Information Flow Control via Dependency Tracking

Benedikt Nordhoff

March 17, 2025

Abstract

We provide a characterisation of how information is propagated by program executions based on the tracking data and control dependencies within executions themselves. The characterisation might be used for deriving approximative safety properties to be targeted by static analyses or checked at runtime. We utilise a simple yet versatile control flow graph model as a program representation. As our model is not assumed to be finite it can be instantiated for a broad class of programs. The targeted security property is indistinguishable security where executions produce sequences of observations and only non-terminating executions are allowed to drop a tail of those.

A very crude approximation of our characterisation is slicing based on program dependence graphs, which we use as a minimal example and derive a corresponding soundness result.

For further details and applications refer to the authors upcoming dissertation.

Contents

1 Definitions

This section contains all necessary definitions of this development. Section 1.1 contains the structural definition of our program model which includes the security specification as well as abstractions of control flow and data. Executions of our program model are defined in section 1.1.1. Additional well-formedness properties are defined in section 1.1.2. Our security property is defined in section 1.2. Our characterisation of how information is propagated by executions of our program model is defined in section 1.3.5, for which the correctness result can be found in section 2.10. Section 1.3.6 contains an additional approximation of this characterisation whose correctness result can be found in section 2.11.

theory *IFC*
  imports *Main*
begin

1.1 Program Model

Our program model contains all necessary components for the remaining development and consists of:

record $('n, 'var, 'val, 'obs)$ *ifc-problem* =
— A set of nodes representing program locations:
  nodes :: $\langle 'n\ set\rangle$
— An initial node where all executions start:
  entry :: $\langle 'n\rangle$
— A final node where executions can terminate:
  return :: $\langle 'n\rangle$
— An abstraction of control flow in the form of an edge relation:
  edges :: $\langle('n \times 'n)\ set\rangle$
— An abstraction of variables written at program locations:
  writes :: $\langle 'n \Rightarrow 'var\ set\rangle$
— An abstraction of variables read at program locations:
  reads :: $\langle 'n \Rightarrow 'var\ set\rangle$
— A set of variables containing the confidential information in the initial state:
  hvars :: $\langle 'var\ set\rangle$
— A step function on location state pairs:
  step :: $\langle('n \times ('var \Rightarrow 'val)) \Rightarrow ('n \times ('var \Rightarrow 'val))\rangle$
— An attacker model producing observations based on the reached state at certain locations:
  att :: $\langle 'n \rightharpoonup (('var \Rightarrow 'val) \Rightarrow 'obs)\rangle$

We fix a program in the following in order to define the central concepts. The necessary well-formedness assumptions will be made in section 1.1.2.

locale *IFC-def* =
fixes *prob* :: $\langle('n, 'var, 'val, 'obs)$ *ifc-problem*$\rangle$
begin

Some short hands to the components of the program which we will utilise exclusively in the following.

definition *nodes* **where** $\langle$*nodes* = *ifc-problem.nodes prob*$\rangle$
definition *entry* **where** $\langle$*entry* = *ifc-problem.entry prob*$\rangle$
definition *return* **where** $\langle$*return* = *ifc-problem.return prob*$\rangle$
definition *edges* **where** $\langle$*edges* = *ifc-problem.edges prob*$\rangle$
definition *writes* **where** $\langle$*writes* = *ifc-problem.writes prob*$\rangle$
definition *reads* **where** $\langle$*reads* = *ifc-problem.reads prob*$\rangle$
definition *hvars* **where** $\langle$*hvars* = *ifc-problem.hvars prob*$\rangle$
definition *step* **where** $\langle$*step* = *ifc-problem.step prob*$\rangle$
definition *att* **where** $\langle$*att* = *ifc-problem.att prob*$\rangle$

The components of the step function for convenience.

definition *suc* **where** ‹*suc n σ = fst (step (n, σ))›
definition *sem* **where** ‹*sem n σ = snd (step (n, σ))›

lemma *step-suc-sem*: ‹*step (n,σ) = (suc n σ, sem n σ)›* ⟨*proof*⟩

1.1.1 Executions

In order to define what it means for a program to be well-formed, we first require concepts of executions and program paths.

The sequence of nodes visited by the execution corresponding to an input state.

definition *path* **where**
‹*path σ k= fst ((step^^k) (entry,σ))›

The sequence of states visited by the execution corresponding to an input state.

definition *kth-state* (‹-^-› [*111*,*111*] *110*) **where**
‹σ^k = *snd ((step^^k) (entry,σ))›

A predicate asserting that a sequence of nodes is a valid program path according to the control flow graph.

definition *is-path* **where**
‹*is-path π = (∀ n. (π n, π (Suc n)) ∈ edges)›*
end

1.1.2 Well-formed Programs

The following assumptions define our notion of valid programs.

locale *IFC = IFC-def ‹prob›* **for** *prob*:: ‹('*n*, '*var*, '*val*, '*out*) *ifc-problem*› +
assumes *ret-is-node*[*simp*,*intro*]: ‹*return ∈ nodes*›
and *entry-is-node*[*simp*,*intro*]: ‹*entry ∈ nodes*›
and *writes*: ‹⋀ *v n. (∃σ. σ v ≠ sem n σ v) ⟹ v ∈ writes n*›
and *writes-return*: ‹*writes return = {}*›
and *uses-writes*: ‹⋀ *n σ σ'. (∀ v ∈ reads n. σ v = σ' v) ⟹ ∀ v ∈ writes n. sem n σ v = sem n σ' v*›
and *uses-suc*: ‹⋀ *n σ σ'. (∀ v ∈ reads n. σ v = σ' v) ⟹ suc n σ = suc n σ'*›
and *uses-att*: ‹⋀ *n f σ σ'. att n = Some f ⟹ (∀ v ∈ reads n. σ v = σ' v) ⟹ f σ = f σ'*›
and *edges-complete*[*intro*,*simp*]: ‹⋀*m σ. m ∈ nodes ⟹ (m,suc m σ) ∈ edges*›
and *edges-return* : ‹⋀*x. (return,x) ∈ edges ⟹ x = return* ›
and *edges-nodes*: ‹*edges ⊆ nodes × nodes*›
and *reaching-ret*: ‹⋀ *x. x ∈ nodes ⟹ ∃ π n. is-path π ∧ π 0 = x ∧ π n = return*›

1.2 Security

We define our notion of security, which corresponds to what Bohannon et al. [1] refer to as indistinguishable security. In order to do so we require notions of observations made by the attacker, termination and equivalence of input states.

context *IFC-def*
begin

1.2.1 Observations

The observation made at a given index within an execution.

definition *obsp* **where**

‹*obsp* σ k = (*case att*(*path* σ k) *of Some f* ⇒ *Some* (*f* (σ^k)) | *None* ⇒ *None*)›

The indices within a path where an observation is made.

definition *obs-ids* :: ‹(*nat* ⇒ *'n*) ⇒ *nat set*› **where**
‹*obs-ids* π = {*k*. *att* (π *k*) ≠ *None*}›

A predicate relating an observable index to the number of observations made before.

definition *is-kth-obs* :: ‹(*nat* ⇒ *'n*) ⇒ *nat* ⇒ *nat* ⇒ *bool*›**where**
‹*is-kth-obs* π *k i* = (*card* (*obs-ids* π ∩ {..<*i*}) = *k* ∧ *att* (π *i*) ≠ *None*)›

The final sequence of observations made for an execution.

definition *obs* **where**
‹*obs* σ *k* = (*if* (∃ *i*. *is-kth-obs* (*path* σ) *k i*) *then obsp* σ (*THE i*. *is-kth-obs* (*path* σ) *k i*) *else None*)›

Comparability of observations.

definition *obs-prefix* :: ‹(*nat* ⇒ *'obs option*) ⇒ (*nat* ⇒ *'obs option*) ⇒ *bool*› (**infix** ‹$\lesssim$› *50*) **where**
‹*a* $\lesssim$ *b* ≡ ∀ *i*. *a i* ≠ *None* ⟶ *a i* = *b i*›

definition *obs-comp* (**infix** ‹≈› *50*) **where**
‹*a* ≈ *b* ≡ *a* $\lesssim$ *b* ∨ *b* $\lesssim$ *a*›

1.2.2 Low equivalence of input states

definition *restrict* (**infix** ‹↾› *100*) **where**
‹*f*↾*U* = (λ *n*. *if n* ∈ *U then f n else undefined*)›

Two input states are low equivalent if they coincide on the non high variables.

definition *loweq* (**infix** ‹=$_L$› *50*)
where ‹σ =$_L$ σ' = (σ↾(−*hvars*) = σ'↾(−*hvars*))›

1.2.3 Termination

An execution terminates iff it reaches the terminal node at any point.

definition *terminates* **where**
‹*terminates* σ ≡ ∃ *i*. *path* σ *i* = *return*›

1.2.4 Security Property

The fixed program is secure if and only if for all pairs of low equivalent inputs the observation sequences are comparable and if the execution for an input state terminates then the observation sequence is not missing any observations.

definition *secure* **where**
‹*secure* ≡ ∀ σ σ'. σ =$_L$ σ' ⟶ (*obs* σ ≈ *obs* σ' ∧ (*terminates* σ ⟶ *obs* σ' $\lesssim$ *obs* σ))›

1.3 Characterisation of Information Flows

We now define our characterisation of information flows which tracks data and control dependencies within executions. To do so we first require some additional concepts.

1.3.1 Post Dominance

We utilise the post dominance relation in order to define control dependence.

The basic post dominance relation.

definition *is-pd* (**infix** ‹*pd*→› *50*) **where**
‹*y pd*→ *x* ⟷ *x* ∈ *nodes* ∧ (∀ π *n*. *is-path* π ∧ π (*0*::*nat*) = *x* ∧ π *n* = *return* ⟶ (∃ *k*≤*n*. π *k* = *y*))›

The immediate post dominance relation.

definition *is-ipd* (**infix** ‹*ipd*→› *50*)**where**
‹*y ipd*→ *x* ⟷ *x* ≠ *y* ∧ *y pd*→ *x* ∧ (∀ *z*. *z*≠*x* ∧ *z pd*→ *x* ⟶ *z pd*→ *y*)›

definition *ipd* **where**
‹*ipd x* = (*THE y*. *y ipd*→ *x*)›

The post dominance tree.

definition *pdt* **where**
‹*pdt* = {(*x*,*y*). *x*≠*y* ∧ *y pd*→ *x*}›

1.3.2 Control Dependence

An index on an execution path is control dependent upon another if the path does not visit the immediate post domiator of the node reached by the smaller index.

definition *is-cdi* (‹- *cd*⁻→ -› [*51*,*51*,*51*]*50*) **where**
‹*i cd*π→ *k* ⟷ *is-path* π ∧ *k* < *i* ∧ π *i* ≠ *return* ∧ (∀ *j* ∈ {*k*..*i*}. π *j* ≠ *ipd* (π *k*))›

The largest control dependency of an index is the immediate control dependency.

definition *is-icdi* (‹- *icd*⁻→ -› [*51*,*51*,*51*]*50*) **where**
‹*n icd*π→ *n*′ ⟷ *is-path* π ∧ *n cd*π→ *n*′ ∧ (∀ *m* ∈ {*n*′<..<*n*}.¬ *n cd*π→ *m*)›

For the definition of the control slice, which we will define next, we require the uniqueness of the immediate control dependency.

lemma *icd-uniq*: **assumes** ‹*m icd*π→ *n*› ‹ *m icd*π→ *n*′› **shows** ‹*n* = *n*′›
⟨*proof*⟩

1.3.3 Control Slice

We utilise the control slice, that is the sequence of nodes visited by the control dependencies of an index, to match indices between executions.

function *cs*:: ‹(*nat* ⇒ ′*n*) ⇒ *nat* ⇒ ′*n list*› (‹*cs*⁻ -› [*51*,*70*] *71*) **where**
‹*cs*π *n* = (*if* (∃ *m*. *n icd*π→ *m*) *then* (*cs* π (*THE m*. *n icd*π→ *m*))@[π *n*] *else* [π *n*])›
⟨*proof*⟩
termination ‹*cs*› ⟨*proof*⟩

inductive *cs-less* (**infix** ‹≺› *50*) **where**
‹*length xs* < *length ys* ⟹ *take* (*length xs*) *ys* = *xs* ⟹ *xs* ≺ *ys*›

definition *cs-select* (**infix** ‹¡› *50*) **where**
‹π¡*xs* = (*THE k*. *cs*π *k* = *xs*)›

1.3.4 Data Dependence

Data dependence is defined straight forward. An index is data dependent upon another, if the index reads a variable written by the earlier index and the variable in question has not been written by any index in between.

definition *is-ddi* ($\langle$- $dd^{-,-}\!\!\rightarrow$ -$\rangle$ [*51*,*51*,*51*,*51*] *50*) **where**
$\langle n\ dd^{\pi,v}\!\!\rightarrow m \longleftrightarrow$ *is-path* $\pi \wedge m < n \wedge v \in$ *reads* $(\pi\ n) \cap$ (*writes* $(\pi\ m)) \wedge (\forall\ l \in \{m<..<n\}.\ v \notin$ *writes* $(\pi\ l))\rangle$

1.3.5 Characterisation via Critical Paths

With the above we define the set of critical paths which as we will prove characterise the matching points in executions where diverging data is read.

inductive-set *cp* **where**

— Any pair of low equivalent input states and indices where a diverging high variable is first read is critical.

$\langle[\![\sigma =_L \sigma';$
$cs^{path\ \sigma}\ n = cs^{path\ \sigma'}\ n';$
$h \in reads(path\ \sigma\ n);$
$(\sigma^n)\ h \neq (\sigma'^{m'})\ h;$
$\forall\ k<n.\ h \notin writes(path\ \sigma\ k);$
$\forall\ k'<n'.\ h \notin writes(path\ \sigma'\ k')$
$]\!] \Longrightarrow ((\sigma,n),(\sigma',n')) \in cp\rangle$ |

— If from a pair of critical indices in two executions there exist data dependencies from both indices to a pair of matching indices where the variable diverges, the later pair of indices is critical.

$\langle[\![((\sigma,k),(\sigma',k')) \in cp;$
$n\ dd^{path\ \sigma,v}\!\!\rightarrow k;$
$n'\ dd^{path\ \sigma',v}\!\!\rightarrow k';$
$cs^{path\ \sigma}\ n = cs^{path\ \sigma'}\ n';$
$(\sigma^n)\ v \neq (\sigma'^{m'})\ v$
$]\!] \Longrightarrow ((\sigma,n),(\sigma',n')) \in cp\rangle$ |

— If from a pair of critical indices the executions take different branches and one of the critical indices is a control dependency of an index that is data dependency of a matched index where diverging data is read and the variable in question is not written by the other execution after the executions first reached matching indices again, then the later matching pair of indices is critical.

$\langle[\![((\sigma,k),(\sigma',k')) \in cp;$
$n\ dd^{path\ \sigma,v}\!\!\rightarrow l;$
$l\ cd^{path\ \sigma}\!\!\rightarrow k;$
$cs^{path\ \sigma}\ n = cs^{path\ \sigma'}\ n';$
$path\ \sigma\ (Suc\ k) \neq path\ \sigma'\ (Suc\ k');$
$(\sigma^n)\ v \neq (\sigma'^{m'})\ v;$
$\forall j' \in \{(LEAST\ i'.\ k' < i' \wedge (\exists\, i.\ cs^{path\ \sigma}\ i = cs^{path\ \sigma'}\ i'))..<n'\}.\ v \notin writes\ (path\ \sigma'\ j')$
$]\!] \Longrightarrow ((\sigma,n),(\sigma',n')) \in cp\rangle$ |

— The relation is symmetric.

$\langle[\![((\sigma,k),(\sigma',k')) \in cp]\!] \Longrightarrow ((\sigma',k'),(\sigma,k)) \in cp\rangle$

Based on the set of critical paths, the critical observable paths are those that either directly reach observable nodes or are diverging control dependencies of an observable index.

inductive-set *cop* **where**
‹⟦((σ,n),(σ′,n′)) ∈ *cp*;
path σ *n* ∈ *dom att*
⟧ ⟹ ((σ,n),(σ′,n′)) ∈ *cop*› |

‹⟦((σ,k),(σ′,k′)) ∈ *cp*;
n $cd^{path\ \sigma}\rightarrow$ *k*;
path σ (*Suc k*) ≠ *path* σ′ (*Suc k*′);
path σ *n* ∈ *dom att*
⟧ ⟹ ((σ,n),(σ′,k′)) ∈ *cop*›

1.3.6 Approximation via Single Critical Paths

For applications we also define a single execution approximation.

definition *is-dcdi-via* (‹- $dcd^{-,-}\rightarrow$ - *via* - -› [*51*,*51*,*51*,*51*,*51*,*51*] *50*) **where**
‹*n* $dcd^{\pi,v}\rightarrow$ *m via* π′ *m*′ = (*is-path* π ∧ *m* < *n* ∧ (∃ *l*′ *n*′. cs^{π} *m* = $cs^{\pi'}$ *m*′ ∧ cs^{π} *n* = $cs^{\pi'}$ *n*′ ∧ *n*′ $dd^{\pi',v}\rightarrow$ *l*′ ∧ *l*′ $cd^{\pi'}\rightarrow$ *m*′) ∧ (∀ *l* ∈ {*m*..<*n*}. *v*∉ *writes*(π *l*)))›

inductive-set *scp* **where**
‹⟦*h* ∈ *hvars*; *h* ∈ *reads* (*path* σ *n*); (∀ *k*<*n*. *h*∉ *writes*(*path* σ *k*))⟧ ⟹ (*path* σ,*n*) ∈ *scp*› |
‹⟦(π,*m*) ∈ *scp*; *n* $cd^{\pi}\rightarrow$ *m*⟧ ⟹ (π,*n*) ∈ *scp*›|
‹⟦(π,*m*) ∈ *scp*; *n* $dd^{\pi,v}\rightarrow$ *m*⟧ ⟹ (π,*n*) ∈ *scp*›|
‹⟦(π,*m*) ∈ *scp*; (π′,*m*′) ∈ *scp*; *n* $dcd^{\pi,v}\rightarrow$ *m via* π′ *m*′⟧ ⟹ (π,*n*) ∈ *scp*›

inductive-set *scop* **where**
‹⟦(π,*n*) ∈ *scp*; π *n* ∈ *dom att*⟧ ⟹ (π,*n*) ∈ *scop*›

1.3.7 Further Definitions

The following concepts are utilised by the proofs.

inductive *contradicts* (**infix** ‹𝔠› *50*) **where**
‹⟦$cs^{\pi'}$ *k*′ ≺ cs^{π} *k* ; π = *path* σ; π′ = *path* σ′ ; π (*Suc* (π¡$cs^{\pi'}$ *k*′)) ≠ π′ (*Suc k*′)⟧ ⟹ (σ′, *k*′) 𝔠 (σ, *k*)›|
‹⟦$cs^{\pi'}$ *k*′ = cs^{π} *k* ; π = *path* σ; π′ = *path* σ′ ; σ^{k} ↾ (*reads* (π *k*)) ≠ $\sigma'^{k'}$ ↾ (*reads* (π *k*))⟧ ⟹ (σ′,*k*′) 𝔠 (σ,*k*)›

definition *path-shift* (**infixl** ‹«› *51*) **where**
[*simp*]: ‹π«*m* = (λ *n*. π (*m*+*n*))›

definition *path-append* :: ‹(*nat* ⇒ 'n) ⇒ *nat* ⇒ (*nat* ⇒ 'n) ⇒ (*nat* ⇒ 'n)› (‹- @⁻ -› [*0*,*0*,*999*] *51*) **where**
[*simp*]: ‹π @^{m} π′ = (λ*n*.(*if n* ≤ *m then* π *n else* π′ (*n*−*m*)))›

definition *eq-up-to* :: ‹(*nat* ⇒ 'n) ⇒ *nat* ⇒ (*nat* ⇒ 'n) ⇒ *bool*› (‹- =_ -› [*55*,*55*,*55*] *50*) **where**
‹π =$_k$ π′ = (∀ *i* ≤ *k*. π *i* = π′ *i*)›

end

2 Proofs

2.1 Miscellaneous Facts

lemma *option-neq-cases*: **assumes** ‹*x* ≠ *y*› **obtains** (*none1*) *a* **where** ‹*x* = *None*› ‹*y* = *Some a*› | (*none2*) *a* **where** ‹*x* = *Some a*› ‹*y* = *None*› | (*some*) *a b* **where** ‹*x* = *Some a*› ‹*y* = *Some b*› ‹*a* ≠ *b*› ⟨*proof*⟩

lemmas *nat-sym-cases*[*case-names less sym eq*] = *linorder-less-wlog*

lemma *mod-bound-instance*: **assumes** ‹$j < (i::nat)$› **obtains** j' **where** ‹$k < j'$› **and** ‹$j' \ mod \ i = j$› ⟨*proof*⟩

lemma *list-neq-prefix-cases*: **assumes** ‹$ls \neq ls'$› **and** ‹$ls \neq Nil$› **and** ‹$ls' \neq Nil$›
obtains (*diverge*) *xs x x' ys ys'* **where** ‹$ls = xs@[x]@ys$› ‹$ls' = xs@[x']@ys'$› ‹$x \neq x'$› |
(*prefix1*) *xs* **where** ‹$ls = ls'@xs$› **and** ‹$xs \neq Nil$› |
(*prefix2*) *xs* **where** ‹$ls@xs = ls'$› **and** ‹$xs \neq Nil$›
⟨*proof*⟩

lemma *three-cases*: **assumes** ‹$A \vee B \vee C$› **obtains** ‹A› | ‹B› | ‹C› ⟨*proof*⟩

lemma *insort-greater*: ‹$\forall \ x \in set \ ls. \ x < y \Longrightarrow insort \ y \ ls = ls@[y]$› ⟨*proof*⟩

lemma *insort-append-first*: **assumes** ‹$\forall \ y \in set \ ys. \ x \leq y$› **shows** ‹$insort \ x \ (xs@ys) = insort \ x \ xs \ @ \ ys$›
⟨*proof*⟩

lemma *sorted-list-of-set-append*: **assumes** ‹*finite xs*› ‹*finite ys*› ‹$\forall \ x \in xs. \forall \ y \in ys. \ x < y$› **shows** ‹*sorted-list-of-set* $(xs \cup ys) =$ *sorted-list-of-set xs* @ (*sorted-list-of-set ys*)›
⟨*proof*⟩

lemma *filter-insort*: ‹$sorted \ xs \Longrightarrow filter \ P \ (insort \ x \ xs) = (if \ P \ x \ then \ insort \ x \ (filter \ P \ xs) \ else \ filter \ P \ xs)$›
⟨*proof*⟩

lemma *filter-sorted-list-of-set*: **assumes** ‹*finite xs*› **shows** ‹*filter P* (*sorted-list-of-set xs*) = *sorted-list-of-set* {*x* ∈ *xs*. *P x*}› ⟨*proof*⟩

lemma *unbounded-nat-set-infinite*: **assumes** ‹$\forall \ (i::nat). \ \exists \ j \geq i. \ j \in A$› **shows** ‹$\neg \ finite \ A$› ⟨*proof*⟩

lemma *infinite-ascending*: **assumes** *nf*: ‹$\neg \ finite \ (A::nat \ set)$› **obtains** f **where** ‹$range \ f = A$› ‹$\forall \ i. \ f \ i < f \ (Suc \ i)$› ⟨*proof*⟩

lemma *mono-ge-id*: ‹$\forall \ i. \ f \ i < f \ (Suc \ i) \Longrightarrow i \leq f \ i$›
⟨*proof*⟩

lemma *insort-map-mono*: **assumes** *mono*: ‹$\forall \ n \ m. \ n < m \longrightarrow f \ n < f \ m$› **shows** ‹$map \ f \ (insort \ n \ ns) = insort \ (f \ n) \ (map \ f \ ns)$›
⟨*proof*⟩

lemma *sorted-list-of-set-map-mono*: **assumes** *mono*: ‹$\forall \ n \ m. \ n < m \longrightarrow f \ n < f \ m$› **and** *fin*: ‹*finite A*›
shows ‹*map f* (*sorted-list-of-set A*) = *sorted-list-of-set* (*f*‘*A*)›
⟨*proof*⟩

lemma *GreatestIB*:
fixes n :: ‹*nat*› **and** P
assumes *a*:‹$\exists \ k \leq n. \ P \ k$›
shows *GreatestBI*: ‹$P \ (GREATEST \ k. \ k \leq n \wedge P \ k)$› **and** *GreatestB*: ‹$(GREATEST \ k. \ k \leq n \wedge P \ k) \leq n$›
⟨*proof*⟩

lemma *GreatestB-le*:
fixes n :: ‹*nat*›
assumes ‹$x \leq n$› **and** ‹$P \ x$›
shows ‹$x \leq (GREATEST \ k. \ k \leq n \wedge P \ k)$›
⟨*proof*⟩

lemma *LeastBI-ex*: **assumes** ‹$\exists\, k \leq n.\ P\ k$› **shows** ‹$P\ (LEAST\ k{::}'c{::}wellorder.\ P\ k)$› **and** ‹$(LEAST\ k.\ P\ k) \leq n$›
⟨*proof*⟩

lemma *allB-atLeastLessThan-lower*: **assumes** ‹$(i{::}nat) \leq j$› ‹$\forall\ x{\in}\{i..{<}n\}.\ P\ x$› **shows** ‹$\forall\ x{\in}\{j..{<}n\}.\ P\ x$›
⟨*proof*⟩

2.2 Facts about Paths

context *IFC*
begin

lemma *path0*: ‹$path\ \sigma\ 0 = entry$› ⟨*proof*⟩

lemma *path-in-nodes*[*intro*]: ‹$path\ \sigma\ k \in nodes$› ⟨*proof*⟩

lemma *path-is-path*[*simp*]: ‹$is\text{-}path\ (path\ \sigma)$› ⟨*proof*⟩

lemma *term-path-stable*: **assumes** ‹$is\text{-}path\ \pi$› ‹$\pi\ i = return$› **and** *le*: ‹$i \leq j$› **shows** ‹$\pi\ j = return$›
⟨*proof*⟩

lemma *path-path-shift*: **assumes** ‹$is\text{-}path\ \pi$› **shows** ‹$is\text{-}path\ (\pi{\ll}m)$›
⟨*proof*⟩

lemma *path-cons*: **assumes** ‹$is\text{-}path\ \pi$› ‹$is\text{-}path\ \pi'$› ‹$\pi\ m = \pi'\ 0$› **shows** ‹$is\text{-}path\ (\pi\ @^m\ \pi')$›
⟨*proof*⟩

lemma *is-path-loop*: **assumes** ‹$is\text{-}path\ \pi$› ‹$0 < i$› ‹$\pi\ i = \pi\ 0$› **shows** ‹$is\text{-}path\ (\lambda\ n.\ \pi\ (n\ mod\ i))$› ⟨*proof*⟩

lemma *path-nodes*: ‹$is\text{-}path\ \pi \Longrightarrow \pi\ k \in nodes$› ⟨*proof*⟩

lemma *direct-path-return'*: **assumes** ‹$is\text{-}path\ \pi$› ‹$\pi\ 0 = x$› ‹$x \neq return$› ‹$\pi\ n = return$›
obtains $\pi'\ n'$ **where** ‹$is\text{-}path\ \pi'$› ‹$\pi'\ 0 = x$› ‹$\pi'\ n' = return$› ‹$\forall\ i{>}\ 0.\ \pi'\ i \neq x$›
⟨*proof*⟩

lemma *direct-path-return*: **assumes** ‹$x \in nodes$› ‹$x \neq return$›
obtains $\pi\ n$ **where** ‹$is\text{-}path\ \pi$› ‹$\pi\ 0 = x$› ‹$\pi\ n = return$› ‹$\forall\ i{>}\ 0.\ \pi\ i \neq x$›
⟨*proof*⟩

lemma *path-append-eq-up-to*: ‹$(\pi\ @^k\ \pi') =_k \pi$› ⟨*proof*⟩

lemma *eq-up-to-le*: **assumes** ‹$k \leq n$› ‹$\pi =_n \pi'$› **shows** ‹$\pi =_k \pi'$› ⟨*proof*⟩

lemma *eq-up-to-refl*: **shows** ‹$\pi =_k \pi$› ⟨*proof*⟩

lemma *eq-up-to-sym*: **assumes** ‹$\pi =_k \pi'$› **shows** ‹$\pi' =_k \pi$› ⟨*proof*⟩

lemma *eq-up-to-apply*: **assumes** ‹$\pi =_k \pi'$› ‹$j \leq k$› **shows** ‹$\pi\ j = \pi'\ j$› ⟨*proof*⟩

lemma *path-swap-ret*: **assumes** ‹$is\text{-}path\ \pi$› **obtains** $\pi'\ n$ **where** ‹$is\text{-}path\ \pi'$› ‹$\pi =_k \pi'$› ‹$\pi'\ n = return$›
⟨*proof*⟩

lemma *path-suc*: ‹$path\ \sigma\ (Suc\ k) = fst\ (step\ (path\ \sigma\ k,\ \sigma^k))$› ⟨*proof*⟩

lemma *kth-state-suc*: ‹$\sigma^{Suc\ k} = snd\ (step\ (path\ \sigma\ k,\ \sigma^k))$› ⟨*proof*⟩

2.3 Facts about Post Dominators

lemma *pd-trans*: **assumes** *1*: ‹*y pd*→ *x*› **and** *2*: ‹*z pd*→*y*› **shows** ‹*z pd*→*x*›
⟨*proof*⟩

lemma *pd-path*: **assumes** ‹*y pd*→ *x*›
obtains π *n k* **where** ‹*is-path* π› **and** ‹π *0* = *x*› **and** ‹π *n* = *return*› **and** ‹π *k* = *y*› **and** ‹*k* ≤ *n*›
⟨*proof*⟩

lemma *pd-antisym*: **assumes** *xpdy*: ‹*x pd*→ *y*› **and** *ypdx*: ‹*y pd*→ *x*› **shows** ‹*x* = *y*›
⟨*proof*⟩

lemma *pd-refl*[*simp*]: ‹*x* ∈ *nodes* ⟹ *x pd*→ *x*› ⟨*proof*⟩

lemma *pdt-trans-in-pdt*: ‹(*x*,*y*) ∈ *pdt*$^+$ ⟹ (*x*,*y*) ∈ *pdt*›
⟨*proof*⟩

lemma *pdt-trancl-pdt*: ‹*pdt*$^+$ = *pdt*› ⟨*proof*⟩

lemma *trans-pdt*: ‹*trans pdt*› ⟨*proof*⟩

definition [*simp*]: ‹*pdt-inv* = *pdt*$^{-1}$›

lemma *wf-pdt-inv*: ‹*wf* (*pdt-inv*)› ⟨*proof*⟩

lemma *return-pd*: **assumes** ‹*x* ∈ *nodes*› **shows** ‹*return pd*→ *x*› ⟨*proof*⟩

lemma *pd-total*: **assumes** *xz*: ‹*x pd*→ *z*› **and** *yz*: ‹*y pd*→ *z*› **shows** ‹*x pd*→ *y* ∨ *y pd*→*x*›
⟨*proof*⟩

lemma *pds-finite*: ‹*finite* {*y* . (*x*,*y*) ∈ *pdt*}› ⟨*proof*⟩

lemma *ipd-exists*: **assumes** *node*: ‹*x* ∈ *nodes*› **and** *not-ret*: ‹*x*≠*return*› **shows** ‹∃ *y*. *y ipd*→ *x*›
⟨*proof*⟩

lemma *ipd-unique*: **assumes** *yipd*: ‹*y ipd*→ *x*› **and** *y'ipd*: ‹*y'* *ipd*→ *x*› **shows** ‹*y* = *y'*›
⟨*proof*⟩

lemma *ipd-is-ipd*: **assumes** ‹*x* ∈ *nodes*› **and** ‹*x*≠*return*› **shows** ‹*ipd x ipd*→ *x*› ⟨*proof*⟩

lemma *is-ipd-in-pdt*: ‹*y ipd*→ *x* ⟹ (*x*,*y*) ∈ *pdt*› ⟨*proof*⟩

lemma *ipd-in-pdt*: ‹*x* ∈ *nodes* ⟹ *x*≠*return* ⟹ (*x*,*ipd x*) ∈ *pdt*› ⟨*proof*⟩

lemma *no-pd-path*: **assumes** ‹*x* ∈ *nodes*› **and** ‹¬ *y pd*→ *x*›
obtains π *n* **where** ‹*is-path* π› **and** ‹π *0* = *x*› **and** ‹π *n* = *return*› **and** ‹∀ *k* ≤ *n*. π *k* ≠ *y*›
⟨*proof*⟩

lemma *pd-pd-ipd*: **assumes** ‹*x* ∈ *nodes*› ‹*x*≠*return*› ‹*y*≠*x*› ‹*y pd*→ *x*› **shows** ‹*y pd*→ *ipd x*›
⟨*proof*⟩

lemma *pd-nodes*: **assumes** ‹*y pd*→ *x*› **shows** *pd-node1*: ‹*y* ∈ *nodes*› **and** *pd-node2*: ‹*x* ∈ *nodes*›
⟨*proof*⟩

lemma *pd-ret-is-ret*: ‹*x pd*→ *return* ⟹ *x* = *return*› ⟨*proof*⟩

lemma *ret-path-none-pd*: **assumes** ‹$x \in nodes$› ‹$x \neq return$› **obtains** π n **where** ‹*is-path* π› ‹$\pi\ 0 = x$› ‹$\pi\ n = return$› ‹$\forall\ i{>}0.\ \neg\ x\ pd{\rightarrow}\ \pi\ i$›
⟨*proof*⟩

lemma *path-pd-ipd0'*: **assumes** ‹*is-path* π› **and** ‹$\pi\ n \neq return$› ‹$\pi\ n \neq \pi\ 0$› **and** ‹$\pi\ n\ pd{\rightarrow}\ \pi\ 0$› **obtains** k **where** ‹$k \leq n$› **and** ‹$\pi\ k = ipd(\pi\ 0)$›
⟨*proof*⟩

lemma *path-pd-ipd0*: **assumes** ‹*is-path* π› **and** ‹$\pi\ 0 \neq return$› ‹$\pi\ n \neq \pi\ 0$› **and** ‹$\pi\ n\ pd{\rightarrow}\ \pi\ 0$› **obtains** k **where** ‹$k \leq n$› **and** ‹$\pi\ k = ipd(\pi\ 0)$›
⟨*proof*⟩

lemma *path-pd-ipd*: **assumes** ‹*is-path* π› **and** ‹$\pi\ k \neq return$› ‹$\pi\ n \neq \pi\ k$› **and** ‹$\pi\ n\ pd{\rightarrow}\ \pi\ k$› **and** *kn*: ‹$k < n$› **obtains** l **where** ‹$k < l$› **and** ‹$l \leq n$› **and** ‹$\pi\ l = ipd(\pi\ k)$›
⟨*proof*⟩

lemma *path-ret-ipd*: **assumes** ‹*is-path* π› **and** ‹$\pi\ k \neq return$› ‹$\pi\ n = return$› **obtains** l **where** ‹$k < l$› **and** ‹$l \leq n$› **and** ‹$\pi\ l = ipd(\pi\ k)$›
⟨*proof*⟩

lemma *pd-intro*: **assumes** ‹$l\ pd{\rightarrow}\ k$› ‹*is-path* π› ‹$\pi\ 0 = k$› ‹$\pi\ n = return$› **obtains** i **where** ‹$i \leq n$› ‹$\pi\ i = l$› ⟨*proof*⟩

lemma *path-pd-pd0*: **assumes** *path*: ‹*is-path* π› **and** *lpdn*: ‹$\pi\ l\ pd{\rightarrow}\ n$› **and** *npd0*: ‹$n\ pd{\rightarrow}\ \pi\ 0$› **obtains** k **where** ‹$k \leq l$› ‹$\pi\ k = n$›
⟨*proof*⟩

2.4 Facts about Control Dependencies

lemma *icd-imp-cd*: ‹$n\ icd^{\pi}{\rightarrow}\ k \Longrightarrow n\ cd^{\pi}{\rightarrow}\ k$› ⟨*proof*⟩

lemma *ipd-impl-not-cd*: **assumes** ‹$j \in \{k..i\}$› **and** ‹$\pi\ j = ipd\ (\pi\ k)$› **shows** ‹$\neg\ i\ cd^{\pi}{\rightarrow}\ k$›
⟨*proof*⟩

lemma *cd-not-ret*: **assumes** ‹$i\ cd^{\pi}{\rightarrow}\ k$ › **shows** ‹$\pi\ k \neq return$› ⟨*proof*⟩

lemma *cd-path-shift*: **assumes** ‹$j \leq k$› ‹*is-path* π › **shows** ‹$(i\ cd^{\pi}{\rightarrow}\ k) = (i - j\ cd^{\pi\ll j}{\rightarrow}\ k{-}j)$› ⟨*proof*⟩

lemma *cd-path-shift0*: **assumes** ‹*is-path* π› **shows** ‹$(i\ cd^{\pi}{\rightarrow}\ k) = (i{-}k\ cd^{\pi\ll k}{\rightarrow}0)$›
⟨*proof*⟩

lemma *icd-path-shift*: **assumes** ‹$l \leq k$› ‹*is-path* π› **shows** ‹$(i\ icd^{\pi}{\rightarrow}\ k) = (i - l\ icd^{\pi\ll l}{\rightarrow}\ k - l)$›
⟨*proof*⟩

lemma *icd-path-shift0*: **assumes** ‹*is-path* π› **shows** ‹$(i\ icd^{\pi}{\rightarrow}\ k) = (i{-}k\ icd^{\pi\ll k}{\rightarrow}0)$›
⟨*proof*⟩

lemma *cdi-path-swap*: **assumes** ‹*is-path* π'› ‹$j\ cd^{\pi}{\rightarrow}k$› ‹$\pi =_j \pi'$› **shows** ‹$j\ cd^{\pi'}{\rightarrow}k$› ⟨*proof*⟩

lemma *cdi-path-swap-le*: **assumes** ‹*is-path* π'› ‹$j\ cd^{\pi}{\rightarrow}k$› ‹$\pi =_n \pi'$› ‹$j \leq n$› **shows** ‹$j\ cd^{\pi'}{\rightarrow}k$› ⟨*proof*⟩

lemma *not-cd-impl-ipd*: **assumes** ‹*is-path* π› **and** ‹$k < i$› **and** ‹$\neg\ i\ cd^{\pi}{\rightarrow}\ k$› **and** ‹$\pi\ i \neq return$› **obtains** j **where** ‹$j \in \{k..i\}$› **and** ‹$\pi\ j = ipd\ (\pi\ k)$›
⟨*proof*⟩

lemma *icd-is-the-icd*: **assumes** ‹i $icd^{\pi}\!\!\rightarrow$ k› **shows** ‹$k = (THE\ k.\ i\ icd^{\pi}\!\!\rightarrow k)$› ⟨*proof*⟩

lemma *all-ipd-imp-ret*: **assumes** ‹*is-path* π› **and** ‹$\forall\ i.\ \pi\ i \neq return \longrightarrow (\exists\ j{>}i.\ \pi\ j = ipd\ (\pi\ i))$› **shows** ‹$\exists j.\ \pi\ j = return$›
⟨*proof*⟩

lemma *loop-has-cd*: **assumes** ‹*is-path* π› ‹$0 < i$› ‹$\pi\ i = \pi\ 0$› ‹$\pi\ 0 \neq return$› **shows** ‹$\exists\ k < i.\ i\ cd^{\pi}\!\!\rightarrow k$›
⟨*proof*⟩

lemma *loop-has-cd′*: **assumes** ‹*is-path* π› ‹$j < i$› ‹$\pi\ i = \pi\ j$› ‹$\pi\ j \neq return$› **shows** ‹$\exists\ k \in \{j..{<}i\}.\ i\ cd^{\pi}\!\!\rightarrow k$›
⟨*proof*⟩

lemma *claim″*: **assumes** *path*π: ‹*is-path* π› **and** *path*π': ‹*is-path* π'›
and πi: ‹$\pi\ i = \pi'\ i'$› **and** πj: ‹$\pi\ j = \pi'\ j'$›
and *not-cd*: ‹$\forall\ k.\ \neg\ j\ cd^{\pi}\!\!\rightarrow k$› ‹$\forall\ k.\ \neg\ i'\ cd^{\pi'}\!\!\rightarrow k$›
and *nret*: ‹$\pi\ i \neq return$›
and *ilj*: ‹$i < j$›
shows ‹$i' < j'$› ⟨*proof*⟩

lemma *other-claim′*: **assumes** *path*: ‹*is-path* π› **and** *eq*: ‹$\pi\ i = \pi\ j$› **and** ‹$\pi\ i \neq return$›
and *icd*: ‹$\forall\ k.\ \neg\ i\ cd^{\pi}\!\!\rightarrow k$› **and** ‹$\forall\ k.\ \neg\ j\ cd^{\pi}\!\!\rightarrow k$› **shows** ‹$i = j$›
⟨*proof*⟩

lemma *icd-no-cd-path-shift*: **assumes** ‹$i\ icd^{\pi}\!\!\rightarrow 0$› **shows** ‹$(\forall\ k.\ \neg\ i - 1\ cd^{\pi\langle 1}\!\!\rightarrow k)$›
⟨*proof*⟩

lemma *claim′*: **assumes** *path*π: ‹*is-path* π› **and** *path*π': ‹*is-path* π'› **and**
πi: ‹$\pi\ i = \pi'\ i'$› **and** πj: ‹$\pi\ j = \pi'\ j'$› **and** *not-cd*:
‹$i\ icd^{\pi}\!\!\rightarrow 0$› ‹$j\ icd^{\pi}\!\!\rightarrow 0$›
‹$i'\ icd^{\pi'}\!\!\rightarrow 0$› ‹$j'\ icd^{\pi'}\!\!\rightarrow 0$›
and *ilj*: ‹$i < j$›
and *nret*: ‹$\pi\ i \neq return$›
shows ‹$i' < j'$›
⟨*proof*⟩

lemma *other-claim*: **assumes** *path*: ‹*is-path* π› **and** *eq*: ‹$\pi\ i = \pi\ j$› **and** ‹$\pi\ i \neq return$›
and *icd*: ‹$i\ icd^{\pi}\!\!\rightarrow 0$› **and** ‹$j\ icd^{\pi}\!\!\rightarrow 0$› **shows** ‹$i = j$› ⟨*proof*⟩

lemma *cd-trans0*: **assumes** ‹$j\ cd^{\pi}\!\!\rightarrow 0$› **and** ‹$k\ cd^{\pi}\!\!\rightarrow j$› **shows** ‹$k\ cd^{\pi}\!\!\rightarrow 0$› ⟨*proof*⟩

lemma *cd-trans*: **assumes** ‹$j\ cd^{\pi}\!\!\rightarrow i$› **and** ‹$k\ cd^{\pi}\!\!\rightarrow j$› **shows** ‹$k\ cd^{\pi}\!\!\rightarrow i$› ⟨*proof*⟩

lemma *excd-impl-exicd*: **assumes** ‹$\exists\ k.\ i\ cd^{\pi}\!\!\rightarrow k$› **shows** ‹$\exists\ k.\ i\ icd^{\pi}\!\!\rightarrow k$›
⟨*proof*⟩

lemma *cd-split*: **assumes** ‹$i\ cd^{\pi}\!\!\rightarrow k$› **and** ‹$\neg\ i\ icd^{\pi}\!\!\rightarrow k$› **obtains** m **where** ‹$i\ icd^{\pi}\!\!\rightarrow m$› **and** ‹$m\ cd^{\pi}\!\!\rightarrow k$›
⟨*proof*⟩

lemma *cd-induct*[*consumes 1*, *case-names base IS*]: **assumes** *prem*: ‹$i\ cd^{\pi}\!\!\rightarrow k$› **and** *base*: ‹$\bigwedge i.\ i\ icd^{\pi}\!\!\rightarrow k \Longrightarrow P\ i$›
and *IH*: ‹$\bigwedge k'\ i'.\ k'\ cd^{\pi}\!\!\rightarrow k \Longrightarrow P\ k' \Longrightarrow i'\ icd^{\pi}\!\!\rightarrow k' \Longrightarrow P\ i'$› **shows** ‹$P\ i$›
⟨*proof*⟩

lemma *cdi-prefix*: ‹$n\ cd^{\pi}\!\!\rightarrow m \Longrightarrow m < n' \Longrightarrow n' \leq n \Longrightarrow n'\ cd^{\pi}\!\!\rightarrow m$› ⟨*proof*⟩

lemma *cr-wn'*: **assumes** *1*: ‹n $cd^{\pi}\!\!\rightarrow$ m› **and** *nc*: ‹$\neg$ m' $cd^{\pi}\!\!\rightarrow$ m› **and** *3*: ‹$m < m'$› **shows** ‹$n < m'$›
⟨*proof*⟩

lemma *cr-wn''*: **assumes** ‹i $cd^{\pi}\!\!\rightarrow$ m› **and** ‹j $cd^{\pi}\!\!\rightarrow$ n› **and** ‹$\neg$ m $cd^{\pi}\!\!\rightarrow$ n› **and** ‹$i \leq j$› **shows** ‹$m \leq n$›
⟨*proof*⟩

lemma *ret-no-cd*: **assumes** ‹π n = *return*› **shows** ‹$\neg$ n $cd^{\pi}\!\!\rightarrow$ k› ⟨*proof*⟩

lemma *ipd-not-self*: **assumes** ‹$x \in$ *nodes*› ‹$x \neq$ *return*› **shows** ‹$x \neq$ *ipd* x› ⟨*proof*⟩

lemma *icd-cs*: **assumes** ‹l $icd^{\pi}\!\!\rightarrow k$› **shows** ‹$cs^{\pi}$ l = cs^{π} k @ [π l]›
⟨*proof*⟩

lemma *cd-not-pd*: **assumes** ‹l $cd^{\pi}\!\!\rightarrow$ k› ‹π $l \neq \pi$ k› **shows** ‹$\neg$ π l *pd*$\rightarrow$ π k› ⟨*proof*⟩

lemma *cd-ipd-is-cd*: **assumes** ‹$k<m$› ‹π m = *ipd* (π k)› ‹$\forall$ $n \in \{k..<m\}$. π $n \neq$ *ipd* (π k)› **and** *mcdj*: ‹m $cd^{\pi}\!\!\rightarrow$ j› **shows** ‹k $cd^{\pi}\!\!\rightarrow$ j› ⟨*proof*⟩

lemma *ipd-pd-cd0*: **assumes** *lcd*: ‹n $cd^{\pi}\!\!\rightarrow$ *0*› **shows** ‹*ipd* (π *0*) *pd*$\rightarrow$ (π n)›
⟨*proof*⟩

lemma *ipd-pd-cd*: **assumes** *lcd*: ‹l $cd^{\pi}\!\!\rightarrow$ k› **shows** ‹*ipd* (π k) *pd*$\rightarrow$ (π l)›
⟨*proof*⟩

lemma *cd-is-cd-ipd*: **assumes** *km*: ‹$k<m$› **and** *ipd*: ‹π m = *ipd* (π k)› ‹$\forall$ $n \in \{k..<m\}$. π $n \neq$ *ipd* (π k)› **and** *cdj*: ‹k $cd^{\pi}\!\!\rightarrow$ j› **and** *nipdj*: ‹*ipd* (π j) $\neq \pi$ m› **shows** ‹m $cd^{\pi}\!\!\rightarrow$ j› ⟨*proof*⟩

lemma *ipd-icd-greatest-cd-not-ipd*: **assumes** *ipd*: ‹π m = *ipd* (π k)› ‹$\forall$ $n \in \{k..<m\}$. π $n \neq$ *ipd* (π k)›
and *km*: ‹$k < m$› **and** *icdj*: ‹m $icd^{\pi}\!\!\rightarrow$ j› **shows** ‹j = (*GREATEST* j. k $cd^{\pi}\!\!\rightarrow$ $j \wedge$ *ipd* (π j) $\neq \pi$ m)›
⟨*proof*⟩

lemma *cd-impl-icd-cd*: **assumes** ‹i $cd^{\pi}\!\!\rightarrow$ l› **and** ‹i $icd^{\pi}\!\!\rightarrow$ k› **and** ‹$\neg$ i $icd^{\pi}\!\!\rightarrow$ l› **shows** ‹k $cd^{\pi}\!\!\rightarrow$ l›
⟨*proof*⟩

lemma *cdi-is-cd-icdi*: **assumes** ‹k $icd^{\pi}\!\!\rightarrow$ j› **shows** ‹k $cd^{\pi}\!\!\rightarrow$ $i \longleftrightarrow j$ $cd^{\pi}\!\!\rightarrow$ $i \vee i = j$›
⟨*proof*⟩

lemma *same-ipd-stable*: **assumes** ‹k $cd^{\pi}\!\!\rightarrow$ i› ‹k $cd^{\pi}\!\!\rightarrow$ j› ‹$i<j$› ‹*ipd* (π i) = *ipd* (π k)› **shows** ‹*ipd* (π j) = *ipd* (π k)›
⟨*proof*⟩

lemma *icd-pd-intermediate'*: **assumes** *icd*: ‹i $icd^{\pi}\!\!\rightarrow$ k› **and** *j*: ‹$k < j$› ‹$j < i$› **shows** ‹π i *pd*$\rightarrow$ (π j)›
⟨*proof*⟩

lemma *icd-pd-intermediate*: **assumes** *icd*: ‹i $icd^{\pi}\!\!\rightarrow$ k› **and** *j*: ‹$k < j$› ‹$j \leq i$› **shows** ‹π i *pd*$\rightarrow$ (π j)›
⟨*proof*⟩

lemma *no-icd-pd*: **assumes** *path*: ‹*is-path* π› **and** *noicd*: ‹$\forall$ $l \geq n$. $\neg$ k $icd^{\pi}\!\!\rightarrow$ l› **and** *nk*: ‹$n \leq k$› **shows** ‹π k *pd*$\rightarrow$ π n›
⟨*proof*⟩

lemma *first-pd-no-cd*: **assumes** *path*: ‹*is-path* π› **and** *pd*: ‹π n *pd*$\rightarrow$ π *0*› **and** *first*: ‹$\forall$ $l < n$. π $l \neq \pi$ n›
shows ‹$\forall$ l. $\neg$ n $cd^{\pi}\!\!\rightarrow$ l›
⟨*proof*⟩

lemma *first-pd-no-icd*: **assumes** *path*: ‹*is-path* π› **and** *pd*: ‹π *n pd*→ π *0*› **and** *first*: ‹$\forall$ *l* < *n*. π *l* ≠ π *n*› **shows** ‹$\forall$ *l*. ¬ *n* *icd*$^{\pi}$→ *l*›
⟨*proof*⟩

lemma *path-nret-ex-nipd*: **assumes** ‹*is-path* π› ‹$\forall$ *i*. π *i* ≠ *return*› **shows** ‹$\forall$ *i*. ($\exists$ *j*≥*i*. ($\forall$ *k*>*j*. π *k* ≠ *ipd* (π *j*)))› ⟨*proof*⟩

lemma *path-nret-ex-all-cd*: **assumes** ‹*is-path* π› ‹$\forall$ *i*. π *i* ≠ *return*› **shows** ‹$\forall$ *i*. ($\exists$ *j*≥*i*. ($\forall$ *k*>*j*. *k* *cd*$^{\pi}$→ *j*))›
⟨*proof*⟩

lemma *path-nret-inf-all-cd*: **assumes** ‹*is-path* π› ‹$\forall$ *i*. π *i* ≠ *return*› **shows** ‹¬ *finite* {*j*. $\forall$ *k*>*j*. *k* *cd*$^{\pi}$→ *j*}›
⟨*proof*⟩

lemma *path-nret-inf-icd-seq*: **assumes** *path*: ‹*is-path* π› **and** *nret*: ‹$\forall$ *i*. π *i* ≠ *return*›
obtains *f* **where** ‹$\forall$ *i*. *f* (*Suc i*) *icd*$^{\pi}$→ *f i*› ‹*range f* = {*i*. $\forall$ *j*>*i*. *j* *cd*$^{\pi}$→ *i*}› ‹¬ ($\exists$ *i*. *f 0* *cd*$^{\pi}$→ *i*)›
⟨*proof*⟩

lemma *cdi-iff-no-strict-pd*: ‹*i* *cd*$^{\pi}$→ *k* ⟷ *is-path* π ∧ *k* < *i* ∧ π *i* ≠ *return* ∧ ($\forall$ *j* ∈ {*k*..*i*}. ¬ (π *k*, π *j*) ∈ *pdt*)›
⟨*proof*⟩

2.5 Facts about Control Slices

lemma *last-cs*: ‹*last* (*cs*$^{\pi}$ *i*) = π *i*› ⟨*proof*⟩

lemma *cs-not-nil*: ‹*cs*$^{\pi}$ *n* ≠ []› ⟨*proof*⟩

lemma *cs-return*: **assumes** ‹π *n* = *return*› **shows** ‹*cs*$^{\pi}$ *n* = [π *n*]› ⟨*proof*⟩

lemma *cs-0*[*simp*]: ‹*cs*$^{\pi}$ *0* = [π *0*]› ⟨*proof*⟩

lemma *cs-inj*: **assumes** ‹*is-path* π› ‹π *n* ≠ *return*› ‹*cs*$^{\pi}$ *n* = *cs*$^{\pi}$ *n*′› **shows** ‹*n* = *n*′›
⟨*proof*⟩

lemma *cs-cases*: **fixes** π *i*
obtains (*base*) ‹*cs*$^{\pi}$ *i* = [π *i*]› **and** ‹$\forall$ *k*. ¬ *i* *cd*$^{\pi}$→ *k*› |
(*depend*) *k* **where** ‹*cs*$^{\pi}$ *i* = (*cs*$^{\pi}$ *k*)@[π *i*]› **and** ‹*i* *icd*$^{\pi}$→ *k*›
⟨*proof*⟩

lemma *cs-length-one*: **assumes** ‹*length* (*cs*$^{\pi}$ *i*) = *1*› **shows** ‹*cs*$^{\pi}$ *i* = [π *i*]› **and** ‹$\forall$ *k*. ¬ *i* *cd*$^{\pi}$→ *k*›
⟨*proof*⟩

lemma *cs-length-g-one*: **assumes** ‹*length* (*cs*$^{\pi}$ *i*) ≠ *1*› **obtains** *k* **where** ‹*cs*$^{\pi}$ *i* = (*cs*$^{\pi}$ *k*)@[π *i*]› **and** ‹*i* *icd*$^{\pi}$→ *k*›
⟨*proof*⟩

lemma *claim*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π′› **and** *ii*: ‹*cs*$^{\pi}$ *i* = *cs*$^{\pi'}$ *i*′› **and** *jj*: ‹*cs*$^{\pi}$ *j* = *cs*$^{\pi'}$ *j*′›
and *bl*: ‹*butlast* (*cs*$^{\pi}$ *i*) = *butlast* (*cs*$^{\pi}$ *j*)› **and** *nret*: ‹π *i* ≠ *return*› **and** *ilj*: ‹*i* < *j*›
shows ‹*i*′ < *j*′›
⟨*proof*⟩

lemma *cs-split′*: **assumes** ‹*cs*$^{\pi}$ *i* = *xs*@[*x*,*x*′]@*ys*› **shows** ‹$\exists$ *m*. *cs*$^{\pi}$ *m* = *xs*@[*x*] ∧ *i* *cd*$^{\pi}$→ *m*›
⟨*proof*⟩

lemma *cs-split*: **assumes** ‹$cs^\pi\ i = xs@[x]@ys@[\pi\ i]$› **shows** ‹$\exists\ m.\ cs^\pi\ m = xs@[x] \wedge i\ cd^\pi\!\!\rightarrow\ m$› ⟨*proof*⟩

lemma *cs-less-split*: **assumes** ‹$xs \prec ys$› **obtains** *a as* **where** ‹$ys = xs@a\#as$›
⟨*proof*⟩

lemma *cs-select-is-cs*: **assumes** ‹*is-path* π› ‹$xs \neq Nil$› ‹$xs \prec cs^\pi\ k$› **shows** ‹$cs^\pi\ (\pi{¡}xs) = xs$› ‹$k\ cd^\pi\!\!\rightarrow (\pi{¡}xs)$›⟨*proof*⟩

lemma *cd-in-cs*: **assumes** ‹$n\ cd^\pi\!\!\rightarrow\ m$› **shows** ‹$\exists\ ns.\ cs^\pi\ n = (cs^\pi\ m)\ @\ ns\ @[\pi\ n]$›
⟨*proof*⟩

lemma *butlast-cs-not-cd*: **assumes** ‹*butlast* $(cs^\pi\ m) =$ *butlast* $(cs^\pi\ n)$› **shows** ‹$\neg\ m\ cd^\pi\!\!\rightarrow n$›
⟨*proof*⟩

lemma *wn-cs-butlast*: **assumes** ‹*butlast* $(cs^\pi\ m) =$ *butlast* $(cs^\pi\ n)$› ‹$i\ cd^\pi\!\!\rightarrow\ m$› ‹$j\ cd^\pi\!\!\rightarrow\ n$› ‹$m<n$› **shows** ‹$i<j$›
⟨*proof*⟩

This is the central theorem making the control slice suitable for matching indices between executions.

theorem *cs-order*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *csi*: ‹$cs^\pi\ i = cs^{\pi'}\ i'$›
and *csj*: ‹$cs^\pi\ j = cs^{\pi'}\ j'$› **and** *nret*: ‹$\pi\ i \neq return$› **and** *ilj*: ‹$i < j$›
shows ‹$i'<j'$›
⟨*proof*⟩

lemma *cs-order-le*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *csi*: ‹$cs^\pi\ i = cs^{\pi'}\ i'$›
and *csj*: ‹$cs^\pi\ j = cs^{\pi'}\ j'$› **and** *nret*: ‹$\pi\ i \neq return$› **and** *ilj*: ‹$i \leq j$›
shows ‹$i'\leq j'$› ⟨*proof*⟩

lemmas *cs-induct*[*case-names cs*] = *cs.induct*

lemma *icdi-path-swap*: **assumes** ‹*is-path* π'› ‹$j\ icd^\pi\!\!\rightarrow k$› ‹$\pi =_j\ \pi'$› **shows** ‹$j\ icd^{\pi'}\!\!\rightarrow k$› ⟨*proof*⟩

lemma *icdi-path-swap-le*: **assumes** ‹*is-path* π'› ‹$j\ icd^\pi\!\!\rightarrow k$› ‹$\pi =_n\ \pi'$› ‹$j \leq n$› **shows** ‹$j\ icd^{\pi'}\!\!\rightarrow k$› ⟨*proof*⟩

lemma *cs-path-swap*: **assumes** ‹*is-path* π› ‹*is-path* π'› ‹$\pi =_k \pi'$› **shows** ‹$cs^\pi\ k = cs^{\pi'}\ k$› ⟨*proof*⟩

lemma *cs-path-swap-le*: **assumes** ‹*is-path* π› ‹*is-path* π'› ‹$\pi =_n\ \pi'$› ‹$k \leq n$› **shows** ‹$cs^\pi\ k = cs^{\pi'}\ k$› ⟨*proof*⟩

lemma *cs-path-swap-cd*: **assumes** ‹*is-path* π› **and** ‹*is-path* π'› **and** ‹$cs^\pi\ n = cs^{\pi'}\ n'$› **and** ‹$n\ cd^\pi\!\!\rightarrow\ k$›
obtains k' **where** ‹$n'\ cd^{\pi'}\!\!\rightarrow\ k'$› **and** ‹$cs^\pi\ k = cs^{\pi'}\ k'$›
⟨*proof*⟩

lemma *path-ipd-swap*: **assumes** ‹*is-path* π› ‹$\pi\ k \neq return$› ‹$k < n$›
obtains $\pi'\ m$ **where** ‹*is-path* π'› ‹$\pi =_n\ \pi'$› ‹$k < m$› ‹$\pi'\ m = ipd\ (\pi'\ k)$› ‹$\forall\ l \in \{k..<m\}.\ \pi'\ l \neq ipd\ (\pi'\ k)$›
⟨*proof*⟩

lemma *cs-sorted-list-of-cd'*: ‹$cs^\pi\ k = map\ \pi$ (*sorted-list-of-set* $\{\ i\ .\ k\ cd^\pi\!\!\rightarrow i\}) @ [\pi\ k]$›
⟨*proof*⟩

lemma *cs-sorted-list-of-cd*: ‹$cs^\pi\ k = map\ \pi$ (*sorted-list-of-set* $(\{\ i\ .\ k\ cd^\pi\!\!\rightarrow i\} \cup \{k\}))$› ⟨*proof*⟩

lemma *cs-not-ipd*: **assumes** ‹$k\ cd^\pi\!\!\rightarrow j \wedge ipd\ (\pi\ j) \neq ipd\ (\pi\ k)$› (**is** ‹*?Q j*›)
shows ‹cs^π (*GREATEST j*. $k\ cd^\pi\!\!\rightarrow j \wedge ipd\ (\pi\ j) \neq ipd\ (\pi\ k)) = [n{\leftarrow}cs^\pi\ k\ .\ ipd\ n \neq ipd\ (\pi\ k)]$›
(**is** ‹cs^π *?j* = *filter ?P* -›)
⟨*proof*⟩

lemma *cs-ipd*: **assumes** *ipd*: ‹$\pi\ m = ipd\ (\pi\ k)$› ‹$\forall\ n \in \{k..<m\}.\ \pi\ n \neq ipd\ (\pi\ k)$›
and *km*: ‹$k < m$› **shows** ‹$cs^{\pi}\ m = [n \leftarrow cs^{\pi}\ k\ .\ ipd\ n \neq \pi\ m]\ @\ [\pi\ m]$›
⟨*proof*⟩

lemma *converged-ipd-same-icd*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *converge*: ‹$l < m$› ‹$cs^{\pi}\ m = cs^{\pi'}\ m'$›
and *csk*: ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› **and** *icd*: ‹$l\ icd^{\pi}\rightarrow k$› **and** *suc*: ‹$\pi\ (Suc\ k) = \pi'\ (Suc\ k')$›
and *ipd*: ‹$\pi'\ m' = ipd\ (\pi\ k)$› ‹$\forall\ n \in \{k'..<m'\}.\ \pi'\ n \neq ipd\ (\pi\ k)$›
shows ‹$\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l'$›
⟨*proof*⟩

lemma *converged-same-icd*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *converge*: ‹$l < n$› ‹$cs^{\pi}\ n = cs^{\pi'}\ n'$›
and *csk*: ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› **and** *icd*: ‹$l\ icd^{\pi}\rightarrow k$› **and** *suc*: ‹$\pi\ (Suc\ k) = \pi'\ (Suc\ k')$›
shows ‹$\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l'$› ⟨*proof*⟩

lemma *cd-is-cs-less*: **assumes** ‹$l\ cd^{\pi}\rightarrow k$› **shows** ‹$cs^{\pi}\ k \prec cs^{\pi}\ l$› ⟨*proof*⟩

lemma *cs-select-id*: **assumes** ‹*is-path* π› ‹$\pi\ k \neq return$› **shows** ‹$\pi ¡ cs^{\pi}\ k = k$› (**is** ‹$?k = k$›) ⟨*proof*⟩

lemma *cs-single-nocd*: **assumes** ‹$cs^{\pi}\ i = [x]$› **shows** ‹$\forall\ k.\ \neg\ i\ cd^{\pi}\rightarrow k$› ⟨*proof*⟩

lemma *cs-single-pd-intermed*: **assumes** ‹*is-path* π› ‹$cs^{\pi}\ n = [\pi\ n]$› ‹$k \leq n$› **shows** ‹$\pi\ n\ pd\rightarrow \pi\ k$› ⟨*proof*⟩

lemma *cs-first-pd*: **assumes** *path*: ‹*is-path* π› **and** *pd*: ‹$\pi\ n\ pd\rightarrow \pi\ 0$› **and** *first*: ‹$\forall\ l < n.\ \pi\ l \neq \pi\ n$› **shows** ‹$cs^{\pi}\ n = [\pi\ n]$›
⟨*proof*⟩

lemma *converged-pd-cs-single*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *converge*: ‹$l < m$› ‹$cs^{\pi}\ m = cs^{\pi'}\ m'$›
and $\pi 0$: ‹$\pi\ 0 = \pi'\ 0$› **and** *mpdl*: ‹$\pi\ m\ pd\rightarrow \pi\ l$› **and** *csl*: ‹$cs^{\pi}\ l = [\pi\ l]$›
shows ‹$\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l'$› ⟨*proof*⟩

lemma *converged-cs-single*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *converge*: ‹$l < m$› ‹$cs^{\pi}\ m = cs^{\pi'}\ m'$›

and $\pi 0$: ‹$\pi\ 0 = \pi'\ 0$› **and** *csl*: ‹$cs^{\pi}\ l = [\pi\ l]$›
shows ‹$\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l'$› ⟨*proof*⟩

lemma *converged-cd-same-suc*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *init*: ‹$\pi\ 0 = \pi'\ 0$›
and *cd-suc*: ‹$\forall\ k\ k'.\ cs^{\pi}\ k = cs^{\pi'}\ k' \wedge l\ cd^{\pi}\rightarrow k \longrightarrow \pi\ (Suc\ k) = \pi'\ (Suc\ k')$› **and** *converge*: ‹$l < m$› ‹$cs^{\pi}\ m = cs^{\pi'}\ m'$›
shows ‹$\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l'$›
⟨*proof*⟩

lemma *converged-cd-diverge*:
assumes *path*: ‹*is-path* π› ‹*is-path* π'› **and** *init*: ‹$\pi\ 0 = \pi'\ 0$› **and** *notin*: ‹$\neg\ (\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l')$› **and** *converge*: ‹$l < m$› ‹$cs^{\pi}\ m = cs^{\pi'}\ m'$›
obtains $k\ k'$ **where** ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› ‹$l\ cd^{\pi}\rightarrow k$› ‹$\pi\ (Suc\ k) \neq \pi'\ (Suc\ k')$›
⟨*proof*⟩

lemma *converged-cd-same-suc-return*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** $\pi 0$: ‹$\pi\ 0 = \pi'\ 0$›

and *cd-suc*: ‹$\forall\ k\ k'.\ cs^{\pi}\ k = cs^{\pi'}\ k' \wedge l\ cd^{\pi}\rightarrow k \longrightarrow \pi\ (Suc\ k) = \pi'\ (Suc\ k')$› **and** *ret*: ‹$\pi'\ n' = return$›
shows ‹$\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l'$›⟨*proof*⟩

lemma *converged-cd-diverge-return*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *init*: ‹$\pi\ 0 = \pi'\ 0$›
and *notin*: ‹$\neg\ (\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l')$› **and** *ret*: ‹$\pi'\ m' = return$›
obtains $k\ k'$ **where** ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› ‹$l\ cd^{\pi}\rightarrow k$› ‹$\pi\ (Suc\ k) \neq \pi'\ (Suc\ k')$› ⟨*proof*⟩

lemma *returned-missing-cd-or-loop*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** $\pi 0$: ‹$\pi\ 0 = \pi'\ 0$›
and *notin'*: ‹$\neg(\exists\ k'.\ cs^{\pi}\ k = cs^{\pi'}\ k')$› **and** *nret*: ‹$\forall\ n'.\ \pi'\ n' \neq return$› **and** *ret*: ‹$\pi\ n = return$›
obtains $i\ i'$ **where** ‹$i<k$› ‹$cs^{\pi}\ i = cs^{\pi'}\ i'$› ‹$\pi\ (Suc\ i) \neq \pi'\ (Suc\ i')$› ‹$k\ cd^{\pi}\rightarrow i \vee (\forall\ j'> i'.\ j'\ cd^{\pi'}\rightarrow i')$›
⟨*proof*⟩

lemma *missing-cd-or-loop*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** $\pi 0$: ‹$\pi\ 0 = \pi'\ 0$› **and** *notin'*: ‹$\neg(\exists\ k'.\ cs^{\pi}\ k = cs^{\pi'}\ k')$›
obtains $i\ i'$ **where** ‹$i < k$› ‹$cs^{\pi}\ i = cs^{\pi'}\ i'$› ‹$\pi\ (Suc\ i) \neq \pi'\ (Suc\ i')$› ‹$k\ cd^{\pi}\rightarrow i \vee (\forall\ j'> i'.\ j'\ cd^{\pi'}\rightarrow i')$›
⟨*proof*⟩

lemma *path-shift-set-cd*: **assumes** ‹*is-path* π› **shows** ‹$\{k + j|\ j\ .\ n\ cd^{\pi\ll k}\rightarrow j\ \} = \{i.\ (k+n)\ cd^{\pi}\rightarrow i \wedge k \leq i\ \}$›
⟨*proof*⟩

lemma *cs-path-shift-set-cd*: **assumes** *path*: ‹*is-path* π› **shows** ‹$cs^{\pi\ll k}\ n = map\ \pi\ (sorted\text{-}list\text{-}of\text{-}set\ \{i.\ k+n\ cd^{\pi}\rightarrow i \wedge k \leq i\ \})\ @\ [\pi\ (k+n)]$›
⟨*proof*⟩

lemma *cs-split-shift-cd*: **assumes** ‹$n\ cd^{\pi}\rightarrow j$› **and** ‹$j < k$› **and** ‹$k < n$› **and** ‹$\forall j'<k.\ n\ cd^{\pi}\rightarrow j' \longrightarrow j' \leq j$›
shows ‹$cs^{\pi}\ n = cs^{\pi}\ j\ @\ cs^{\pi\ll k}\ (n-k)$›
⟨*proof*⟩

lemma *cs-split-shift-nocd*: **assumes** ‹*is-path* π› **and** ‹$k < n$› **and** ‹$\forall j.\ n\ cd^{\pi}\rightarrow j \longrightarrow k \leq j$› **shows** ‹$cs^{\pi}\ n = cs^{\pi\ll k}\ (n-k)$›
⟨*proof*⟩

lemma *shifted-cs-eq-is-eq*: **assumes** ‹*is-path* π› **and** ‹*is-path* π'› **and** ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› **and** ‹$cs^{\pi\ll k}\ n = cs^{\pi'\ll k'}\ n'$› **shows** ‹$cs^{\pi}\ (k+n) = cs^{\pi'}\ (k'+n')$›
⟨*proof*⟩

lemma *cs-eq-is-eq-shifted*: **assumes** ‹*is-path* π› **and** ‹*is-path* π'› **and** ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› **and** ‹$cs^{\pi}\ (k+n) = cs^{\pi'}\ (k'+n')$› **shows** ‹$cs^{\pi\ll k}\ n = cs^{\pi'\ll k'}\ n'$›
⟨*proof*⟩

lemma *converged-cd-diverge-cs*: **assumes** ‹*is-path* π› **and** ‹*is-path* π'› **and** ‹$cs^{\pi}\ j\ = cs^{\pi'}\ j'$› **and** ‹$j<l$› **and**
‹$\neg\ (\exists\, l'.\ cs^{\pi}\ l = cs^{\pi'}\ l')$› **and** ‹$l < m$› **and** ‹$cs^{\pi}\ m = cs^{\pi'}\ m'$›
obtains $k\ k'$ **where** ‹$j\leq k$› ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› **and** ‹$l\ cd^{\pi}\rightarrow k$› **and** ‹$\pi\ (Suc\ k) \neq \pi'\ (Suc\ k')$›
⟨*proof*⟩

lemma *cs-ipd-conv*: **assumes** *csk*: ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› **and** *ipd*: ‹$\pi\ l = ipd\ (\pi\ k)$› ‹$\pi'\ l' = ipd(\pi'\ k')$›
and *nipd*: ‹$\forall\, n\in\{k..<l\}.\ \pi\ n \neq ipd\ (\pi\ k)$› ‹$\forall\, n'\in\{k'..<l'\}.\ \pi'\ n' \neq ipd\ (\pi'\ k')$› **and** *kl*: ‹$k < l$› ‹$k' < l'$›
shows ‹$cs^{\pi}\ l = cs^{\pi'}\ l'$› ⟨*proof*⟩

lemma *cp-eq-cs*: **assumes** ‹$((\sigma,k),(\sigma',k'))\in cp$› **shows** ‹$cs^{path\ \sigma}\ k = cs^{path\ \sigma'}\ k'$›
⟨*proof*⟩

lemma *cd-cs-swap*: **assumes** ‹$l\ cd^{\pi}\rightarrow k$› ‹$cs^{\pi}\ l = cs^{\pi'}\ l'$› ‹$cs^{\pi}\ k = cs^{\pi'}\ k'$› **shows** ‹$l'\ cd^{\pi'}\rightarrow k'$› ⟨*proof*⟩

2.6 Facts about Observations

lemma *kth-obs-not-none*: **assumes** ‹*is-kth-obs* (*path* σ) $k\ i$› **obtains** a **where** ‹*obsp* $\sigma\ i$ = *Some* a› ⟨*proof*⟩

lemma *kth-obs-unique*: ‹*is-kth-obs* $\pi\ k\ i \Longrightarrow$ *is-kth-obs* $\pi\ k\ j \Longrightarrow i = j$› ⟨*proof*⟩

lemma *obs-none-no-kth-obs*: **assumes** ‹*obs* $\sigma\ k$ = *None*› **shows** ‹$\neg$ ($\exists\ i$. *is-kth-obs* (*path* σ) $k\ i$)›
⟨*proof*⟩

lemma *obs-some-kth-obs* : **assumes** ‹*obs* $\sigma\ k \neq$ *None*› **obtains** i **where** ‹*is-kth-obs* (*path* σ) $k\ i$› ⟨*proof*⟩

lemma *not-none-is-obs*: **assumes** ‹*att*($\pi\ i$) $\neq$ *None*› **shows** ‹*is-kth-obs* π (*card* (*obs-ids* $\pi \cap \{..<i\}$)) i› ⟨*proof*⟩

lemma *in-obs-ids-is-kth-obs*: **assumes** ‹$i \in$ *obs-ids* π› **obtains** k **where** ‹*is-kth-obs* $\pi\ k\ i$› ⟨*proof*⟩

lemma *kth-obs-stable*: **assumes** ‹*is-kth-obs* $\pi\ l\ j$› ‹$k < l$› **shows** ‹$\exists\ i$. *is-kth-obs* $\pi\ k\ i$› ⟨*proof*⟩

lemma *kth-obs-mono*: **assumes** ‹*is-kth-obs* $\pi\ k\ i$› ‹*is-kth-obs* $\pi\ l\ j$› ‹$k < l$› **shows** ‹$i < j$› ⟨*proof*⟩

lemma *kth-obs-le-iff*: **assumes** ‹*is-kth-obs* $\pi\ k\ i$› ‹*is-kth-obs* $\pi\ l\ j$› **shows** ‹$k < l \longleftrightarrow i < j$› ⟨*proof*⟩

lemma *ret-obs-all-obs*: **assumes** *path*: ‹*is-path* π› **and** *iki*: ‹*is-kth-obs* $\pi\ k\ i$› **and** *ret*: ‹$\pi\ i$ = *return*› **and** *kl*: ‹$k < l$› **obtains** j **where** ‹*is-kth-obs* $\pi\ l\ j$›
⟨*proof*⟩

lemma *no-kth-obs-missing-cs*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *iki*: ‹*is-kth-obs* $\pi\ k\ i$› **and** *not-in-*π': ‹$\neg(\exists\ i'$. *is-kth-obs* $\pi'\ k\ i')$› **obtains** $l\ j$ **where** ‹*is-kth-obs* $\pi\ l\ j$› ‹$\neg$ ($\exists\ j'$. $cs^{\pi}\ j = cs^{\pi'}\ j'$)›
⟨*proof*⟩

lemma *kth-obs-cs-missing-cs*: **assumes** *path*: ‹*is-path* π› ‹*is-path* π'› **and** *iki*: ‹*is-kth-obs* $\pi\ k\ i$› **and** *iki'*: ‹*is-kth-obs* $\pi'\ k\ i'$› **and** *csi*: ‹$cs^{\pi}\ i \neq cs^{\pi'}\ i'$›
obtains $l\ j$ **where** ‹$j \leq i$› ‹*is-kth-obs* $\pi\ l\ j$› ‹$\neg$ ($\exists\ j'$. $cs^{\pi}\ j = cs^{\pi'}\ j'$)› | $l'\ j'$ **where** ‹$j' \leq i'$› ‹*is-kth-obs* $\pi'\ l'\ j'$› ‹$\neg$ ($\exists\ j$. $cs^{\pi}\ j = cs^{\pi'}\ j'$)›
⟨*proof*⟩

2.7 Facts about Data

lemma *reads-restrict1*: ‹$\sigma \restriction$ (*reads* n) = $\sigma' \restriction$ (*reads* n) $\Longrightarrow \forall\ x \in$ *reads* n. $\sigma\ x = \sigma'\ x$› ⟨*proof*⟩

lemma *reads-restrict2*: ‹$\forall\ x \in$ *reads* n. $\sigma\ x = \sigma'\ x \Longrightarrow \sigma \restriction$ (*reads* n) = $\sigma' \restriction$ (*reads* n)› ⟨*proof*⟩

lemma *reads-restrict*: ‹($\sigma \restriction$ (*reads* n) = $\sigma' \restriction$ (*reads* n)) = ($\forall\ x \in$ *reads* n. $\sigma\ x = \sigma'\ x$)› ⟨*proof*⟩

lemma *reads-restr-suc*: ‹$\sigma \restriction$ (*reads* n) = $\sigma' \restriction$ (*reads* n) $\Longrightarrow$ *suc* $n\ \sigma$ = *suc* $n\ \sigma'$› ⟨*proof*⟩

lemma *reads-restr-sem*: ‹$\sigma \restriction$ (*reads* n) = $\sigma' \restriction$ (*reads* n) $\Longrightarrow \forall\ v \in$ *writes* n. *sem* $n\ \sigma\ v$ = *sem* $n\ \sigma'\ v$› ⟨*proof*⟩

lemma *reads-obsp*: **assumes** ‹*path* $\sigma\ k$ = *path* $\sigma'\ k'$› ‹$\sigma^{k} \restriction$ (*reads* (*path* $\sigma\ k$)) = $\sigma'^{k'} \restriction$ (*reads* (*path* $\sigma\ k$))› **shows** ‹*obsp* $\sigma\ k$ = *obsp* $\sigma'\ k'$›
⟨*proof*⟩

lemma *no-writes-unchanged0*: **assumes** ‹$\forall\ l{<}k$. $v \notin$ *writes*(*path* $\sigma\ l$)› **shows** ‹(σ^{k}) $v = \sigma\ v$› ⟨*proof*⟩

lemma *written-read-dd*: **assumes** ‹*is-path* π› ‹$v \in$ *reads* $(\pi\ k)$ › ‹$v \in$ *writes* $(\pi\ j)$› ‹$j<k$› **obtains** l **where** ‹k $dd^{\pi,v}\!\!\rightarrow$ l›
⟨*proof*⟩

lemma *no-writes-unchanged*: **assumes** ‹$k \leq l$› ‹$\forall\ j \in \{k..<l\}.\ v \notin$ *writes*$(path\ \sigma\ j)$› **shows** ‹$(\sigma^l)\ v = (\sigma^k)\ v$›
⟨*proof*⟩

lemma *ddi-value*: **assumes** ‹l $dd^{(path\ \sigma),v}\!\!\rightarrow$ k› **shows** ‹$(\sigma^l)\ v = (\sigma^{Suc\ k}\ )\ v$›
⟨*proof*⟩

lemma *written-value*: **assumes** ‹$path\ \sigma\ l = path\ \sigma'\ l'$› ‹$\sigma^l \restriction reads\ (path\ \sigma\ l) = \sigma'^{l'} \restriction reads\ (path\ \sigma\ l)$› ‹$v \in$ *writes* $(path\ \sigma\ l)$›
shows ‹$(\sigma^{Suc\ l}\ )\ v = (\sigma'^{Suc\ l'}\ )\ v$›
⟨*proof*⟩

2.8 Facts about Contradicting Paths

lemma *obsp-contradict*: **assumes** *csk*: ‹$cs^{path\ \sigma}\ k = cs^{path\ \sigma'}\ k'$› **and** *obs*: ‹$obsp\ \sigma\ k \neq obsp\ \sigma'\ k'$› **shows** ‹$(\sigma',\ k')$ $\mathfrak{c}$ $(\sigma,\ k)$›
⟨*proof*⟩

lemma *missing-cs-contradicts*: **assumes** *notin*: ‹$\neg(\exists\ k'.\ cs^{path\ \sigma}\ k = cs^{path\ \sigma'}\ k')$› **and** *converge*: ‹$k<n$› ‹$cs^{path\ \sigma}\ n = cs^{path\ \sigma'}\ n'$› **shows** ‹$\exists\ j'.\ (\sigma',\ j')$ $\mathfrak{c}$ $(\sigma,\ k)$›
⟨*proof*⟩

theorem *obs-neq-contradicts-term*: **fixes** $\sigma\ \sigma'$ **defines** π: ‹$\pi \equiv path\ \sigma$› **and** π': ‹$\pi' \equiv path\ \sigma'$› **assumes** *ret*: ‹$\pi\ n = return$› ‹$\pi'\ n' = return$› **and** *obsne*: ‹$obs\ \sigma \neq obs\ \sigma'$›
shows ‹$\exists\ k\ k'.\ ((\sigma',\ k')$ $\mathfrak{c}$ $(\sigma\ ,k) \wedge \pi\ k \in dom\ (att)) \vee ((\sigma,\ k)$ $\mathfrak{c}$ $(\sigma'\ ,k') \wedge \pi'\ k' \in dom\ (att))$›
⟨*proof*⟩

lemma *obs-neq-some-contradicts'*: **fixes** $\sigma\ \sigma'$ **defines** π: ‹$\pi \equiv path\ \sigma$› **and** π': ‹$\pi' \equiv path\ \sigma'$›
assumes *obsnecs*: ‹$obsp\ \sigma\ i \neq obsp\ \sigma'\ i' \vee cs^{\pi}\ i \neq cs^{\pi'}\ i'$›
and *iki*: ‹*is-kth-obs* $\pi\ k\ i$› **and** *iki'*: ‹*is-kth-obs* $\pi'\ k\ i'$›
shows ‹$\exists\ k\ k'.\ ((\sigma',\ k')$ $\mathfrak{c}$ $(\sigma\ ,k) \wedge \pi\ k \in dom\ att) \vee ((\sigma,\ k)$ $\mathfrak{c}$ $(\sigma'\ ,k') \wedge \pi'\ k' \in dom\ att)$›
⟨*proof*⟩

theorem *obs-neq-some-contradicts*: **fixes** $\sigma\ \sigma'$ **defines** π: ‹$\pi \equiv path\ \sigma$› **and** π': ‹$\pi' \equiv path\ \sigma'$›
assumes *obsne*: ‹$obs\ \sigma\ k \neq obs\ \sigma'\ k$› **and** *not-none*: ‹$obs\ \sigma\ k \neq None$› ‹$obs\ \sigma'\ k \neq None$›
shows ‹$\exists\ k\ k'.\ ((\sigma',\ k')$ $\mathfrak{c}$ $(\sigma\ ,k) \wedge \pi\ k \in dom\ att) \vee ((\sigma,\ k)$ $\mathfrak{c}$ $(\sigma'\ ,k') \wedge \pi'\ k' \in dom\ att)$›
⟨*proof*⟩

theorem *obs-neq-ret-contradicts*: **fixes** $\sigma\ \sigma'$ **defines** π: ‹$\pi \equiv path\ \sigma$› **and** π': ‹$\pi' \equiv path\ \sigma'$›
assumes *ret*: ‹$\pi\ n = return$› **and** *obsne*: ‹$obs\ \sigma'\ i \neq obs\ \sigma\ i$› **and** *obs*:‹$obs\ \sigma'\ i \neq None$›
shows ‹$\exists\ k\ k'.\ ((\sigma',\ k')$ $\mathfrak{c}$ $(\sigma\ ,k) \wedge \pi\ k \in dom\ (att)) \vee ((\sigma,\ k)$ $\mathfrak{c}$ $(\sigma'\ ,k') \wedge \pi'\ k' \in dom\ (att))$›
⟨*proof*⟩

2.9 Facts about Critical Observable Paths

lemma *contradicting-in-cp*: **assumes** *leq*:‹$\sigma =_L \sigma'$› **and** *cseq*: ‹$cs^{path\ \sigma}\ k = cs^{path\ \sigma'}\ k'$›
and *readv*: ‹$v \in reads(path\ \sigma\ k)$› **and** *vneq*: ‹$(\sigma^k)\ v \neq (\sigma'^{k'})\ v$› **shows** ‹$((\sigma,k),(\sigma',k')) \in cp$›
⟨*proof*⟩

theorem *contradicting-in-cop*: **assumes** ‹$\sigma =_L \sigma'$› **and** ‹(σ',k') $\mathfrak{c}$ (σ,k)› **and** ‹$path\ \sigma\ k \in dom\ att$›
shows ‹$((\sigma,k),\sigma',k') \in cop$› ⟨*proof*⟩

theorem *cop-correct-term*: **fixes** $\sigma\ \sigma'$ **defines** π: ‹$\pi \equiv path\ \sigma$› **and** π': ‹$\pi' \equiv path\ \sigma'$›
assumes *ret*: ‹$\pi\ n = return$› ‹$\pi'\ n' = return$› **and** *obsne*: ‹$obs\ \sigma \neq obs\ \sigma'$› **and** *leq*: ‹$\sigma =_L \sigma'$›
shows ‹$\exists\ k\ k'.\ ((\sigma,k),\sigma',k') \in cop \vee ((\sigma',k'),\sigma,k) \in cop$›
⟨*proof*⟩

theorem *cop-correct-ret*: **fixes** $\sigma\ \sigma'$ **defines** π: ‹$\pi \equiv path\ \sigma$› **and** π': ‹$\pi' \equiv path\ \sigma'$›
assumes *ret*: ‹$\pi\ n = return$› **and** *obsne*: ‹$obs\ \sigma\ i \neq obs\ \sigma'\ i$› **and** *obs*: ‹$obs\ \sigma'\ i \neq None$› **and** *leq*: ‹$\sigma =_L \sigma'$›
shows ‹$\exists\ k\ k'.\ ((\sigma,k),\sigma',k') \in cop \vee ((\sigma',k'),\sigma,k) \in cop$›
⟨*proof*⟩

theorem *cop-correct-nterm*: **assumes** *obsne*: ‹$obs\ \sigma\ k \neq obs\ \sigma'\ k$› ‹$obs\ \sigma\ k \neq None$› ‹$obs\ \sigma'\ k \neq None$›
and *leq*: ‹$\sigma =_L \sigma'$›
shows ‹$\exists\ k\ k'.\ ((\sigma,k),\sigma',k') \in cop \vee ((\sigma',k'),\sigma,k) \in cop$›
⟨*proof*⟩

2.10 Correctness of the Characterisation

The following is our main correctness result. If there exist no critical observable paths, then the program is secure.

theorem *cop-correct*: **assumes** ‹$cop = empty$› **shows** ‹*secure*› ⟨*proof*⟩

Our characterisation is not only correct, it is also precise in the way that *cp* characterises exactly the matching indices in executions for low equivalent input states where diverging data is read. This follows easily as the inverse implication to lemma *contradicting-in-cp* can be shown by simple induction.

theorem *cp-iff-reads-contradict*: ‹$((\sigma,k),(\sigma',k')) \in cp \longleftrightarrow \sigma =_L \sigma' \wedge cs^{path\ \sigma}\ k = cs^{path\ \sigma'}\ k' \wedge (\exists\ v \in reads(path\ \sigma\ k).\ (\sigma^k)\ v \neq (\sigma'^{k'})\ v)$›
⟨*proof*⟩

In the same way the inverse implication to *contradicting-in-cop* follows easily such that we obtain the following characterisation of *cop*.

theorem *cop-iff-contradicting*: ‹$((\sigma,k),(\sigma',k')) \in cop \longleftrightarrow \sigma =_L \sigma' \wedge (\sigma',k')\ \mathfrak{c}\ (\sigma,k) \wedge path\ \sigma\ k \in dom\ att$›
⟨*proof*⟩

2.11 Correctness of the Single Path Approximation

theorem *cp-in-scp*: **assumes** ‹$((\sigma,k),(\sigma',k')) \in cp$› **shows** ‹$(path\ \sigma,k) \in scp \wedge (path\ \sigma',k') \in scp$›
⟨*proof*⟩

theorem *cop-in-scop*: **assumes** ‹$((\sigma,k),(\sigma',k')) \in cop$› **shows** ‹$(path\ \sigma,k) \in scop \wedge (path\ \sigma',k') \in scp$›
⟨*proof*⟩

The main correctness result for out single execution approximation follows directly.

theorem *scop-correct*: **assumes** ‹$scop = empty$› **shows** ‹*secure*›
⟨*proof*⟩

end

end

3 Example: Program Dependence Graphs

Program dependence graph (PDG) based slicing provides a very crude but direct approximation of our characterisation. As such we can easily derive a corresponding correctness result.

theory *PDG* **imports** *IFC*
begin

context *IFC*
begin

We utilise our established dependencies on program paths to define the PDG. Note that PDGs usually only contain immediate control dependencies instead of the transitive ones we use here. However as slicing is considering reachability questions this does not affect the result.

inductive-set *pdg* **where**
‹$[\![i\ cd^{\pi}\!\rightarrow k]\!] \Longrightarrow (\pi\ k,\ \pi\ i) \in pdg$› |
‹$[\![i\ dd^{\pi,v}\!\rightarrow k]\!] \Longrightarrow (\pi\ k,\ \pi\ i) \in pdg$›

The set of sources is the set of nodes reading high variables.

inductive-set *sources* **where**
‹$n \in nodes \Longrightarrow h \in hvars \Longrightarrow h \in reads\ n \Longrightarrow n \in sources$›

The forward slice is the set of nodes reachable in the PDG from the set of sources. To ensure security slicing aims to prove that no observable node is contained in the

inductive-set *slice* **where**
‹$n \in sources \Longrightarrow n \in slice$› |
‹$m \in slice \Longrightarrow (m,n) \in pdg \Longrightarrow n \in slice$›

As the PDG does not contain data control dependencies themselves we have to decompose these.

lemma *dcd-pdg*: **assumes** ‹$n\ dcd^{\pi,v}\!\rightarrow m\ via\ \pi'\ m'$› **obtains** l **where** ‹$(\pi\ m,l) \in pdg$› **and** ‹$(l,\pi\ n) \in pdg$›
⟨*proof*⟩

By induction it directly follows that the slice is an approximation of the single critical paths.

lemma *scp-slice*: ‹$(\pi,\ i) \in scp \Longrightarrow \pi\ i \in slice$›
⟨*proof*⟩

lemma *scop-slice*: ‹$(\pi,\ i) \in scop \Longrightarrow \pi\ i \in slice \cap dom(att)$› ⟨*proof*⟩

The requirement targeted by slicing, that no observable node is contained in the slice, is thereby a sound criteria for security.

lemma *pdg-correct*: **assumes** ‹$slice \cap dom(att) = \{\}$› **shows** ‹*secure*›
⟨*proof*⟩

end

end

References

[1] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 79–90, New York, NY, USA, 2009. ACM.