eq-method*:

$$\begin{aligned} & \llbracket \text{JVMExec.exec } (P_{wf} \text{ None}, h, \text{frs}) = \llbracket (\text{None}, h', \text{frs}') \rrbracket \rrbracket; \\ & \text{last frs} = (\text{stk}, \text{loc}, C, M, \text{pc}); \\ & \text{last frs}' = (\text{stk}', \text{loc}', C', M', \text{pc}'); \\ & \text{frs} \neq \llbracket \rrbracket; \\ & \text{frs}' \neq \llbracket \rrbracket \rrbracket \\ \implies & M = M' \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *valid-callstack-append-last-class*:

$$\begin{aligned} & \text{valid-callstack } (P, C0, \text{Main}) \text{ (cs@\llbracket (C, M, pc) \rrbracket)} \implies C = C0 \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *valid-callstack-append-last-method*:

valid-callstack ($P, C0, Main$) ($cs @ [(C, M, pc)]$) $\implies M = Main$
 ⟨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)
 @ [($stkss$ ($stkLength$ P C M pc) ($\lambda a. stk$ (θ, a)),
 $locss$ ($locLength$ P C M pc) ($\lambda a. loc$ (θ, 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
 ⟨proof⟩

end

theory *Slicing*

imports

Basic/StandardControlDependence
Basic/WeakControlDependence
Basic/ControlDependenceRelations
Basic/SemanticsCFG
Dynamic/DynSlice
StaticIntra/CDepInstantiations
While/SemanticsWellFormed
While/DynamicControlDependences
While/StaticControlDependences
JinjaVM/JVMCFG-wf
JinjaVM/JVMPostdomination
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.