A shallow embedding of HyperCTL*

Markus N. Rabe Peter Lammich Andrei Popescu

Contents

1 Introduction

We formalize HyperCTL*, a temporal logic for expressing security properties introduced in [1,2]. We first define a shallow embedding of HyperCTL*, within which we prove inductive and coinductive rules for the operators. Then we show that a HyperCTL* formula captures Goguen-Meseguer noninterference, a landmark information flow property. We also define a deep embedding and connect it to the shallow embedding by a denotational semantics, for which we prove sanity w.r.t. dependence on the free variables. Finally, we show that under some finiteness assumptions about the model, noninterference is given by a (finitary) syntactic formula.

For the semantics of HyperCTL*, we mainly follow the earlier paper [1]. The Kripke structure for representing noninterference is essentially that of [1,Appendix B] – however, instead of using the formula from [1,Appendix B], we further add idle transitions to the Kripke structure and use the simpler formula from [1,Section 2.4].

[1] Bernd Finkbeiner, Markus N. Rabe and César Sánchez. A Temporal Logic for Hyperproperties. CoRR, abs/1306.6657, 2013.

[2] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe and César Sánchez. Temporal Logics for Hyperproperties. POST 2014, 265-284.

2 Preliminaries

abbreviation *any* **where** *any* ≡ *undefined*

lemma *append-singl-rev*: *a* # *as* = [*a*] @ *as* **by** *simp*

lemma *list-pair-induct*[*case-names Nil Cons*]:
assumes *P* [] **and** ⋀*a b list*. *P list* ⟹ *P* ((*a*,*b*) # *list*)
shows *P lista*
using *assms* **by** (*induction lista*) *auto*

lemma *list-pair-case*[*elim*, *case-names Nil Cons*]:
assumes *xs* = [] ⟹ *P* **and** ⋀*a b list*. *xs* = (*a*,*b*) # *list* ⟹ *P*
shows *P*
using *assms* **by**(*cases xs*, *auto*)

definition *asList* :: *'a set* ⇒ *'a list* **where**
asList A ≡ *SOME as*. *distinct as* ∧ *set as* = *A*

lemma *asList*:
assumes *finite A* **shows** *distinct* (*asList A*) ∧ *set* (*asList A*) = *A*
unfolding *asList-def* **by** (*rule someI-ex*) (*metis assms finite-distinct-list*)

lemmas *distinct-asList* = *asList*[*THEN conjunct1*]
lemmas *set-asList* = *asList*[*THEN conjunct2*]

lemma *map-sdrop*[*simp*]: *sdrop 0* = *id*
by (*auto intro*: *ext*)

lemma *stl-o-sdrop*[*simp*]: *stl o sdrop n* = *sdrop* (*Suc n*)
by (*auto intro*: *ext*)

lemma *sdrop-o-stl*[*simp*]: *sdrop n o stl* = *sdrop* (*Suc n*)
by (*auto intro*: *ext*)

lemma *hd-stake*[*simp*]: $i > 0 \Longrightarrow hd\ (stake\ i\ \pi) = shd\ \pi$
by (*cases i*) *auto*

3 Shallow embedding of HyperCTL*

We define a notion of "shallow" HyperCTL* formula (sfmla) that captures HyperCTL* binders as meta-level HOL binders. We also define a proof system for this shallow embedding.

3.1 Kripke structures and paths

type-synonym (*'state*,*'aprop*) *path* = (*'state* × *'aprop set*) *stream*

abbreviation *stateOf* **where** $stateOf\ \pi \equiv fst\ (shd\ \pi)$
abbreviation *apropsOf* **where** $apropsOf\ \pi \equiv snd\ (shd\ \pi)$

locale *Kripke* =
  fixes S :: *'state set* **and** *s0* :: *'state* **and** δ :: *'state* ⇒ *'state set*
  and AP :: *'aprop set* **and** L :: *'state* ⇒ *'aprop set*
  assumes *s0*: $s0 \in S$ **and** δ: $\bigwedge s.\ s \in S \Longrightarrow \delta\ s \subseteq S$
  and L : $\bigwedge s.\ s \in S \Longrightarrow L\ s \subseteq AP$
begin

Well-formed paths

coinductive *wfp* :: *'aprop set* ⇒ (*'state*,*'aprop*) *path* ⇒ *bool*
for AP' :: *'aprop set*
where
intro:
$[\![s \in S;\ A \subseteq AP';\ A \cap AP = L\ s;\ stateOf\ \pi \in \delta\ s;\ wfp\ AP'\ \pi]\!]$
$\Longrightarrow$
$wfp\ AP'\ ((s,A)\ \#\#\ \pi)$

lemma *wfp*:
$wfp\ AP'\ \pi \longleftrightarrow$
$(\forall\ i.\ fst\ (\pi\ !!\ i) \in S \wedge snd\ (\pi\ !!\ i) \subseteq AP' \wedge$
$\quad snd\ (\pi\ !!\ i) \cap AP = L\ (fst\ (\pi\ !!\ i)) \wedge$
$\quad fst\ (\pi\ !!\ (Suc\ i)) \in \delta\ (fst\ (\pi\ !!\ i))$
$)$
(**is** $?L \longleftrightarrow (\forall\ i.\ ?R\ i)$)
proof (*intro iffI allI*)
  fix *i* **assume** *?L* **thus** *?R i*
  apply(*induction i arbitrary*: π)
  by (*metis snth.simps fst-conv snd-conv stream.sel wfp.cases*)+
next
  assume *R*: $\forall\ i.\ ?R\ i$ **thus** *?L*

apply (*coinduct*)
using *s0 fst-conv snd-conv snth.simps stream.sel*
by (*metis inf-commute stream.collapse surj-pair*)
qed

lemma *wfp-sdrop*[*simp*]:
wfp AP' π ⟹ wfp AP' (sdrop i π)
unfolding *wfp* **by** *simp* (*metis sdrop-add sdrop-simps*(*1*))

end-of-context Kripke

3.2 Shallow representations of formulas

A shallow (representation of a) HyperCTL* formula will be a predicate on lists of paths. The atomic formulas (operator *atom*) are parameterized by atomic propositions (as customary in temporal logic), and additionally by a number indicating the position, in the list of paths, of the path to which the atomic proposition refers – for example, *atom a i* holds for the list of paths πl just in case proposition a holds at the first state of $\pi l!i$, the i'th path in πl. The temporal operators *next* and *until* act on all the paths of the argument list πl synchronously. Finally, the existential quantifier refers to the existence of a path whose origin state is that of the last path in πl.

As an example: *exi* (*exi* (*until* (*atom a* 0) (*atom b* 1))) holds for the empty list iff there exist two paths ρ_0 and ρ_1 such that, synchronously, a holds on ρ_0 until b holds on ρ_1. Another example will be the formula encoding Goguen-Meseguer noninterference.

Shallow HyperCTL* formulas:

type-synonym ('*state*,'*aprop*) *sfmla* = ('*state*,'*aprop*) *path list* ⇒ *bool*

locale *Shallow* = *Kripke S s0 δ AP L*
for *S* :: '*state set* **and** *s0* :: '*state* **and** *δ* :: '*state* ⇒ '*state set*
and *AP* :: '*aprop set* **and** *L* :: '*state* ⇒ '*aprop set*
+
fixes *AP'* **assumes** *AP-AP'*: *AP* ⊆ *AP'*
begin

Primitive operators

definition *fls* :: ('*state*,'*aprop*) *sfmla* **where**
fls πl ≡ *False*

definition *atom* :: '*aprop* ⇒ *nat* ⇒ ('*state*,'*aprop*) *sfmla* **where**
atom a i πl ≡ *a* ∈ *apropsOf* (*πl*!*i*)

definition *neg* :: ('*state*,'*aprop*) *sfmla* ⇒ ('*state*,'*aprop*) *sfmla* **where**
neg φ πl ≡ ¬ *φ πl*

definition *dis* :: ('*state*,'*aprop*) *sfmla* ⇒ ('*state*,'*aprop*) *sfmla* ⇒ ('*state*,'*aprop*) *sfmla* **where**

dis φ ψ $\pi l \equiv \varphi$ $\pi l \vee \psi$ πl

definition *next* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
next φ $\pi l \equiv \varphi$ (*map stl* πl)

definition *until* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
until φ ψ $\pi l \equiv$
∃ *i*. ψ (*map* (*sdrop i*) πl) ∧ (∀ *j* ∈{*0*..<*i*}. φ (*map* (*sdrop j*) πl))

definition *exii* :: (*'state*,*'aprop*) *sfmla* ⇒ (*'state*,*'aprop*) *sfmla* **where**
exii φ $\pi l \equiv$
∃ π. *wfp AP'* π ∧ *stateOf* π = (*if* $\pi l \neq$ [] *then stateOf* (*last* πl) *else s0*)
∧ φ (πl @ [π])

definition *exi* :: ((*'state*,*'aprop*) *path* ⇒ (*'state*,*'aprop*) *sfmla*) ⇒ (*'state*,*'aprop*) *sfmla* **where**
exi F $\pi l \equiv$
∃ π. *wfp AP'* π ∧ *stateOf* π = (*if* $\pi l \neq$ [] *then stateOf* (*last* πl) *else s0*)
∧ *F* π πl

Derived operators

definition *tr* ≡ *neg fls*
definition *con* φ $\psi \equiv$ *neg* (*dis* (*neg* φ) (*neg* ψ))
definition *imp* φ $\psi \equiv$ *dis* (*neg* φ) ψ
definition *eq* φ $\psi \equiv$ *con* (*imp* φ ψ) (*imp* ψ φ)
definition *fall F* ≡ *neg* (*exi* (λ π. *neg* (*F* π)))
definition *ev* $\varphi \equiv$ *until tr* φ
definition *alw* $\varphi \equiv$ *neg* (*ev* (*neg* φ))
definition *wuntil* φ $\psi \equiv$ *dis* (*until* φ ψ) (*alw* φ)

lemmas *main-op-defs* =
fls-def atom-def neg-def dis-def next-def until-def exi-def

lemmas *der-op-defs* =
tr-def con-def imp-def eq-def ev-def alw-def wuntil-def fall-def

lemmas *op-defs* = *main-op-defs der-op-defs*

3.3 Reasoning rules

We provide introduction, elimination, unfolding and (co)induction rules for the connectives and quantifiers.

Boolean operators

lemma *fls-elim*[*elim*!]:
assumes *fls* πl **shows** φ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *tr-intro*[*intro*!, *simp*]: *tr* πl
unfolding *op-defs* **by** *auto*

lemma *dis-introL*[*intro*]:
assumes $\varphi\ \pi l$ **shows** *dis* $\varphi\ \psi\ \pi l$
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *dis-introR*[*intro*]:
assumes $\psi\ \pi l$ **shows** *dis* $\varphi\ \psi\ \pi l$
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *dis-elim*[*elim*]:
assumes *dis* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l \Longrightarrow \chi$ **and** $\psi\ \pi l \Longrightarrow \chi$
shows χ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *con-intro*[*intro*!]:
assumes $\varphi\ \pi l$ **and** $\psi\ \pi l$ **shows** *con* $\varphi\ \psi\ \pi l$
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *con-elim*[*elim*]:
assumes *con* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l \Longrightarrow \psi\ \pi l \Longrightarrow \chi$ **shows** χ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *neg-intro*[*intro*!]:
assumes $\varphi\ \pi l \Longrightarrow$ *False* **shows** *neg* $\varphi\ \pi l$
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *neg-elim*[*elim*]:
assumes *neg* $\varphi\ \pi l$ **and** $\varphi\ \pi l$ **shows** χ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *imp-intro*[*intro*!]:
assumes $\varphi\ \pi l \Longrightarrow \psi\ \pi l$ **shows** *imp* $\varphi\ \psi\ \pi l$
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *imp-elim*[*elim*]:
assumes *imp* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l$ **and** $\psi\ \pi l \Longrightarrow \chi$ **shows** χ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *imp-mp*[*elim*]:
assumes *imp* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l$ **shows** $\psi\ \pi l$
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *eq-intro*[*intro*!]:
assumes $\varphi\ \pi l \Longrightarrow \psi\ \pi l$ **and** $\psi\ \pi l \Longrightarrow \varphi\ \pi l$ **shows** *eq* $\varphi\ \psi\ \pi l$
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *eq-elimL*[*elim*]:
assumes *eq* $\varphi\ \psi\ \pi l$ **and** $\varphi\ \pi l$ **and** $\psi\ \pi l \Longrightarrow \chi$ **shows** χ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *eq-elimR*[*elim*]:
assumes *eq* φ ψ πl **and** ψ πl **and** φ $\pi l \Longrightarrow \chi$ **shows** χ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *eq-equals*: *eq* φ ψ $\pi l \longleftrightarrow \varphi$ $\pi l = \psi$ πl
by (*metis eq-elimL eq-elimR eq-intro*)

Quantifiers

lemma *exi-intro*[*intro*]:
assumes *wfp* AP' π
and $\pi l \neq [] \Longrightarrow$ *stateOf* π = *stateOf* (*last* πl)
and $\pi l = [] \Longrightarrow$ *stateOf* π = *s0*
and F π πl
shows *exi* F πl
using *assms* **unfolding** *exi-def* **by** *auto*

lemma *exi-elim*[*elim*]:
assumes *exi* F πl
and
$\bigwedge \pi$. ⟦*wfp* AP' π; $\pi l \neq [] \Longrightarrow$ *stateOf* π = *stateOf* (*last* πl); $\pi l = [] \Longrightarrow$ *stateOf* π = *s0*; F π πl⟧ $\Longrightarrow \chi$
shows χ
using *assms* **unfolding** *exi-def* **by** *auto*

lemma *fall-intro*[*intro*]:
assumes
$\bigwedge \pi$. ⟦*wfp* AP' π; $\pi l \neq [] \Longrightarrow$ *stateOf* π = *stateOf* (*last* πl) ; $\pi l = [] \Longrightarrow$ *stateOf* π = *s0*⟧
$\Longrightarrow$ F π πl
shows *fall* F πl
using *assms* **unfolding** *fall-def* **by** (*metis exi-def neg-def*)

lemma *fall-elim*[*elim*]:
assumes *fall* F πl
and
(⋀π. ⟦*wfp* AP' π; $\pi l \neq [] \Longrightarrow$ *stateOf* π = *stateOf* (*last* πl); $\pi l = [] \Longrightarrow$ *stateOf* π = *s0*⟧
$\Longrightarrow$ F π πl)
$\Longrightarrow \chi$
shows χ
using *assms* **unfolding** *fall-def*
by (*metis exi-def neg-elim neg-intro*)

Temporal connectives

lemma *next-intro*[*intro*]:
assumes φ (*map stl* πl) **shows** *next* φ πl
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *next-elim*[*elim*]:
assumes *next* φ πl **and** φ (*map stl* πl) $\Longrightarrow \chi$ **shows** χ
using *assms* **unfolding** *op-defs* **by** *auto*

lemma *until-introR*[*intro*]:
assumes $\psi\ \pi l$ **shows** *until* $\varphi\ \psi\ \pi l$
using *assms* **unfolding** *op-defs* **by** (*auto intro*: *exI*[*of - 0*])

lemma *until-introL*[*intro*]:
assumes φ: $\varphi\ \pi l$ **and** *u*: *until* $\varphi\ \psi$ (*map stl* πl)
shows *until* $\varphi\ \psi\ \pi l$
proof–
obtain *i* **where** ψ: ψ (*map* (*sdrop* (*Suc i*)) πl) **and** *1*: $\forall j{\in}\{0..{<}i\}$. φ (*map* (*sdrop* (*Suc j*)) πl)
using *u* **unfolding** *op-defs* **by** *auto*
{fix *j* **assume** $j \in \{0..{<}Suc\ i\}$
hence φ (*map* (*sdrop j*) πl) **using** *1* φ **by** (*cases j*) *auto*
}
thus *?thesis* **using** ψ **unfolding** *op-defs* **by** *auto*
qed

The elimination rules for until and eventually are induction rules.

lemma *until-induct*[*induct pred*: *until*, *consumes 1*, *case-names Base Step*]:
assumes *u*: *until* $\varphi\ \psi\ \pi l$
and *b*: $\bigwedge \pi l.\ \psi\ \pi l \Longrightarrow \chi\ \pi l$
and *i*: $\bigwedge \pi l.\ [\![\varphi\ \pi l;\ until\ \varphi\ \psi\ (map\ stl\ \pi l);\ \chi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi\ \pi l$
shows $\chi\ \pi l$
proof–
obtain *i* **where** ψ: ψ (*map* (*sdrop i*) πl) **and** *1*: $\forall j{\in}\{0..{<}i\}$. φ (*map* (*sdrop j*) πl)
using *u* **unfolding** *until-def next-def* **by** *auto*
{fix *k* **assume** *k*: $k \le i$
hence *until* $\varphi\ \psi$ (*map* (*sdrop k*) πl) $\wedge$ χ (*map* (*sdrop k*) πl)
proof (*induction i*−*k arbitrary*: *k*)
case *0* **hence** *k*=*i* **by** *auto*
with *b*[*OF* ψ] *u* ψ **show** *?case* **by** (*auto intro*: *until-introR*)
next
case (*Suc ii*) **let** $?\pi l' = map\ (sdrop\ k)\ \pi l$
have *until* $\varphi\ \psi$ (*map stl* $?\pi l'$) $\wedge$ χ (*map stl* $?\pi l'$) **using** *Suc* **by** *auto*
moreover have φ $?\pi l'$ **using** *1 Suc* **by** *auto*
ultimately show *?case* **using** *i* **by** *auto*
qed
}
from *this*[*of 0*] **show** *?thesis* **by** *simp*
qed

lemma *until-unfold*:
until $\varphi\ \psi\ \pi l$ = ($\psi\ \pi l \vee \varphi\ \pi l \wedge$ *until* $\varphi\ \psi$ (*map stl* πl)) (**is** *?L* = *?R*)
proof
assume *?L* **thus** *?R* **by** *induct auto*
qed *auto*

lemma *ev-introR*[*intro*]:
assumes $\varphi\ \pi l$ **shows** *ev* $\varphi\ \pi l$

using *assms* **unfolding** *ev-def* **by** (*auto intro*: *until-introR*)

lemma *ev-introL*[*intro*]:
assumes *ev* φ (*map stl* πl) **shows** *ev* φ πl
using *assms* **unfolding** *ev-def* **by** (*auto intro*: *until-introL*)

lemma *ev-induct*[*induct pred*: *ev*, *consumes 1*, *case-names Base Step*]:
assumes *ev* φ πl **and** $\bigwedge \pi l.\ \varphi\ \pi l \Longrightarrow \chi\ \pi l$
and $\bigwedge \pi l.\ [\![ev\ \varphi\ (map\ stl\ \pi l);\ \chi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi\ \pi l$
shows $\chi\ \pi l$
using *assms* **unfolding** *ev-def* **by** *induct* (*auto simp*: *assms*)

lemma *ev-unfold*:
ev φ πl = (φ πl $\vee$ *ev* φ (*map stl* πl))
unfolding *ev-def* **by** (*metis tr-intro until-unfold*)

lemma *ev*: *ev* φ πl $\longleftrightarrow$ ($\exists$ *i*. φ (*map* (*sdrop i*) πl))
unfolding *ev-def until-def* **by** *auto*

The introduction rules for always and weak until are coinduction rules.

lemma *alw-coinduct*[*coinduct pred*: *alw*, *consumes 1*, *case-names Hyp*]:
assumes $\chi\ \pi l$
and $\bigwedge \pi l.\ \chi\ \pi l \Longrightarrow alw\ \varphi\ \pi l \vee (\varphi\ \pi l \wedge \chi\ (map\ stl\ \pi l))$
shows *alw* φ πl
proof−
 {assume *ev* (*neg* φ) πl
  hence ¬ χ πl
  apply *induct*
  using *assms* **unfolding** *op-defs* **by** *auto* (*metis assms alw-def ev-def neg-def until-introR*)
 }
 thus *?thesis* **using** *assms* **unfolding** *op-defs* **by** *auto*
qed

lemma *alw-elim*[*elim*]:
assumes *alw* φ πl
and $[\![\varphi\ \pi l;\ alw\ \varphi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi$
shows χ
using *assms* **unfolding** *alw-def* **by**(*auto elim*: *ev-introR simp*: *neg-def*)

lemma *alw-destL*: *alw* φ $\pi l \Longrightarrow \varphi\ \pi l$ **by** *auto*
lemma *alw-destR*: *alw* φ $\pi l \Longrightarrow alw\ \varphi\ (map\ stl\ \pi l)$ **by** *auto*

lemma *alw-unfold*:
alw φ πl = (φ πl $\wedge$ *alw* φ (*map stl* πl))
by (*metis alw-def ev-unfold neg-elim neg-intro*)

lemma *alw*: *alw* φ πl $\longleftrightarrow$ ($\forall$ *i*. φ (*map* (*sdrop i*) πl))
unfolding *alw-def ev neg-def* **by** *simp*

lemma *sdrop-imp-alw*:
assumes $\bigwedge i.\ (\bigwedge j.\ j \leq i \Longrightarrow \varphi\ [sdrop\ j\ \pi,\ sdrop\ j\ \pi']) \Longrightarrow \psi\ [sdrop\ i\ \pi,\ sdrop\ i\ \pi']$
shows *imp* (*alw* φ) (*alw* ψ) $[\pi,\ \pi']$
using *assms* **by**(*auto simp*: *alw*)

lemma *wuntil-coinduct*[*coinduct pred*: *wuntil*, *consumes 1*, *case-names Hyp*]:
assumes χ: $\chi\ \pi l$
and *0*: $\bigwedge \pi l.\ \chi\ \pi l \Longrightarrow \psi\ \pi l \vee (\varphi\ \pi l \wedge \chi\ (map\ stl\ \pi l))$
shows *wuntil* $\varphi\ \psi\ \pi l$
proof−
 {assume ¬ *until* $\varphi\ \psi\ \pi l \wedge \chi\ \pi l$
  hence *alw* $\varphi\ \pi l$
  apply *coinduct* **using** *0* **by** (*auto intro*: *until-introL until-introR*)
 }
 thus *?thesis* **using** χ **unfolding** *wuntil-def dis-def* **by** *auto*
qed

lemma *wuntil-elim*[*elim*]:
assumes *w*: *wuntil* $\varphi\ \psi\ \pi l$
and *1*: $\psi\ \pi l \Longrightarrow \chi$
and *2*: $[\![\varphi\ \pi l;\ wuntil\ \varphi\ \psi\ (map\ stl\ \pi l)]\!] \Longrightarrow \chi$
shows χ
proof(*cases alw* $\varphi\ \pi l$)
 case *True*
 thus *?thesis* **apply** *standard* **using** *2* **unfolding** *wuntil-def* **by** *auto*
next
 case *False*
 hence *until* $\varphi\ \psi\ \pi l$ **using** *w* **unfolding** *wuntil-def dis-def* **by** *auto*
 thus *?thesis* **by** (*metis assms dis-introL until-unfold wuntil-def*)
qed

lemma *wuntil-unfold*:
wuntil $\varphi\ \psi\ \pi l = (\psi\ \pi l \vee \varphi\ \pi l \wedge wuntil\ \varphi\ \psi\ (map\ stl\ \pi l))$
by (*metis alw-unfold dis-def until-unfold wuntil-def*)

3.4 More derived operators

The conjunction of an arbitrary set of formulas:

definition *scon* ::
('*state*,'*aprop*) *sfmla set* ⇒ ('*state*,'*aprop*) *sfmla* **where**
scon $\varphi s\ \pi l \equiv \forall\ \varphi \in \varphi s.\ \varphi\ \pi l$

lemma *mcon-intro*[*intro*!]:
assumes $\bigwedge \varphi.\ \varphi \in \varphi s \Longrightarrow \varphi\ \pi l$ **shows** *scon* $\varphi s\ \pi l$
using *assms* **unfolding** *scon-def* **by** *auto*

lemma *scon-elim*[*elim*]:
assumes *scon* $\varphi s\ \pi l$ **and** $(\bigwedge \varphi.\ \varphi \in \varphi s \Longrightarrow \varphi\ \pi l) \Longrightarrow \chi$
shows χ

using *assms* **unfolding** *scon-def* **by** *auto*

Double-binding forall:

definition *fall2 F* ≡ *fall* (λ π. *fall* (*F* π))

lemma *fall2-intro*[*intro*]:
assumes
⋀ π π′. ⟦*wfp AP′* π; *wfp AP′* π′;
π*l* ≠ [] ⟹ *stateOf* π = *stateOf* (*last* π*l*);
π*l* = [] ⟹ *stateOf* π = *s0*;
stateOf π′ = *stateOf* π
⟧
⟹ *F* π π′ π*l*
shows *fall2 F* π*l*
using *assms* **unfolding** *fall2-def* **by** (*auto intro*!: *fall-intro*)

lemma *fall2-elim*[*elim*]:
assumes *fall2 F* π*l*
and
(⋀π π′. ⟦*wfp AP′* π; *wfp AP′* π′;
π*l* ≠ [] ⟹ *stateOf* π = *stateOf* (*last* π*l*); π*l* = [] ⟹ *stateOf* π = *s0*;
stateOf π′ = *stateOf* π
⟧
⟹ *F* π π′ π*l*)
⟹ χ
shows χ
using *assms* **unfolding** *fall2-def* **by** (*auto elim*!: *fall-elim*) (*metis fall-elim*)

end-of-context Shallow

4 Noninterference à la Goguen and Meseguer

4.1 Goguen-Meseguer noninterference

Definition

locale *GM-sec-model* =
fixes *st0* :: *'St*
and *do* :: *'St* ⇒ *'U* ⇒ *'C* ⇒ *'St*
and *out* :: *'St* ⇒ *'U* ⇒ *'Out*
and *GH* :: *'U set*
and *GL* :: *'U set*
begin

Extension of "do" to sequences of pairs (user, command):

fun *doo* :: *'St* ⇒ (*'U* × *'C*) *list* ⇒ *'St* **where**
doo st [] = *st*
|*doo st* ((*u*,*c*) # *ucl*) = (*doo* (*do st u c*) *ucl*)

definition *purge* :: *'U set* ⇒ (*'U* × *'C*) *list* ⇒ (*'U* × *'C*) *list* **where**
purge G ucl ≡ *filter* (λ (*u*,*c*). *u* ∉ *G*) *ucl*

lemma *purge-Nil*[*simp*]: *purge G* [] = []
and *purge-Cons-in*[*simp*]: *u* ∉ *G* ⟹ *purge G* ((*u*,*c*) # *ucl*) = (*u*,*c*) # *purge G ucl*
and *purge-Cons-notIn*[*simp*]: *u* ∈ *G* ⟹ *purge G* ((*u*,*c*) # *ucl*) = *purge G ucl*
unfolding *purge-def* **by** *auto*

lemma *purge-append*:
purge G (*ucl1* @ *ucl2*) = *purge G ucl1* @ *purge G ucl2*
unfolding *purge-def* **by** (*metis filter-append*)

definition *nonint* :: *bool* **where**
nonint ≡ ∀ *ucl*. ∀ *u* ∈ *GL*. *out* (*doo st0 ucl*) *u* = *out* (*doo st0* (*purge GH ucl*)) *u*

end-of-context GM-sec-model

4.2 Specialized Kripke structures

As a preparation for representing noninterference in HyperCTL*, we define a specialized notion of Kripke structure. It is enriched with the following date: two binary state predicates f and g, intuitively capturing high-input and low-output equivalence, respectively; a set Sink of states immediately accessible from any state and such that, for the states in Sink, there exist self-transitions and f holds.

This specialized structure, represented by the locale Shallow-Idle, is an auxiliary that streamlines our proofs, easing the connection between finite paths (specific to Goguen-Meseguer noninterference) and infinite paths (specific to the HyperCTL* semantics). The desired Kripke structure produced from a Goguen-Meseguer model will actually be such a specialized structure.

locale *Shallow-Idle* = *Shallow S s0* δ *AP*
for *S* :: *'state set* **and** *s0* :: *'state* **and** δ :: *'state* ⇒ *'state set*
and *AP* :: *'aprop set*
and *f* :: *'state* ⇒ *'state* ⇒ *bool* **and** *g* :: *'state* ⇒ *'state* ⇒ *bool*
and *Sink* :: *'state set*
+
assumes *Sink-S*: *Sink* ⊆ *S*
and *Sink*: ⋀ *s*. *s* ∈ *S* ⟹ ∃ *s'*. *s'* ∈ δ *s* ∩ *Sink*
and *Sink-idle*: ⋀ *s*. *s* ∈ *Sink* ⟹ *s* ∈ δ *s*
and *Sink-f*: ⋀ *s1 s2*. {*s1*,*s2*} ⊆ *Sink* ⟹ *f s1 s2*
begin

definition *toSink s* ≡ *SOME s'*. *s'* ∈ δ *s* ∩ *Sink*

lemma *toSink*: *s* ∈ *S* ⟹ *toSink s* ∈ δ *s* ∩ *Sink*
unfolding *toSink-def* **by** (*metis Sink someI*)

lemma *fall2-imp-alw*:
fall2 ($\lambda\ \pi'\ \pi\ \pi l$. *imp* (*alw* ($\varphi\ \pi l$)) (*alw* ($\psi\ \pi l$)) ($\pi l$ @ [π,π'])) []
$\longleftrightarrow$
($\forall\ \pi\ \pi'$. *wfp* $AP'\ \pi \wedge$ *wfp* $AP'\ \pi' \wedge$ *stateOf* π = *s0* $\wedge$ *stateOf* π' = *s0*
$\longrightarrow$ *imp* (*alw* (φ [])) (*alw* (ψ [])) [π,π']
)
by (*auto intro*!: *fall2-intro imp-intro elim*!: *fall2-elim imp-elim*) (*metis imp-elim*)+

lemma *wfp-stateOf-shift-stake-sconst*:
fixes π *i*
defines $\pi 1 \equiv$ *shift* (*stake* (*Suc i*) π) (*sconst* (*toSink* (*fst* (π !! *i*)), *L* (*toSink* (*fst* (π !! *i*)))))
assumes π: *wfp* $AP'\ \pi$
shows *wfp* $AP'\ \pi 1 \wedge$ *stateOf* $\pi 1$ = *stateOf* π
proof
have $\pi 1$-*less*[*simp*]: $\bigwedge$ *k*. *k* < *Suc i* $\Longrightarrow$ $\pi 1$!! *k* = π !! *k*
and $\pi 1$-*geq*[*simp*]: $\bigwedge$ *k*. *k* > *i* $\Longrightarrow$ $\pi 1$!! *k* = (*toSink* (*fst* (π !! *i*)) , *L* (*toSink* (*fst* (π !! *i*))))
unfolding $\pi 1$-*def* **by** (*auto simp del*: *stake.simps*)
{fix *k* **have** *fst* ($\pi 1$!! *Suc k*) $\in$ δ (*fst* ($\pi 1$!! *k*))
proof(*cases k* < *Suc i*)
case *True*
hence *0*: $\pi 1$!! *k* = π !! *k* **by** *simp*
show *?thesis*
proof(*cases k* < *i*)
case *True* **hence** *1*: *Suc k* < *Suc i* **by** *simp*
show *?thesis* **using** π **unfolding** $\pi 1$-*less*[*OF 1*] *0 wfp* **by** *auto*
next
case *False* **hence** *1*: *Suc k* > *i* **and** *k*: *k* = *i* **using** *True* **by** *simp-all*
show *?thesis* **using** π **unfolding** $\pi 1$-*geq*[*OF 1*] *0 wfp* **unfolding** *k* **by** (*metis IntD1 fstI toSink*)
qed
next
case *False*
hence *k*: *k* > *i* **and** *sk*: *Suc k* > *i* **by** *auto*
show *?thesis* **unfolding** $\pi 1$-*geq*[*OF k*] $\pi 1$-*geq*[*OF sk*] **using** π *wfp Sink-idle toSink* **by** *auto*
qed
}
moreover
{fix *k* **have** *fst* ($\pi 1$!! *k*) $\in$ *S* $\wedge$ *snd* ($\pi 1$!! *k*) $\subseteq$ AP' $\wedge$ *snd* ($\pi 1$!! *k*) $\cap$ *AP* = *L* (*fst* ($\pi 1$!! *k*))
apply(*cases k* < *Suc i*,*simp-all*)
by (*metis* (*lifting*, *no-types*) π *wfp AP-AP' IntD1 L* δ *inf.orderE order-trans rev-subsetD toSink*)+
}
ultimately show *wfp* $AP'\ \pi 1$ **unfolding** *wfp* **by** *auto*
show *stateOf* $\pi 1$ = *stateOf* π
by (*metis* $\pi 1$-*def shift.simps*(*2*) *stake.simps*(*2*) *stream.sel*(*1*))
qed

lemma *fall2-imp-alw-index*:
assumes *0*: $\bigwedge\ \pi\ \pi'$. *wfp* $AP'\ \pi \wedge$ *wfp* $AP'\ \pi' \longrightarrow$

φ [] [π,π'] = f (*stateOf* π) (*stateOf* π') ∧
ψ [] [π,π'] = g (*stateOf* π) (*stateOf* π')

shows

fall2 (λ π' π πl. *imp* (*alw* (φ πl)) (*alw* (ψ πl)) (πl @ [π,π'])) []
⟷
(∀ π π'. *wfp AP'* π ∧ *wfp AP'* π' ∧ *stateOf* π = *s0* ∧ *stateOf* π' = *s0*
⟶
(∀ *i*. (∀ *j* ≤ *i*. *f* (*fst* (π !! *j*)) (*fst* (π' !! *j*))) ⟶ *g* (*fst* (π !! *i*)) (*fst* (π' !! *i*)))
)
(**is** *?L* ⟷ *?R*)
proof−
have *1*: ⋀ *i* π π'. *wfp AP'* π ∧ *wfp AP'* π' ⟶
f (*fst* (π !! *i*)) (*fst* (π' !! *i*)) = φ [] [*sdrop i* π, *sdrop i* π'] ∧
g (*fst* (π !! *i*)) (*fst* (π' !! *i*)) = ψ [] [*sdrop i* π, *sdrop i* π']
using *0* **by** *auto*
show *?thesis* **unfolding** *fall2-imp-alw* **proof**(*intro iffI allI impI*, *elim conjE*)
fix π π' *i*
assume *L*: ∀ π π'. *wfp AP'* π ∧ *wfp AP'* π' ∧ *stateOf* π = *s0* ∧ *stateOf* π' = *s0*
⟶ *imp* (*alw* (φ [])) (*alw* (ψ [])) [π,π']
and $\pi\pi'$[*simp*]: *wfp AP'* π *wfp AP'* π' *stateOf* π = *s0 stateOf* π' = *s0*
and φ: ∀*j*≤*i*. *f* (*fst* (π !! *j*)) (*fst* (π' !! *j*))
have $\pi\pi' i$[*simp*]: ⋀ *i*. *wfp AP'* (*sdrop i* π) ∧ *wfp AP'* (*sdrop i* π') **by** (*metis* $\pi\pi'$ *wfp-sdrop*)
define $\pi 1$ **where** $\pi 1$ = *shift* (*stake* (*Suc i*) π) (*sconst* (*toSink* (*fst* (π !! *i*)), *L* (*toSink* (*fst* (π !! *i*)))))
define $\pi 1'$ **where** $\pi 1'$ = *shift* (*stake* (*Suc i*) π') (*sconst* (*toSink* (*fst* (π' !! *i*)), *L* (*toSink* (*fst* (π' !! *i*)))))
have $\pi 1 \pi 1'$: *wfp AP'* $\pi 1$ ∧ *stateOf* $\pi 1$ = *s0* ∧ *wfp AP'* $\pi 1'$ ∧ *stateOf* $\pi 1'$ = *s0*
using *wfp-stateOf-shift-stake-sconst* **unfolding** $\pi 1$*-def* $\pi 1'$*-def* **by** *auto*
hence $\pi 1 \pi 1' i$: ⋀ *i*. *wfp AP'* (*sdrop i* $\pi 1$) ∧ *wfp AP'* (*sdrop i* $\pi 1'$) **by** (*metis* $\pi\pi'$ *wfp-sdrop*)
have *imp*: *imp* (*alw* (φ [])) (*alw* (ψ [])) [$\pi 1$,$\pi 1'$] **using** *L* $\pi 1 \pi 1'$ **by** *auto*
moreover have *alw* (φ []) [$\pi 1$,$\pi 1'$] **unfolding** *alw* **proof**
fix *k*
have *a*: *fst* (π !! *i*) ∈ *S* **and** *b*: *fst* (π' !! *i*) ∈ *S* **using** $\pi\pi'$ **unfolding** *wfp* **by** *auto*
thus φ [] (*map* (*sdrop k*) [$\pi 1$, $\pi 1'$])
using φ *0* $\pi 1 \pi 1' i$ **unfolding** $\pi 1$*-def* $\pi 1'$*-def*
apply(*cases k* < *Suc i*, *simp-all del*: *stake.simps*)
using *toSink*[*OF a*] *toSink*[*OF b*] *Sink-f* **by** *auto*
qed
ultimately have *alw* (ψ []) [$\pi 1$,$\pi 1'$] **by** *auto*
hence ψ [] [*sdrop i* $\pi 1$, *sdrop i* $\pi 1'$] **unfolding** *alw* **by** *simp*
hence *g* (*fst* ($\pi 1$!! *i*)) (*fst* ($\pi 1'$!! *i*)) **using** *0* $\pi 1 \pi 1' i$ **by** *simp*
thus *g* (*fst* (π !! *i*)) (*fst* (π' !! *i*))
unfolding $\pi 1$*-def* $\pi 1'$*-def* **by** (*auto simp del*: *stake.simps*)
qed(*auto simp*: *sdrop-imp-alw 1*)
qed

end-of-context Shallow-Idle

4.3 Faithful representation as a HyperCTL* property

Starting with a Goguen-Meseguer model, we will produce a specialized Kripke structure and a shallow HyperCTL* formula. Then we we will prove that the structure satisfies the formula iff the Goguen-Meseguer model satisfies noninterference.

The Kripke structure has two kinds of states: "idle" states storing Goguen-Meseguer states, and normal states storing Goguen-Meseguer states, users and commands: the former will be used for synchronization and the latter for Goguen-Meseguer steps. The Kripke labels store user-command actions and user-output observations.

datatype $('St,'U,'C)$ *state* =
isIdle: *Idle* (*getGMState*: $'St$) | *isState*: *State* (*getGMState*: $'St$) (*getGMUser*: $'U$) (*getGMCom*: $'C$)

datatype $('U,'C,'Out)$ *aprop* = *Last* $'U$ $'C$ | *Obs* $'U$ $'Out$

definition *getGMUserCom* **where** *getGMUserCom s* = (*getGMUser s*, *getGMCom s*)

lemma *getGMUserCom*[*simp*]: *getGMUserCom* (*State st u c*) = (*u*,*c*)
unfolding *getGMUserCom-def* **by** *auto*

context *GM-sec-model*
begin

primrec $L :: ('St,'U,'C)$ *state* $\Rightarrow$ $('U,'C,'Out)$ *aprop set* **where**
L (*Idle st*) = {*Obs u'* (*out st u'*) | *u'*. *True*}
|*L* (*State st u c*) = {*Last u c*} $\cup$ {*Obs u'* (*out st u'*) | *u'*. *True*}

Get the Goguen-Meseguer state:

primrec *getGMState* **where**
getGMState (*Idle st*) = *st*
|*getGMState* (*State st u c*) = *st*

lemma *Last-in-L*[*simp*]: *Last u c* $\in$ *L s* $\longleftrightarrow$ ($\exists$ *st*. *s* = *State st u c*)
by (*cases s*) *auto*

lemma *Obs-in-L*[*simp*]: *Obs u ou* $\in$ *L s* $\longleftrightarrow$ *ou* = *out* (*getGMState s*) *u*
by (*cases s*) *auto*

primrec $\delta :: ('St,'U,'C)$ *state* $\Rightarrow$ $('St,'U,'C)$ *state set* **where**
δ (*Idle st*) = {*Idle st*} $\cup$ {*State* (*do st u' c'*) *u' c'* | *u' c'*. *True*}
|δ (*State st u c*) = {*Idle st*} $\cup$ {*State* (*do st u' c'*) *u' c'* | *u' c'*. *True*}

abbreviation *s0* **where** *s0* $\equiv$ *State st0 any any*

definition $f :: ('a, 'U, 'b)$ *state* $\Rightarrow$ $('c, 'U, 'b)$ *state* $\Rightarrow$ *bool*
where
f s s' $\equiv$
$\forall$ *u c*. *u* $\notin$ *GH* $\longrightarrow$ (($\exists$ *st*. *s* = *State st u c*) $\longleftrightarrow$ ($\exists$ *st'*. *s'* = *State st' u c*))

definition $g :: ('St,\ 'a,\ 'b)\ state \Rightarrow ('St,\ 'c,\ 'd)\ state \Rightarrow bool$
where
$g\ s\ s' \equiv \forall\ u1.\ u1 \in GL \longrightarrow out\ (getGMState\ s)\ u1 = out\ (getGMState\ s')\ u1$

lemma *f-id*[*simp*,*intro*!]: $f\ s\ s$ **unfolding** *f-def* **by** *auto*

definition $Sink :: ('St,'U,'C)\ state\ set$
where
$Sink = \{Idle\ st \mid st\ .\ True\}$

end

sublocale *GM-sec-model* < *Shallow-Idle*
where $S = UNIV::('St,'U,'C)\ state\ set$
and $AP = UNIV :: ('U,'C,'Out)\ aprop\ set$ **and** $AP' = UNIV :: ('U,'C,'Out)\ aprop\ set$
and $s0 = s0$ **and** $L = L$ **and** $\delta = \delta$ **and** $f = f$ **and** $g = g$ **and** $Sink = Sink$
proof
  fix s **show** $\exists\, s'.\ s' \in \delta\ s \cap Sink$
  by (*rule exI*[*of - Idle* (*getGMState s*)]) (*cases s*, *auto simp*: *Sink-def*)
next
  fix s **assume** $s \in Sink$ **thus** $s \in \delta\ s$ **unfolding** *Sink-def* **by** (*cases s*) *auto*
next
  fix *s1 s2* **assume** $\{s1,\ s2\} \subseteq Sink$ **thus** $f\ s1\ s2$
  unfolding *Sink-def f-def* **by** *auto*
qed *auto*

context *GM-sec-model*
begin

lemma *apropsOf-L-stateOf*[*simp*]:
$wfp\ AP'\ \pi \Longrightarrow apropsOf\ \pi = L\ (stateOf\ \pi)$
unfolding *wfp* **by** (*metis Int-UNIV-right snth.simps*(*1*))

The equality of two states w.r.t. a given "last" user-command pair:

definition *eqOnUC* ::
$nat \Rightarrow nat \Rightarrow 'U \Rightarrow 'C \Rightarrow (('St,'U,'C)\ state,('U,'C,'Out)\ aprop)\ sfmla$
where
$eqOnUC\ i\ i'\ u\ c \equiv eq\ (atom\ (Last\ u\ c)\ i)\ (atom\ (Last\ u\ c)\ i')$

The equality of two states w.r.t. all their "last" user-command pairs with the user not in GH:

definition *eqButGH* ::
$nat \Rightarrow nat \Rightarrow (('St,'U,'C)\ state,('U,'C,'Out)\ aprop)\ sfmla$
where
$eqButGH\ i\ i' \equiv scon\ \{eqOnUC\ i\ i'\ u\ c \mid u\ c.\ (u,c) \in (UNIV - GH) \times UNIV\}$

The equality of two states w.r.t. a given "observed" user-observation pair:

definition *eqOnUOut* ::
$nat \Rightarrow nat \Rightarrow 'U \Rightarrow 'Out \Rightarrow (('St,'U,'C)\ state,('U,'C,'Out)\ aprop)\ sfmla$

where
eqOnUOut i i′ u ou ≡ *eq* (*atom* (*Obs u ou*) *i*) (*atom* (*Obs u ou*) *i′*)

The equality of two states w.r.t. all their "observed" user-observation pairs with the user in GL:

definition *eqOnGL* ::
nat ⇒ *nat* ⇒ (('*St*,'*U*,'*C*) *state*,('*U*,'*C*,'*Out*) *aprop*) *sfmla*
where
eqOnGL i i′ ≡ *scon* {*eqOnUOut i i′ u ou* | *u ou*. (*u*,*ou*) ∈ *GL* × *UNIV*}

lemma *eqOnUC-0-Suc0*[*simp*]:
assumes *wfp AP′ π* **and** *wfp AP′ π′*
shows
eqOnUC 0 (*Suc 0*) *u c* [*π*, *π′*]
⟷
((∃ *st*. *stateOf π* = *State st u c*) =
(∃ *st′*. *stateOf π′* = *State st′ u c*)
)
using *assms* **unfolding** *eqOnUC-def atom-def*[*abs-def*] *eq-equals* **by** *simp*

lemma *eqOnUOut-0-Suc0*[*simp*]:
assumes *wfp AP′ π* **and** *wfp AP′ π′*
shows
eqOnUOut 0 (*Suc 0*) *u ou* [*π*, *π′*]
⟷
(*ou* = *out* (*getGMState* (*stateOf π*)) *u* ⟷
ou = *out* (*getGMState* (*stateOf π′*)) *u*
)
using *assms* **unfolding** *eqOnUOut-def atom-def*[*abs-def*] *eq-equals* **by** *simp*

The (shallow) noninterference formula – it will be proved equivalent to nonint, the original statement of noninterference.

definition *nonintSfmla* :: (('*St*,'*U*,'*C*) *state*,('*U*,'*C*,'*Out*) *aprop*) *sfmla* **where**
nonintSfmla ≡
fall2 (λ *π′ π πl*.
imp (*alw* (*eqButGH* (*length πl*) (*Suc* (*length πl*))))
(*alw* (*eqOnGL* (*length πl*) (*Suc* (*length πl*))))
(*πl* @ [*π*,*π′*])
)

First, we show that nonintSfmla is equivalent to nonintSI, a variant of noninterference that speaks about Synchronized Infinite paths.

definition *nonintSI* :: *bool* **where**
nonintSI ≡
∀ *π π′*. *wfp UNIV π* ∧ *wfp UNIV π′* ∧ *stateOf π* = *s0* ∧ *stateOf π′* = *s0*
⟶
(∀ *i*. (∀ *j* ≤ *i*. *f* (*fst* (*π* !! *j*)) (*fst* (*π′* !! *j*))) ⟶ *g* (*fst* (*π* !! *i*)) (*fst* (*π′* !! *i*)))

lemma *nonintSfmla-nonintSI*: *nonintSfmla* [] ⟷ *nonintSI*

proof–
define φ **where** $\varphi\ \pi l = eqButGH\ (length\ \pi l)\ (Suc\ (length\ \pi l))$
for $\pi l :: (('St,'U,'C)\ state,('U,'C,'Out)\ aprop)\ path\ list$
define ψ **where** $\psi\ \pi l = eqOnGL\ (length\ \pi l)\ (Suc\ (length\ \pi l))$
for $\pi l :: (('St,'U,'C)\ state,('U,'C,'Out)\ aprop)\ path\ list$
have $\bigwedge\ \pi\ \pi'.\ wfp\ UNIV\ \pi \wedge wfp\ UNIV\ \pi' \longrightarrow$
$\varphi\ []\ [\pi,\pi'] = f\ (stateOf\ \pi)\ (stateOf\ \pi') \wedge$
$\psi\ []\ [\pi,\pi'] = g\ (stateOf\ \pi)\ (stateOf\ \pi')$
unfolding *φ-def ψ-def f-def g-def eqButGH-def eqOnGL-def*
by (*fastforce simp add*: *scon-def eqOnUC-0-Suc0*)
from *fall2-imp-alw-index*[*of* φ ψ, *OF this*]
show *?thesis* **unfolding** *nonintSfmla-def nonintSI-def φ-def ψ-def* **.**
qed

In turn, nonintSI will be shown equivalent to nonintS, a variant speaking about Synchronized finite paths. To this end, we introduce a notion of well-formed finite path (wffp) – besides finiteness, another difference from the previously defined infinite paths is that, thanks to the fact that here AP coincides with AP', paths are mere sequences of states as opposed to pairs (state,set of atomic predicates).

inductive *wffp* :: $('St,'U,'C)\ state\ list \Rightarrow bool$
where
Singl[*simp,intro*!]: *wffp* [*s*]
|
Cons[*intro*]:
$[\![s' \in \delta\ s;\ wffp\ (s'\ \#\ sl)]\!]$
$\Longrightarrow$
$wffp\ (s\ \#\ s'\ \#\ sl)$

lemma *wffp-induct2*[*consumes 1, case-names Singl Cons*]:
assumes *wffp sl*
and $\bigwedge\ s.\ P\ [s]$
and $\bigwedge\ s\ sl.\ [\![hd\ sl \in \delta\ s;\ wffp\ sl;\ P\ sl]\!] \Longrightarrow P\ (s\ \#\ sl)$
shows *P sl*
using *assms* **by** *induct auto*

definition *nonintS* :: *bool* **where**
nonintS $\equiv$
$\forall\ sl\ sl'.\ wffp\ sl \wedge wffp\ sl' \wedge hd\ sl = s0 \wedge hd\ sl' = s0 \wedge$
$list\text{-}all2\ f\ sl\ sl' \longrightarrow g\ (last\ sl)\ (last\ sl')$

lemma *wffp-NE*: **assumes** *wffp sl* **shows** $sl \neq []$
using *assms* **by** *induct auto*

lemma *wffp*:
$wffp\ sl \longleftrightarrow sl \neq [] \wedge (\forall\ i.\ Suc\ i < length\ sl \longrightarrow sl!(Suc\ i) \in \delta(sl!i))$
(**is** $?L \longleftrightarrow ?A \wedge (\forall\ i.\ ?R\ i)$)
proof (*intro iffI allI conjI*)
fix *i* **assume** *?L* **thus** *?R i*
proof (*induct arbitrary*: *i*)

```
    case (Cons s' s sl i) thus ?case by(cases i) auto
  qed auto
next
  assume ?A ∧ (∀ i. ?R i) thus ?L proof(induct sl)
    case (Cons s sl) thus ?case apply safe
    by (cases sl) (force intro!: wffp.intros)+
  qed(auto intro: wffp.intros)
qed (auto simp: wffp-NE)

lemma wffp-hdI[intro]:
assumes wffp sl and hd sl ∈ δ s
shows wffp (s # sl)
using assms by (cases sl) auto

lemma wffp-append:
assumes sl: wffp sl and sl1: wffp sl1 and h: hd sl1 ∈ δ (last sl)
shows wffp (sl @ sl1)
using sl h by (induct sl) (auto simp: sl1)

lemma wffp-append-iff:
wffp (sl @ sl1) ⟷
 (wffp sl ∧ sl1 = []) ∨
 (sl = [] ∧ wffp sl1) ∨
 (wffp sl ∧ wffp sl1 ∧ hd sl1 ∈ δ (last sl))
(is - ⟷ ?R)
proof
  assume wffp (sl @ sl1)
  thus ?R proof(induction sl @ sl1 arbitrary: sl sl1 rule: list.induct)
    case (Cons s sl sl1 sl2) note C = Cons
    show ?case proof(cases sl1 = [] ∨ sl2 = [])
      case False then obtain sll1 where sl1: sl1 = s # sll1 and sl : sl = sll1 @  sl2
      using C(2) by(cases sl1) auto
      have wsl: wffp sl by (metis C False append-is-Nil-conv list.inject sl wffp.simps)
      show ?thesis using C(1)[OF sl, unfolded sl[symmetric], OF wsl]
      by (metis (no-types) C False wffp-hdI append-is-Nil-conv list.sel(1) hd-append
            last.simps list.inject sl sl1 wffp.simps)
    qed(insert C, auto)
  qed auto
qed (auto simp: wffp-append)

lemma wffp-to-wfp:
assumes π-def: π = map (λ s. (s, L s)) sl @− sconst (toSink (last sl), L (toSink (last sl)))
assumes sl: wffp sl
shows
wfp UNIV π ∧
 (∀ i < length sl. sl ! i = fst (π !! i)) ∧
 (∀ i ≥ length sl. fst (π !! i) = toSink (last sl)) ∧
 stateOf π = hd sl
unfolding wfp proof safe
```

fix *i s*
{assume *s* ∈ *snd* (*π* !! *i*) **thus** *s* ∈ *L* (*fst* (*π* !! *i*))
unfolding *π-def wffp* **by** (*cases i* < *length sl*) *auto*
}
{assume *s* ∈ *L* (*fst* (*π* !! *i*)) **thus** *s* ∈ *snd* (*π* !! *i*)
unfolding *π-def wffp* **by** (*cases i* < *length sl*) *auto*
}
{fix *j* **assume** *j* < *length sl* **thus** *sl*!*j* = *fst* (*π* !! *j*)
unfolding *π-def* **apply** (*cases sl*, *simp*) **by** (*cases j*) *auto*
} note *1* = *this*
{fix *j* **assume** *j* ≥ *length sl* **thus** *fst* (*π* !! *j*) = *toSink* (*last sl*)
using *sl* **unfolding** *π-def* **by** *auto*
} note *2* = *this*
show *fst* (*π* !! *Suc i*) ∈ *δ* (*fst* (*π* !! *i*))
proof(*cases length sl* ≤ *Suc i*)
case *False* **hence** *Suc i* < *length sl* **by** *simp*
hence *fst* (*π* !! *Suc i*) = *sl*!(*Suc i*) ∧ *fst* (*π* !! *i*) = *sl*!*i*
using *1* **by** *fastforce*
thus *?thesis* **using** *sl False* **unfolding** *wffp* **by** *auto*
next
case *True* **note** *sl* = *True*
hence *22*: *fst* (*π* !! *Suc i*) = *toSink* (*last sl*) **using** *2* **by** *blast*
show *?thesis*
proof(*cases length sl* ≤ *i*)
case *True*
hence *fst* (*π* !! *i*) = *toSink* (*last sl*) **using** *2* **by** *auto*
thus *?thesis* **using** *22* **by** (*metis IntD2 Sink-idle UNIV-I toSink*)
next
case *False*
hence *last sl* = *sl*!*i* **using** *sl*
by (*metis Suc-eq-plus1 diff-add-inverse2 last-conv-nth le0 le-Suc-eq length-0-conv*)
moreover have *fst* (*π* !! *i*) = *sl*!*i* **using** *False 1* **by** *auto*
ultimately show *?thesis* **using** *22* **by** (*metis IntD1 UNIV-I toSink*)
qed
qed
show *stateOf π* = *hd sl* **using** *wffp-NE*[*OF sl*] **unfolding** *π-def* **by** (*cases sl*) *auto*
qed *auto*

lemma *wffp-imp-appendL*: *wffp* (*sl1* @ *sl2*) ⟹ *sl1* ≠ [] ⟹ *wffp sl1*
by (*metis wffp-append-iff*)

lemma *wffp-imp-appendR*: *wffp* (*sl1* @ *sl2*) ⟹ *sl2* ≠ [] ⟹ *wffp sl2*
by (*metis wffp-append-iff*)

lemma *wffp-iff-map-Idle*:
assumes *wffp sl*
shows
∃ *n st*.
(*n* > *0* ∧ *sl* = *map Idle* (*replicate n st*)) ∨

```
  (∃ st1 u1 c1 sl1. sl = map Idle (replicate n st) @ [State st1 u1 c1] @ sl1)
using assms proof (induction rule: wffp-induct2)
 case (Singl s) show ?case proof (cases s)
  case (Idle st)
  show ?thesis unfolding Idle by (intro exI[of - Suc 0] exI[of - st]) auto
 next
  case (State st1 u1 c1)
  show ?thesis unfolding State by (intro exI[of - 0] exI[of - st]) auto
 qed
next
 case (Cons s sl)
 {fix n st
  assume n: n > 0 and sl: sl = map Idle (replicate n st)
  then obtain n' where n: n = Suc n' by (cases n) auto
  hence sl': sl = (Idle st) # map Idle (replicate n' st) using sl by auto
  have ?case proof(cases s)
   case (Idle st1)
   have st1: st1 = st using ‹hd sl ∈ δ s› unfolding sl' Idle by auto
   show ?thesis apply (intro exI[of - Suc n] exI[of - st]) using n unfolding sl Idle st1 by auto
  next
   case (State st1 u1 c1)
   hence s # sl = map Idle (replicate 0 st) @ [State st1 u1 c1] @ sl by simp
   thus ?thesis by blast
  qed
 }
 moreover
 {fix n st st1 u1 c1 sl1 assume sl: sl = map Idle (replicate n st) @ [State st1 u1 c1] @ sl1
  have ?case proof(cases s)
   case (Idle st2)
   show ?thesis
   proof(cases n)
    case 0
    have s # sl = map Idle (replicate (Suc 0) st2) @ [State st1 u1 c1] @ sl1
    unfolding sl Idle 0 by simp
    thus ?thesis by blast
   next
    case (Suc n')
    hence sl': sl = (Idle st) # map Idle (replicate n' st) @ [State st1 u1 c1] @ sl1 using sl by auto
    have st2: st2 = st using ‹hd sl ∈ δ s› unfolding sl' Idle by auto
    have s # sl = map Idle (replicate (Suc n) st) @ [State st1 u1 c1] @ sl1
    unfolding sl Idle st2 by auto
    thus ?thesis by blast
   qed
  next
   case (State st1 u1 c1)
   hence s # sl = map Idle (replicate 0 st) @ [State st1 u1 c1] @ sl by simp
   thus ?thesis by blast
  qed
 }
```

```
  ultimately show ?case using Cons(3) by auto
qed

lemma wffp-cases3[elim, consumes 1, case-names Idle State Idle-State]:
assumes wffp sl
obtains
n st where
n > 0 and sl = map Idle (replicate n st)
|
st u c sl1 where
sl = State st u c # sl1 and sl1 ≠ [] ⟹ wffp sl1 ∧ hd sl1 ∈ δ (State st u c)
|
n st u c sl1 where
n > 0 and sl = map Idle (replicate n st) @ [State (do st u c) u c] @ sl1
and sl1 ≠ [] ⟹ wffp sl1 ∧ hd sl1 ∈ δ (State (do st u c) u c)
proof−
  {fix n st
   assume n: n > 0 and sl: sl = map Idle (replicate n st)
   hence thesis using that by auto
  }
  moreover
  {fix n st st1 u1 c1 sl1 assume sl: sl = map Idle (replicate n st) @ [State st1 u1 c1] @ sl1
   have 1: sl1 ≠ [] ⟹ wffp sl1 ∧ hd sl1 ∈ δ (State st1 u1 c1)
   by (metis append-is-Nil-conv assms last.simps not-Cons-self2 sl wffp-append-iff)
   have thesis
   proof(cases n)
     case 0
     have sl = State st1 u1 c1 # sl1 using sl unfolding 0 by auto
     thus thesis using that 1 by blast
   next
     case (Suc n')
     hence 2: replicate n st = replicate n' st @ [st] by (metis replicate-Suc replicate-append-same)
     have wffp (map Idle [st] @ [State st1 u1 c1])
     using assms unfolding sl 2 unfolding map-append append-assoc
      by (metis (no-types) append-assoc append-is-Nil-conv append-self-conv
                append-singl-rev neq-Nil-conv wffp-imp-appendL wffp-imp-appendR)
     hence st1: st1 = do st u1 c1 by (auto elim!: wffp.cases)
     have n > 0 using Suc by auto
     thus ?thesis using that 1 by (metis sl st1)
   qed
  }
  ultimately show thesis
  using wffp-iff-map-Idle[OF assms] by auto
qed

lemma wffp-cases2[elim, consumes 1, case-names Idle State]:
assumes wffp sl
obtains
n st where
```

n > *0* **and** *sl* = *map Idle* (*replicate n st*)
|
n st st1 u c sl1 **where**
sl = *map Idle* (*replicate n st*) @ [*State st1 u c*] @ *sl1*
and *sl1* ≠ [] ⟹ *wffp sl1* ∧ *hd sl1* ∈ δ (*State st1 u c*)
using *assms* **apply**(*cases sl rule*: *wffp-cases3*)
apply *metis*
apply (*metis append-Cons append-Nil list.map*(*1*) *replicate-0*)
apply (*metis append-Cons append-Nil*)
done

lemma *wffp-Idle-Idle*:
assumes *wffp* (*sl1* @ [*Idle st1*] @ [*Idle st2*] @ *sl2*)
shows *st2* = *st1*
proof−
have *wffp* [*Idle st1*, *Idle st2*] **using** *assms*
by (*metis wffp-imp-appendR append-assoc append-singl-rev list.distinct*(*1*) *wffp-imp-appendL*)
thus *?thesis* **unfolding** *wffp* **by** *auto*
qed

lemma *wffp-Idle-State*:
assumes *wffp* (*sl1* @ [*Idle st1*] @ [*State st2 u2 c2*] @ *sl2*)
shows *st2* = *st1* ∨ *st2* = *do st1 u2 c2*
proof−
have *wffp* [*Idle st1*, *State st2 u2 c2*] **using** *assms*
by (*metis wffp-imp-appendR append-assoc append-singl-rev list.distinct*(*1*) *wffp-imp-appendL*)
thus *?thesis* **unfolding** *wffp* **by** *auto*
qed

lemma *wffp-State-Idle*:
assumes *wffp* (*sl1* @ [*State st1 u1 c1*] @ [*Idle st2*] @ *sl2*)
shows *st2* = *st1*
proof−
have *wffp* [*State st1 u1 c1*, *Idle st2*] **using** *assms*
by (*metis wffp-imp-appendR append-assoc append-singl-rev list.distinct*(*1*) *wffp-imp-appendL*)
thus *?thesis* **unfolding** *wffp* **by** *auto*
qed

lemma *wffp-State-State*:
assumes *wffp* (*sl1* @ [*State st1 u1 c1*] @ [*State st2 u2 c2*] @ *sl2*)
shows *st2* = *do st1 u2 c2*
proof−
have *wffp* [*State st1 u1 c1*, *State st2 u2 c2*] **using** *assms*
by (*metis wffp-imp-appendR append-assoc append-singl-rev list.distinct*(*1*) *wffp-imp-appendL*)
thus *?thesis* **unfolding** *wffp* **by** *auto*
qed

lemma *wfp-to-wffp*:
assumes *sl-def*: *sl* = *map fst* (*stake i π*) **and** *i*: *i* > *0* **and** π: *wfp UNIV π*

shows
wffp sl ∧
(∀ *j* < *length sl*. *fst* (π !! *j*) = *sl* ! *j*) ∧
stateOf π = *hd sl*
unfolding *wffp* **proof**(*intro conjI allI impI*)
fix *j*
have *1*: *stake i* π ≠ [] **using** *i* **by** *auto*
show *stateOf* π = *hd sl* **unfolding** *sl-def hd-map*[*OF 1*] **using** *i* **by** *simp*
qed(*insert assms*, *unfold sl-def wfp*, *auto*)

lemma *nonintSI-nonintS*: *nonintSI* ⟷ *nonintS*
proof(*unfold nonintS-def nonintSI-def*, *safe*)
fix *sl sl′ i*
obtain π π′ **where**
π: π = *map* (λ *s*. (*s*, *L s*)) *sl* @− *sconst* (*toSink* (*last sl*), *L* (*toSink* (*last sl*))) **and**
π′: π′ = *map* (λ *s*. (*s*, *L s*)) *sl′* @− *sconst* (*toSink* (*last sl′*), *L* (*toSink* (*last sl′*)))
by *blast*
assume *0*: ∀ π π′.
wfp UNIV π ∧ *wfp UNIV* π′ ∧ *stateOf* π = *s0* ∧ *stateOf* π′ = *s0*
⟶
(∀ *i*. (∀ *j* ≤ *i*. *f* (*fst* (π !! *j*)) (*fst* (π′ !! *j*))) ⟶ *g* (*fst* (π !! *i*)) (*fst* (π′ !! *i*)))
and *slsl′*: *wffp sl wffp sl′ hd sl* = *s0 hd sl′* = *s0*
and *list-all2 f sl sl′*
hence *l*: *length sl* = *length sl′* **and** *i*: ∀ *i* < *length sl*. *f* (*sl* ! *i*) (*sl′* ! *i*)
unfolding *list-all2-conv-all-nth* **by** *auto*
define *i0* **where** *i0* = *length sl* − *1*
have *slsl′-NE*: *sl* ≠ [] ∧ *sl′* ≠ [] **using** *slsl′ wffp-NE* **by** *auto*
hence *last*: *last sl* = *sl*!*i0 last sl′* = *sl′*!*i0*
by (*metis i0-def l slsl′ last-conv-nth*)+
have *i0*: *i0* < *length sl i0* < *length sl′* **unfolding** *i0-def* **using** *l slsl′ slsl′-NE* **by** *auto*
have *j*: ∀ *j* ≤ *i0*. *f* (*sl* ! *j*) (*sl′* ! *j*) **using** *i slsl′-NE* **unfolding** *i0-def*
by (*metis Suc-diff-eq-diff-pred Suc-diff-le Zero-neq-Suc diff-is-0-eq′*
le-less-linear length-greater-0-conv)
show *g* (*last sl*) (*last sl′*)
unfolding *last* **using** *0 slsl′ j i0*
using *wffp-to-wfp*[*OF* π] *wffp-to-wfp*[*OF* π′] **by** *auto*
next
fix π π′ *i* **assume**
∀ *sl sl′*. *wffp sl* ∧ *wffp sl′* ∧ *hd sl* = *s0* ∧ *hd sl′* = *s0* ∧ *list-all2 f sl sl′* ⟶ *g* (*last sl*) (*last sl′*)
and ππ′: *wfp UNIV* π *wfp UNIV* π′ **and** *state*: *stateOf* π = *s0 stateOf* π′ = *s0*
and *f*: ∀*j*≤*i*. *f* (*fst* (π !! *j*)) (*fst* (π′ !! *j*))
hence *R*:
∀ *sl sl′*. *wffp sl* ∧ *wffp sl′* ∧ *hd sl* = *s0* ∧ *hd sl′* = *s0* ∧ *length sl* = *length sl′*
⟶
((∀ *i* < *length sl*. *f* (*sl*!*i*) (*sl′*!*i*)) ⟶ *g* (*last sl*) (*last sl′*))
unfolding *list-all2-conv-all-nth* **by** *auto*
define *i0* **where** *i0* = *Suc i*
have *i0-ge*: *i0* > *0* **unfolding** *i0-def* **by** *auto*
have *ii0*: *i* < *i0* **unfolding** *i0-def* **by** *auto*

have *f*: $\forall j{<}i0.\ f\ (fst\ (\pi\ !!\ j))\ (fst\ (\pi'\ !!\ j))$ **using** *f* **unfolding** *i0-def* **by** *auto*
obtain *sl sl'* **where**
sl-def: $sl = map\ fst\ (stake\ i0\ \pi)$ **and** *sl'-def*: $sl' = map\ fst\ (stake\ i0\ \pi')$
by *blast*
have *i0*: $i0 = length\ sl\ length\ sl' = length\ sl$ **unfolding** *i0-def sl-def sl'-def* **by** *auto*
have *1*: $sl!i = last\ sl\ sl'!i = last\ sl'$
using *i0* **unfolding** *i0-def* **using** *last-conv-nth length-greater-0-conv* **by** (*metis diff-Suc-1 i0 i0-ge*)+
show $g\ (fst\ (\pi\ !!\ i))\ (fst\ (\pi'\ !!\ i))$
using *wfp-to-wffp*[*OF sl-def i0-ge* $\pi\pi'(1)$] *wfp-to-wffp*[*OF sl'-def i0-ge* $\pi\pi'(2)$]
using *R state f ii0* **by** (*simp add*: *1 i0*)
qed

Finally, we show that nonintS is equivalent to standard noninterference (predicate nonint).

purgeIdle removes the idle steps from a finite path:

definition *purgeIdle* :: $('St,\ 'U,\ 'C)\ state\ list \Rightarrow ('St,\ 'U,\ 'C)\ state\ list$
where *purgeIdle* $\equiv$ *filter isState*

lemma *purgeIdle-simps*[*simp*]:
$purgeIdle\ [] = []$
$purgeIdle\ ((Idle\ st)\ \#\ sl) = purgeIdle\ sl$
$purgeIdle\ ((State\ st\ u\ c)\ \#\ sl) = (State\ st\ u\ c)\ \#\ purgeIdle\ sl$
unfolding *purgeIdle-def* **by** *auto*

lemma *purgeIdle-append*:
$purgeIdle\ (sl1\ @\ sl2) = purgeIdle\ sl1\ @\ purgeIdle\ sl2$
unfolding *purgeIdle-def* **by** (*metis filter-append*)

lemma *purgeIdle-set-isState*:
assumes $s \in set\ (purgeIdle\ sl)$
shows *isState s*
using *assms* **unfolding** *purgeIdle-def* **by** (*metis filter-set member-filter*)

lemma *purgeIdle-Nil-iff*:
$purgeIdle\ sl = [] \longleftrightarrow (\forall s{\in}set\ sl.\ \neg\ isState\ s)$
unfolding *purgeIdle-def filter-empty-conv* **by** *auto*

lemma *purgeIdle-Cons-iff*:
$purgeIdle\ sl = s\ \#\ sll$
$\longleftrightarrow$
$(\exists\ sl1\ sl2.\ sl = sl1\ @\ s\ \#\ sl2\ \wedge$
$\quad (\forall s1{\in}set\ sl1.\ \neg\ isState\ s1)\ \wedge\ isState\ s\ \wedge\ purgeIdle\ sl2 = sll)$
unfolding *purgeIdle-def filter-eq-Cons-iff* **by** *auto*

lemma *purgeIdle-map-Idle*[*simp*]:
$purgeIdle\ (map\ Idle\ s) = []$
unfolding *purgeIdle-def* **by** *auto*

lemma *purgeIdle-replicate-Idle*[*simp*]:
$purgeIdle\ (replicate\ n\ (Idle\ st)) = []$

unfolding *purgeIdle-def* **by** *auto*

lemma *wffp-purgeIdle-Nil*:
assumes *wffp sl* **and** *purgeIdle sl* = []
shows ∃ *n st*. *n* > *0* ∧ *sl* = *replicate n* (*Idle st*)
using *assms* **proof**(*induction sl rule*: *wffp-induct2*)
  case (*Singl s*) **thus** *?case*
  by (*cases s*) (*auto intro*: *exI*[*of* - *Suc 0*])
next
  case (*Cons s sl*)
  then obtain *n st* **where** *sl*: *sl* = *replicate n* (*Idle st*) **by** (*cases s*) *auto*
  obtain *st1* **where** *s*: *s* = *Idle st1* **using** *Cons* **by** (*cases s*) *auto*
  have *1*: *hd* (*replicate n* (*Idle st*)) = *Idle st* **by** (*metis Cons.hyps*(*2*) *hd-replicate replicate-empty sl wffp*)
  show *?case* **using** *Cons*(*1*) **by** (*auto intro*: *exI*[*of* - *Suc n*] *exI*[*of* - *st*] *simp*: *sl 1 s*)
qed

lemma *wffp-hd-purgeIdle*:
assumes *wsl*: *wffp sl* **and** *psl*: *purgeIdle sl* ≠ []
and *ist*: *isState s* **and** *hsl*: *hd sl* ∈ δ *s*
shows *hd* (*purgeIdle sl*) ∈ δ *s*
using *wsl* **proof**(*cases rule*: *wffp-cases3*)
  case (*Idle n st*)
  show *?thesis* **using** *psl* **unfolding** *Idle* **by** *simp*
next
  case (*State st u c sl1*)
  show *?thesis* **using** *psl hsl* **unfolding** *State* **by** *simp*
next
  case (*Idle-State n st u c sl1*)
  show *?thesis* **using** *psl* ‹*n* > *0*› *ist hsl* **unfolding** *Idle-State purgeIdle-append*
  by (*cases s*) *auto*
qed

lemma *wffp-purgeIdle*:
assumes *wffp sl* **and** *purgeIdle sl* ≠ []
shows *wffp* (*purgeIdle sl*)
using *assms* **proof**(*induction sl rule*: *length-induct*)
  case (*1 sl*) **note** *IH* = *1*
  from ‹*wffp sl*› **show** *?case* **proof**(*cases sl rule*: *wffp-cases2*)
    case (*Idle n st*)
    have *purgeIdle sl* = [] **unfolding** *Idle* **by** *auto*
    thus *?thesis* **using** ‹*purgeIdle sl* ≠ []› **by** *auto*
  next
    case (*State n st st1 u c sl1*)
    hence *1*: *purgeIdle sl* = *State st1 u c* # *purgeIdle sl1*
    by (*auto simp del*: *map-replicate simp add*: *purgeIdle-append*)
    show *?thesis*
    proof(*cases purgeIdle sl1* = []*)*
      case *True* **note** *psl1* = *True*
      show *?thesis* **unfolding** *1 psl1* **by** *auto*

```
    next
      case False hence sl1NE: sl1 ≠ [] by (cases sl1) auto
      hence sl1: wffp sl1 and hsl1: hd sl1 ∈ δ (State st1 u c) by (metis State(2))+
      have length sl1 < length sl using State by auto
      hence sl1: wffp (purgeIdle sl1) using IH(1) sl1 False by auto
      moreover have hd (purgeIdle sl1) ∈ δ (State st1 u c)
      by (metis False GM-sec-model.wffp-hd-purgeIdle State(2) sl1NE state.discI(2))
      ultimately show ?thesis unfolding 1 by auto
    qed
  qed
qed

lemma isState-purgeIdle:
(∃ sl. purgeIdle sl = sll) ⟷ list-all isState sll
unfolding purgeIdle-def
by (metis Ball-set-list-all purgeIdle-def purgeIdle-set-isState filter-True)

lemma wffp-last-purgeIdle:
assumes wffp sl and purgeIdle sl ≠ []
shows getGMState (last (purgeIdle sl)) = getGMState (last sl)
using assms proof(induction sl rule: wffp-induct2)
  case (Singl s) thus ?case by (cases s) auto
next
  case (Cons s sl)
  hence slNE: sl ≠ [] by (metis wffp-NE)
  show ?case
  proof(cases purgeIdle sl = [])
    case True then obtain n st where sl: sl = replicate n (Idle st) by (metis Cons.hyps wffp-purgeIdle-Nil)
    hence n: n > 0 using slNE by auto
    hence hsl: hd sl = Idle st and lsl: last sl = Idle st unfolding sl by auto
    have s: isState s using True Cons by (cases s) auto
    have 1: getGMState s = st using ‹hd sl ∈ δ s› unfolding hsl by(cases s) auto
    show ?thesis using slNE n 1 hsl lsl s unfolding sl purgeIdle-replicate-Idle by (cases s) auto
  next
    case False
    thus ?thesis using Cons by (cases s) auto
  qed
qed

lemma wffp-isState-doo:
assumes wffp sl and list-all isState sl
shows doo (getGMState (hd sl)) (map getGMUserCom (tl sl)) = getGMState (last sl)
using assms proof(induction sl rule: wffp-induct2)
  case (Cons s sl)
  then obtain st u c where s: s = State st u c by(cases s ) auto
  have sl: sl ≠ [] and sl1: sl = hd sl # tl sl using wffp-NE[OF ‹wffp sl›] by auto
  with Cons obtain st1 u1 c1 where hsl: hd sl = State st1 u1 c1
  by (metis isState-purgeIdle isState-def purgeIdle-Cons-iff)
  have 1: getGMState (hd sl) = do st u1 c1 using ‹hd sl ∈ δ s› unfolding hsl s by simp
```

have *doo st* (*map getGMUserCom sl*) = *doo* (*do st u1 c1*) (*map getGMUserCom* (*tl sl*))
by (*subst sl1*) (*simp add*: *1 hsl*)
thus *?case* **using** *sl Cons* **unfolding** *1 s* **by** *auto*
qed *auto*

lemma *isState-hd-purgeIdle*:
assumes *wsl*: *wffp sl* **and** *ist*: *isState* (*hd sl*)
shows *purgeIdle sl* ≠ [] ∧ *hd* (*purgeIdle sl*) = *hd sl*
using *ist*
by (*intro conjI*) (*subst hd-Cons-tl*[*OF wffp-NE*[*OF wsl*], *symmetric*], *cases hd sl*, *cases sl*, *auto*)+

lemma *wffp-isState-doo-purgeIdle*:
fixes *sl* **defines** *sll*: *sll* ≡ *purgeIdle sl*
assumes *wsl*: *wffp sl* **and** *ist*: *isState* (*hd sl*)
shows *doo* (*getGMState* (*hd sl*)) (*map getGMUserCom* (*tl sll*)) = *getGMState* (*last sl*)
proof−
note *1* = *isState-hd-purgeIdle*[*OF wsl ist*]
hence *wsll*: *wffp sll* **by** (*metis sll wffp-purgeIdle wsl*)
hence *doo* (*getGMState* (*hd sll*)) (*map getGMUserCom* (*tl sll*)) = *getGMState* (*last sll*)
by (*metis wffp-isState-doo isState-purgeIdle sll*)
thus *?thesis* **by** (*metis 1 sll wffp-last-purgeIdle wsl*)
qed

lemma *map-getGMUserCom-surj*:
assumes *isState s*
shows ∃ *sl*. *wffp sl* ∧ *list-all isState sl* ∧ *hd sl* = *s* ∧ *map getGMUserCom* (*tl sl*) = *ucl*
using *assms* **proof**(*induction ucl arbitrary*: *s rule*: *list-pair-induct*)
case *Nil* **thus** *?case* **apply**(*intro exI*[*of* - [*s*]]) **by** *auto*
next
case (*Cons u c ucl s*)
then obtain *st1 u1 c1* **where** *s*: *s* = *State st1 u1 c1* **by** (*cases s*) *auto*
define *s1* **where** *s1* = *State* (*do st1 u c*) *u c*
obtain *sl* **where** *sl*: *wffp sl* ∧ *list-all isState sl* **and** *hsl*: *hd sl* = *s1*
and *msl*: *map getGMUserCom* (*tl sl*) = *ucl* **using** *Cons*(*1*)[*of s1*] **unfolding** *s1-def* **by** *auto*
thus *?case* **using** *s s1-def* **by** (*intro exI*[*of* - *s* # *sl*]) *auto*
qed

lemma *purgeIdle-purge-ex*:
assumes *wffp sl* **and** *list-all isState sl* **and** *map getGMUserCom* (*tl sl*) = *ucl*
shows ∃ *sl*′. *hd sl*′ = *ss*′ ∧ *wffp sl*′ ∧
list-all2 f (*tl sl*) (*tl sl*′) ∧
map getGMUserCom (*purgeIdle* (*tl sl*′)) = *purge GH ucl*
using *assms* **proof**(*induction sl arbitrary*: *ucl ss*′ *rule*: *wffp-induct2*)
case (*Singl s ucl*)
thus *?case* **apply** (*intro exI*[*of* - [*ss*′]]) **by** (*cases ss*′) *auto*
next
case (*Cons ss sl ucl ss*′) **note** *wsl* = ‹*wffp sl*›
hence *slNE*: *sl* ≠ [] **by** (*metis wffp-NE*)
obtain *s sl1* **where** *sl*: *sl* = *s* # *sl1* **using** *wffp-NE*[*OF* ‹*wffp sl*›] **by** (*cases sl*) *auto*

```
  then obtain st u c where s: s = State st u c using Cons by (cases s) auto
  define ucl1 where ucl1 = tl ucl
  have ucl: ucl = (u,c) # ucl1 and hsl: hd sl = s using Cons(5) unfolding s ucl1-def sl by auto
  have 1: list-all isState sl and 2: map getGMUserCom (tl sl) = ucl1
  using Cons unfolding ucl1-def s by auto
  define st' where st' = getGMState ss'
  show ?case
  proof(cases u ∈ GH)
    case True note u = True
    define s' :: ('St, 'U, 'C) state where s' = Idle st'
    obtain sl' where hsl': hd sl' = s' and wsl': wffp sl'
    and slsl': list-all2 f (tl sl) (tl sl') and m: map getGMUserCom (purgeIdle (tl sl')) = purge GH ucl1
    using Cons(3)[OF 1 2, of s'] by auto
    hence sl'NE: sl' ≠ [] by (metis wffp-NE)
    have wffp (ss' # sl') using wsl' hsl' unfolding s'-def st'-def by (cases ss') auto
    moreover
    {have f s s' using u unfolding s'-def st'-def s f-def by blast
     hence list-all2 f sl sl' using slsl' hsl hsl' slNE sl'NE
     by (metis list.sel(1,3) list-all2-Cons neq-Nil-conv)
    }
    moreover have map getGMUserCom (purgeIdle sl') = purge GH ucl
    by (subst hd-Cons-tl[OF sl'NE, symmetric]) (auto simp: hsl' ucl s'-def u m)
    ultimately show ?thesis by (intro exI[of - ss' # sl']) auto
  next
    case False note u = False
    define s' where s' = State (do st' u c) u c
    obtain sl' where hsl': hd sl' = s' and wsl': wffp sl'
    and slsl': list-all2 f (tl sl) (tl sl') and m: map getGMUserCom (purgeIdle (tl sl')) = purge GH ucl1
    using Cons(3)[OF 1 2, of s'] by auto
    hence sl'NE: sl' ≠ [] by (metis wffp-NE)
    have wffp (ss' # sl') using wsl' hsl' unfolding s'-def st'-def by (cases ss') auto
    moreover
    {have f s s' unfolding s'-def st'-def s f-def by simp
     hence list-all2 f sl sl' using slsl' hsl hsl' slNE sl'NE
     by (metis list.sel(1,3) list-all2-Cons neq-Nil-conv)
    }
    moreover have map getGMUserCom (purgeIdle sl') = purge GH ucl
    by (subst hd-Cons-tl[OF sl'NE, symmetric]) (auto simp: hsl' ucl s'-def u m)
    ultimately show ?thesis by (intro exI[of - ss' # sl']) auto
  qed
qed

lemma purgeIdle-getGMUserCom-purge:
fixes sl sl'
defines ucl ≡ map getGMUserCom (purgeIdle (tl sl))
   and ucl' ≡ map getGMUserCom (purgeIdle (tl sl'))
assumes wsl: wffp sl and wsl': wffp sl' and f: list-all2 f sl sl'
shows purge GH ucl = purge GH ucl'
proof–
```

```
  have length sl = length sl' using f by (metis list-all2-lengthD)
  thus ?thesis using assms proof(induction arbitrary: ucl ucl' rule: list-induct2)
    case Nil
    thus ?case by auto
  next
    case (Cons s sl s' sl')
    show ?case
    proof(cases sl = [])
      case True hence sl' = [] using Cons by auto
      thus ?thesis using True by auto
    next
      case False hence sl: sl = hd sl # tl sl by (cases sl) auto
      hence sl': sl' = hd sl' # tl sl' using ‹length sl = length sl'› by (cases sl') auto
      hence wsl[simp]: wffp sl and wsl'[simp]: wffp sl' using sl Cons
      by (metis Cons.prems append-singl-rev list.distinct sl' wffp-imp-appendR)+
      have f: f (hd sl) (hd sl') using ‹list-all2 f (s # sl) (s' # sl')› sl sl'
      by (metis list-all2-Cons)
      show ?thesis proof(cases hd sl)
        case (Idle st) note hsl = Idle
        show ?thesis proof(cases hd sl')
          case (Idle st') note hsl' = Idle
          show ?thesis apply(subst sl, subst sl') using Cons unfolding hsl hsl' by auto
        next
          case (State st' u' c') note hsl' = State
          have u': u' ∈ GH using f unfolding hsl hsl' by (auto simp: f-def)
          show ?thesis apply(subst sl, subst sl') using Cons u' unfolding hsl hsl' by auto
        qed
      next
        case (State st u c) note hsl = State
        show ?thesis proof(cases hd sl')
          case (Idle st') note hsl' = Idle
          have u: u ∈ GH using f unfolding hsl hsl' by (auto simp: f-def)
          show ?thesis apply(subst sl, subst sl') using Cons u unfolding hsl hsl' by auto
        next
          case (State st' u' c') note hsl' = State
          have uu': (u' ∈ GH ⟷ u ∈ GH) ∧ (u ∉ GH ⟶ u' = u ∧ c' = c)
          using f unfolding hsl hsl' by (auto simp: f-def)
          show ?thesis
          apply(subst sl, subst sl') using Cons uu' unfolding hsl hsl' by (cases u ∈ GH) auto
        qed
      qed
    qed
  qed
qed

lemma nonintS-iff-nonint:
nonintS ⟷ nonint
unfolding nonintS-def nonint-def proof safe
  fix ucl u
```

assume
1: $\forall\, sl\ sl'.\ wffp\ sl \wedge wffp\ sl' \wedge hd\ sl = s0 \wedge hd\ sl' = s0 \wedge list\text{-}all2\ f\ sl\ sl' \longrightarrow$
$g\ (last\ sl)\ (last\ sl')$
and *u*: $u \in GL$
obtain *sl* **where** *wsl*: *wffp sl* **and** *l*: *list-all isState sl* **and** *hsl*: $hd\ sl = s0$
and *m*: *map getGMUserCom* $(tl\ sl) = ucl$ **using** *map-getGMUserCom-surj*[*of s0*] **by** *auto*
then obtain sl' **where** *hsl'*: $hd\ sl' = hd\ sl$ **and** *wsl'*: *wffp* sl' **and** *f*: *list-all2 f* $(tl\ sl)\ (tl\ sl')$
and m': *map getGMUserCom* $(purgeIdle\ (tl\ sl')) = purge\ GH\ ucl$
by (*metis purgeIdle-purge-ex*)
have *slNE*: $sl \neq []$ **and** $sl'NE$: $sl' \neq []$ **using** *wsl wsl'* **by** (*metis wffp-NE*)+
have *2*: *getGMState* $(hd\ sl) = st0$ **unfolding** *hsl* **by** *auto*
have *3*: $tl\ (purgeIdle\ sl') = purgeIdle\ (tl\ sl')$
apply(*subst hd-Cons-tl*[*OF sl'NE*, *symmetric*], *rule sym*, *subst hd-Cons-tl*[*OF sl'NE*, *symmetric*])
unfolding *hsl hsl'* **by** *auto*
have *f*: *list-all2 f sl sl'*
apply (*subst hd-Cons-tl*[*OF slNE*, *symmetric*], *subst hd-Cons-tl*[*OF sl'NE*, *symmetric*])
using *f hsl'* **unfolding** *f-def* **by** *auto*
hence *g*: *g* $(last\ sl)\ (last\ sl')$ **using** *1 wsl wsl' hsl hsl'* **by** *auto*
moreover have *getGMState* $(last\ sl) = doo\ st0\ ucl$
unfolding *m*[*symmetric*] *2*[*symmetric*] **using** *wffp-isState-doo*[*OF wsl l*] **by** *simp*
moreover have *getGMState* $(last\ sl') = doo\ st0\ (purge\ GH\ ucl)$
using *wffp-isState-doo-purgeIdle*[*OF wsl'*] **unfolding** *hsl' hsl m' 3* **by** *auto*
ultimately show *out* $(doo\ st0\ ucl)\ u = out\ (doo\ st0\ (purge\ GH\ ucl))\ u$
unfolding *g-def* **using** *u* **by** *auto*
next
fix *sl sl'*
assume *1*: $\forall\, ucl.\ \forall\, u{\in}GL.\ out\ (doo\ st0\ ucl)\ u = out\ (doo\ st0\ (purge\ GH\ ucl))\ u$
and *wsl*: *wffp sl* **and** *wsl'*: *wffp* sl' **and** *hsl*: $hd\ sl = s0$ **and** *hsl'*: $hd\ sl' = s0$
and *f*: *list-all2 f sl sl'*
define *ucl* **where** $ucl = map\ getGMUserCom\ (tl\ (purgeIdle\ sl))$
define ucl' **where** $ucl' = map\ getGMUserCom\ (tl\ (purgeIdle\ sl'))$
have *2*: $tl\ (purgeIdle\ sl) = purgeIdle\ (tl\ sl)\ tl\ (purgeIdle\ sl') = purgeIdle\ (tl\ sl')$
by (*subst hd-Cons-tl*[*OF wffp-NE*[*OF wsl*], *symmetric*, *unfolded hsl*], *auto*)[]
(*subst hd-Cons-tl*[*OF wffp-NE*[*OF wsl'*], *symmetric*, *unfolded hsl'*], *auto*)
have $purge\ GH\ ucl = purge\ GH\ ucl'$
unfolding *ucl-def ucl'-def 2* **by** (*metis purgeIdle-getGMUserCom-purge f wsl wsl'*)
moreover have *getGMState* $(last\ sl) = doo\ st0\ ucl \wedge getGMState\ (last\ sl') = doo\ st0\ ucl'$
using *wffp-isState-doo-purgeIdle*[*OF wsl*] *wffp-isState-doo-purgeIdle*[*OF wsl'*]
unfolding *hsl hsl' ucl-def ucl'-def* **by** *auto*
ultimately show *g* $(last\ sl)\ (last\ sl')$ **unfolding** *g-def* **using** *1* **by** *metis*
qed

theorem *nonintSfmla-iff-nonint*:
nonintSfmla $[] \longleftrightarrow$ *nonint*
by (*metis nonintSI-nonintS nonintS-iff-nonint nonintSfmla-nonintSI*)

end-of-context GM-sec-model

5 Deep representation of HyperCTL* – syntax and semantics

5.1 Path variables and environments

datatype *pvar* = *Pvariable* (*natOf* : *nat*)

Deeply embedded (syntactic) formulas

datatype *'aprop dfmla* =
Atom 'aprop pvar |
Fls | *Neg 'aprop dfmla* | *Dis 'aprop dfmla 'aprop dfmla* |
Next 'aprop dfmla | *Until 'aprop dfmla 'aprop dfmla* |
Exi pvar 'aprop dfmla

Derived operators

definition *Tr* ≡ *Neg Fls*
definition *Con* φ ψ ≡ *Neg* (*Dis* (*Neg* φ) (*Neg* ψ))
definition *Imp* φ ψ ≡ *Dis* (*Neg* φ) ψ
definition *Eq* φ ψ ≡ *Con* (*Imp* φ ψ) (*Imp* ψ φ)
definition *Fall p* φ ≡ *Neg* (*Exi p* (*Neg* φ))
definition *Ev* φ ≡ *Until Tr* φ
definition *Alw* φ ≡ *Neg* (*Ev* (*Neg* φ))
definition *Wuntil* φ ψ ≡ *Dis* (*Until* φ ψ) (*Alw* φ)
definition *Fall2 p p'* φ ≡ *Fall p* (*Fall p'* φ)

lemmas *der-Op-defs* =
Tr-def Con-def Imp-def Eq-def Ev-def Alw-def Wuntil-def Fall-def Fall2-def

Well-formed formulas are those that do not have a temporal operator outside the scope of any quantifier – indeed, in HyperCTL* such a situation does not make sense, since the temporal operators refer to previously introduced/quantified paths.

primrec *wff* :: *'aprop dfmla* ⇒ *bool* **where**
wff (*Atom a p*) = *True*
|*wff Fls* = *True*
|*wff* (*Neg* φ) = *wff* φ
|*wff* (*Dis* φ ψ) = (*wff* φ ∧ *wff* ψ)
|*wff* (*Next* φ) = *False*
|*wff* (*Until* φ ψ) = *False*
|*wff* (*Exi p* φ) = *True*

The ability to pick a fresh variable

lemma *finite-fresh-pvar*:
assumes *finite* (*P* :: *pvar set*)
obtains *p* **where** *p* ∉ *P*
proof–
have *finite* (*natOf* ‘ *P*) **by** (*metis assms finite-imageI*)
then obtain *n* **where** *n* ∉ *natOf* ‘ *P* **by** (*metis unbounded-k-infinite*)
hence *Pvariable n* ∉ *P* **by** (*metis imageI pvar.sel*)
thus *?thesis* **using** *that* **by** *auto*
qed

definition *getFresh* :: *pvar set* ⇒ *pvar* **where**
getFresh P ≡ *SOME p*. *p* ∉ *P*

lemma *getFresh*:
assumes *finite P* **shows** *getFresh P* ∉ *P*
by (*metis assms exE-some finite-fresh-pvar getFresh-def*)

The free-variables operator

primrec *FV* :: *'aprop dfmla* ⇒ *pvar set* **where**
FV (*Atom a p*) = {*p*}
|*FV Fls* = {}
|*FV* (*Neg* φ) = *FV* φ
|*FV* (*Dis* φ ψ) = *FV* φ ∪ *FV* ψ
|*FV* (*Next* φ) = *FV* φ
|*FV* (*Until* φ ψ) = *FV* φ ∪ *FV* ψ
|*FV* (*Exi p* φ) = *FV* φ − {*p*}

Environments

type-synonym *env* = *pvar* ⇒ *nat*

definition *eqOn P env env1* ≡ ∀ *p*. *p* ∈ *P* ⟶ *env p* = *env1 p*

lemma *eqOn-Un*[*simp*]:
eqOn (*P* ∪ *Q*) *env env1* ⟷ *eqOn P env env1* ∧ *eqOn Q env env1*
unfolding *eqOn-def* **by** *auto*

lemma *eqOn-update*[*simp*]:
eqOn P (*env*(*p* := π)) (*env1*(*p* := π)) ⟷ *eqOn* (*P* − {*p*}) *env env1*
unfolding *eqOn-def* **by** *auto*

lemma *eqOn-singl*[*simp*]:
eqOn {*p*} *env env1* ⟷ *env p* = *env1 p*
unfolding *eqOn-def* **by** *auto*

context *Shallow*
begin

5.2 The semantic operator

The semantics will interpret deep (syntactic) formulas as shallow formulas. Recall that the latter are predicates on lists of paths – the interpretation will be parameterized by an environment mapping each path variable to a number indicating the index (in the list) for the path denoted by the variable.

The semantics will only be meaningful if the indexes of a formula's free variables are smaller than the length of the path list – we call this property "compatibility".

primrec *sem* :: *'aprop dfmla* ⇒ *env* ⇒ (*'state*,*'aprop*) *sfmla* **where**
sem (*Atom a p*) *env* = *atom a* (*env p*)

$|sem\ Fls\ env = fls$
$|sem\ (Neg\ \varphi)\ env = neg\ (sem\ \varphi\ env)$
$|sem\ (Dis\ \varphi\ \psi)\ env = dis\ (sem\ \varphi\ env)\ (sem\ \psi\ env)$
$|sem\ (Next\ \varphi)\ env = next\ (sem\ \varphi\ env)$
$|sem\ (Until\ \varphi\ \psi)\ env = until\ (sem\ \varphi\ env)\ (sem\ \psi\ env)$
$|sem\ (Exi\ p\ \varphi)\ env = exi\ (\lambda\ \pi\ \pi l.\ sem\ \varphi\ (env(p := length\ \pi l))\ (\pi l\ @\ [\pi]))$

lemma *sem-Exi-explicit*:
$sem\ (Exi\ p\ \varphi)\ env\ \pi l \longleftrightarrow$
$(\exists\ \pi.\ wfp\ AP'\ \pi \wedge stateOf\ \pi = (if\ \pi l \neq []\ then\ stateOf\ (last\ \pi l)\ else\ s0) \wedge$
$sem\ \varphi\ (env(p := length\ \pi l))\ (\pi l\ @\ [\pi]))$
unfolding *sem.simps*
unfolding *exi-def* **..**

lemma *sem-derived*[*simp*]:
$sem\ Tr\ env = tr$
$sem\ (Con\ \varphi\ \psi)\ env = con\ (sem\ \varphi\ env)\ (sem\ \psi\ env)$
$sem\ (Imp\ \varphi\ \psi)\ env = imp\ (sem\ \varphi\ env)\ (sem\ \psi\ env)$
$sem\ (Eq\ \varphi\ \psi)\ env = eq\ (sem\ \varphi\ env)\ (sem\ \psi\ env)$
$sem\ (Fall\ p\ \varphi)\ env = fall\ (\lambda\ \pi\ \pi l.\ sem\ \varphi\ (env(p := length\ \pi l))\ (\pi l\ @\ [\pi]))$
$sem\ (Ev\ \varphi)\ env = ev\ (sem\ \varphi\ env)$
$sem\ (Alw\ \varphi)\ env = alw\ (sem\ \varphi\ env)$
$sem\ (Wuntil\ \varphi\ \psi)\ env = wuntil\ (sem\ \varphi\ env)\ (sem\ \psi\ env)$
unfolding *der-Op-defs der-op-defs* **by** (*auto simp*: *neg-def*[*abs-def*])

lemma *sem-Fall2*[*simp*]:
$sem\ (Fall2\ p\ p'\ \varphi)\ env =$
$fall2\ (\lambda\ \pi'\ \pi\ \pi l.\ sem\ \varphi\ (env\ (p := length\ \pi l,\ p' := Suc(length\ \pi l)))\ (\pi l\ @\ [\pi,\pi']))$
unfolding *Fall2-def fall2-def* **by** (*auto simp add*: *fall-def exi-def*[*abs-def*] *neg-def*[*abs-def*])

Compatibility of a pair (environment,path list) on a set of variables means no out-or-range references:

definition $cpt\ P\ env\ \pi l \equiv \forall\ p \in P.\ env\ p < length\ \pi l$

lemma *cpt-Un*[*simp*]:
$cpt\ (P \cup Q)\ env\ \pi l \longleftrightarrow cpt\ P\ env\ \pi l \wedge cpt\ Q\ env\ \pi l$
unfolding *cpt-def* **by** *auto*

lemma *cpt-singl*[*simp*]:
$cpt\ \{p\}\ env\ \pi l \longleftrightarrow env\ p < length\ \pi l$
unfolding *cpt-def* **by** *auto*

lemma *cpt-map-stl*[*simp*]:
$cpt\ P\ env\ \pi l \Longrightarrow cpt\ P\ env\ (map\ stl\ \pi l)$
unfolding *cpt-def* **by** *auto*

Next we prove that the semantics of well-formed formulas only depends on the interpretation of the free variables of a formula and on the current state of the last recorded path – we call the latter the "pointer" of the path list.

fun *pointerOf* :: $('state,'aprop)\ path\ list \Rightarrow 'state$ **where**

pointerOf πl = (*if* $\pi l \neq []$ *then stateOf* (*last* πl) *else s0*)

Equality of two pairs (environment,path list) on a set of variables:

definition *eqOn* ::
pvar set $\Rightarrow$ *env* $\Rightarrow$ ('*state*,'*aprop*) *path list* $\Rightarrow$ *env* $\Rightarrow$ ('*state*,'*aprop*) *path list* $\Rightarrow$ *bool*
where
eqOn P env πl *env1* $\pi l1 \equiv \forall\ p.\ p \in P \longrightarrow \pi l!(env\ p) = \pi l1!(env1\ p)$

lemma *eqOn-Un*[*simp*]:
eqOn ($P \cup Q$) *env* πl *env1* $\pi l1 \longleftrightarrow$ *eqOn P env* πl *env1* $\pi l1 \wedge$ *eqOn Q env* πl *env1* $\pi l1$
unfolding *eqOn-def* **by** *auto*

lemma *eqOn-singl*[*simp*]:
eqOn $\{p\}$ *env* πl *env1* $\pi l1 \longleftrightarrow \pi l!(env\ p) = \pi l1!(env1\ p)$
unfolding *eqOn-def* **by** *auto*

lemma *eqOn-map-stl*[*simp*]:
cpt P env $\pi l \Longrightarrow$ *cpt P env1* $\pi l1 \Longrightarrow$
eqOn P env πl *env1* $\pi l1 \Longrightarrow$ *eqOn P env* (*map stl* πl) *env1* (*map stl* $\pi l1$)
unfolding *eqOn-def cpt-def* **by** *auto*

lemma *cpt-map-sdrop*[*simp*]:
cpt P env $\pi l \Longrightarrow$ *cpt P env* (*map* (*sdrop i*) πl)
unfolding *cpt-def* **by** *auto*

lemma *cpt-update*[*simp*]:
assumes *cpt* ($P - \{p\}$) *env* πl
shows *cpt P* (*env*(p := *length* πl)) (πl @ [π])
using *assms* **unfolding** *cpt-def* **by** *simp* (*metis Diff-iff less-SucI singleton-iff*)

lemma *eqOn-map-sdrop*[*simp*]:
cpt V env $\pi l \Longrightarrow$ *cpt V env1* $\pi l1 \Longrightarrow$
eqOn V env πl *env1* $\pi l1 \Longrightarrow$ *eqOn V env* (*map* (*sdrop i*) πl) *env1* (*map* (*sdrop i*) $\pi l1$)
unfolding *eqOn-def cpt-def* **by** *auto*

lemma *eqOn-update*[*simp*]:
assumes *cpt* ($P - \{p\}$) *env* πl **and** *cpt* ($P - \{p\}$) *env1* $\pi l1$
shows
eqOn P (*env*(p := *length* πl)) (πl @ [π]) (*env1*(p := *length* $\pi l1$)) ($\pi l1$ @ [π])
$\longleftrightarrow$
eqOn ($P - \{p\}$) *env* πl *env1* $\pi l1$
using *assms* **unfolding** *eqOn-def cpt-def* **by** *simp* (*metis DiffI nth-append singleton-iff*)

lemma *eqOn-FV-sem-NE*:
assumes *cpt* (*FV* φ) *env* πl **and** *cpt* (*FV* φ) *env1* $\pi l1$ **and** *eqOn* (*FV* φ) *env* πl *env1* $\pi l1$
and $\pi l \neq []$ **and** $\pi l1 \neq []$ **and** *last* πl = *last* $\pi l1$
shows *sem* φ *env* πl = *sem* φ *env1* $\pi l1$
using *assms* **proof** (*induction* φ *arbitrary*: *env* πl *env1* $\pi l1$)
case (*Until* φ ψ *env* πl *env1* $\pi l1$)

hence $\bigwedge$ *i*. *sem* φ *env* (*map* (*sdrop i*) πl) = *sem* φ *env1* (*map* (*sdrop i*) $\pi l1$) $\wedge$
sem ψ *env* (*map* (*sdrop i*) πl) = *sem* ψ *env1* (*map* (*sdrop i*) $\pi l1$)
using *Until* **by** (*auto simp*: *last-map*)
thus *?case* **by** (*auto simp*: *op-defs*)
next
case (*Exi p* φ *env* πl *env1* $\pi l1$)
hence *1*:
$\bigwedge$ π. *cpt* (*FV* φ) (*env*(*p* := *length* πl)) (πl @ [π]) $\wedge$
cpt (*FV* φ) (*env1*(*p* := *length* $\pi l1$)) ($\pi l1$ @ [π]) $\wedge$
eqOn (*FV* φ) (*env*(*p* := *length* πl)) (πl @ [π]) (*env1*(*p* := *length* $\pi l1$)) ($\pi l1$ @ [π])
by *simp-all*
thus *?case* **unfolding** *sem.simps exi-def* **using** *Exi*
by (*intro iff-exI*) (*metis append-is-Nil-conv last-snoc*)
qed(*auto simp*: *last-map op-defs*)

The next theorem states that the semantics of a formula on an environment and a list of paths only depends on the pointer of the list of paths.

theorem *eqOn-FV-sem*:
assumes *wff* φ **and** *pointerOf* πl = *pointerOf* $\pi l1$
and *cpt* (*FV* φ) *env* πl **and** *cpt* (*FV* φ) *env1* $\pi l1$ **and** *eqOn* (*FV* φ) *env* πl *env1* $\pi l1$
shows *sem* φ *env* πl = *sem* φ *env1* $\pi l1$
using *assms* **proof** (*induction* φ *arbitrary*: *env* πl *env1* $\pi l1$)
case (*Until* φ ψ *env* πl *env1* $\pi l1$)
hence $\bigwedge$ *i*. *sem* φ *env* (*map* (*sdrop i*) πl) = *sem* φ *env1* (*map* (*sdrop i*) $\pi l1$) $\wedge$
sem ψ *env* (*map* (*sdrop i*) πl) = *sem* ψ *env1* (*map* (*sdrop i*) $\pi l1$)
using *Until* **by** (*auto simp*: *last-map*)
thus *?case* **by** (*auto simp*: *op-defs*)
next
case (*Exi p* φ *env* πl *env1* $\pi l1$)
have $\bigwedge$ π. *sem* φ (*env*(*p* := *length* πl)) (πl @ [π]) =
sem φ (*env1*(*p* := *length* $\pi l1$)) ($\pi l1$ @ [π])
apply(*rule eqOn-FV-sem-NE*) **using** *Exi* **by** *auto*
thus *?case* **unfolding** *sem.simps exi-def* **using** *Exi* **by** (*intro iff-exI conj-cong*) *simp-all*
qed(*auto simp*: *last-map op-defs*)

corollary *FV-sem*:
assumes *wff* φ **and** $\forall$ *p* $\in$ *FV* φ. *env p* = *env1 p*
and *cpt* (*FV* φ) *env* πl **and** *cpt* (*FV* φ) *env1* πl
shows *sem* φ *env* πl = *sem* φ *env1* πl
apply(*rule eqOn-FV-sem*)
using *assms* **unfolding** *eqOn-def* **by** *auto*

As a consequence, the interpretation of a closed formula (i.e., a formula with no free variables) will not depend on the environment and, from the list of paths, will only depend on its pointer:

corollary *interp-closed*:
assumes *wff* φ **and** *FV* φ = {} **and** *pointerOf* πl = *pointerOf* $\pi l1$
shows *sem* φ *env* πl = *sem* φ *env1* $\pi l1$
apply(*rule eqOn-FV-sem*)
using *assms* **unfolding** *eqOn-def cpt-def* **by** *auto*

Therefore, it makes sense to define the interpretation of a closed formula by choosing any environment and any list of paths such that its pointer is the initial state (e.g., the empty list) – knowing that the choices are irrelevant.

definition *semClosed* $\varphi \equiv$ *sem* φ (*any*::*env*) (*SOME* πl. *pointerOf* πl = *s0*)

lemma *semClosed*:
assumes φ: *wff* φ *FV* $\varphi = \{\}$ **and** *p*: *pointerOf* πl = *s0*
shows *semClosed* φ = *sem* φ *env* πl
proof–
  have *pointerOf* (*SOME* πl. *pointerOf* πl = *s0*) = *s0*
  by (*rule someI*[*of* - []]) *simp*
  thus *?thesis* **unfolding** *semClosed-def* **using** *interp-closed*[*OF* φ] *p* **by** *auto*
qed

lemma *semClosed-Nil*:
assumes φ: *wff* φ *FV* $\varphi = \{\}$
shows *semClosed* φ = *sem* φ *env* []
using *assms semClosed* **by** *auto*

5.3 The conjunction of a finite set of formulas

This is defined by making the set into a list (by choosing any ordering of the elements) and iterating binary conjunction.

definition *Scon* :: *'aprop dfmla set* ⇒ *'aprop dfmla* **where**
Scon $\varphi s \equiv$ *foldr Con* (*asList* φs) *Tr*

lemma *sem-Scon*[*simp*]:
assumes *finite* φs
shows *sem* (*Scon* φs) *env* = *scon* ((λ φ. *sem* φ *env*) ‘ φs)
proof–
  define φl **where** φl = *asList* φs
  have *sem* (*foldr Con* φl *Tr*) *env* = *scon* ((λ φ. *sem* φ *env*) ‘ (*set* φl))
  by (*induct* φl) (*auto simp*: *scon-def*)
  thus *?thesis* **unfolding** φ*l-def Scon-def* **by** (*metis assms set-asList*)
qed

lemma *FV-Scon*[*simp*]:
assumes *finite* φs
shows *FV* (*Scon* φs) = $\bigcup$ (*FV* ‘ φs)
proof–
  define φl **where** φl = *asList* φs
  have *FV* (*foldr Con* φl *Tr*) = $\bigcup$ (*set* (*map FV* φl))
  by (*induct* φl) (*auto simp*: *der-Op-defs*)
  thus *?thesis* **unfolding** φ*l-def Scon-def* **by** (*metis assms set-map set-asList*)
qed

end-of-context Shallow

6 Noninterference for models with finitely many users, commands and outputs

In the Noninterference section, we showed how to express Goguen-Meseguer noninterference as a shallow HyperCTL* formula. Here we show that, if one assumes finiteness of the sets of users, commands and outputs, then one can express the property as (the denotation of) a syntactic formula. Note that we do *not* need to assume the state space finite – this is important for a potential application to infinite-state systems.

The Goguen-Meseguer security model with finiteness assumptions

locale *GM-sec-model-finite* = *GM-sec-model st0 do out*
  for *st0* :: *'St*
  and *do* :: *'St* ⇒ *'U* ⇒ *'C* ⇒ *'St*
  and *out* :: *'St* ⇒ *'U* ⇒ *'Out*
  +
  assumes *finite-U*: *finite* (*UNIV* :: *'U set*)
  and *finite-C*: *finite* (*UNIV* :: *'C set*)
  and *finite-Out*: *finite* (*UNIV* :: *'Out set*)
begin

lemma *finite-UminusGH*: *finite* (*UNIV* − *GH*)
by (*metis finite-Diff finite-U*)

lemma *finite-GL*: *finite GL*
by (*metis Diff-UNIV finite-Diff2 finite-U*)

definition *EqOnUC* ::
pvar ⇒ *pvar* ⇒ *'U* ⇒ *'C* ⇒ (*'U*,*'C*,*'Out*) *aprop dfmla*
where
EqOnUC p p' u c ≡ *Eq* (*Atom* (*Last u c*) *p*) (*Atom* (*Last u c*) *p'*)

lemma *EqOnUC-eqOnUC*[*simp*]:
assumes *env p* = *i* **and** *env p'* = *i'*
shows *sem* (*EqOnUC p p' u c*) *env* = *eqOnUC i i' u c*
using *assms* **unfolding** *EqOnUC-def eqOnUC-def* **by** *simp*

definition *EqButGH* ::
pvar ⇒ *pvar* ⇒ (*'U*,*'C*,*'Out*) *aprop dfmla*
where
EqButGH p p' ≡ *Scon* {*EqOnUC p p' u c* | *u c*. (*u*,*c*) ∈ (*UNIV* − *GH*) × *UNIV*}

lemma *finite-EqButGH*:
finite {*EqOnUC p p' u c* | *u c*. (*u*,*c*) ∈ (*UNIV* − *GH*) × *UNIV*} (**is** *finite ?K*)
proof−
  have *1*: *?K* = (λ (*u*,*c*). *EqOnUC p p' u c*) ‘ ((*UNIV* − *GH*) × *UNIV*) **by** *auto*

show *?thesis* **unfolding** *1* **apply**(*rule finite-imageI*)
by (*metis finite-C finite-SigmaI finite-UminusGH*)
qed

lemma *EqButGH-eqButGH*[*simp*]:
assumes *env p* = *i* **and** *env p′* = *i′*
shows *sem* (*EqButGH p p′*) *env* = *eqButGH i i′*
using *assms finite-EqButGH*
unfolding *EqButGH-def eqButGH-def sem-Scon*[*OF finite-EqButGH*] *image-def*
by *simp* (*metis* (*hide-lams*, *no-types*) *EqOnUC-eqOnUC*)

lemma *FV-EqButGH*: *FV* (*EqButGH p p′*) ⊆ {*p*,*p′*} (**is** *?L* ⊆ *?R*)
proof−
have *?L* = ⋃ {*FV* (*EqOnUC p p′ u c*) | *u c*. (*u*,*c*) ∈ (*UNIV* − *GH*) × *UNIV*}
unfolding *EqButGH-def FV-Scon*[*OF finite-EqButGH*] **by** *auto*
also have ... ⊆ *?R* **unfolding** *EqOnUC-def der-Op-defs* **by** *auto*
finally show *?thesis* **.**
qed

definition *EqOnUOut* ::
pvar ⇒ *pvar* ⇒ *′U* ⇒ *′Out* ⇒ (*′U*,*′C*,*′Out*) *aprop dfmla*
where
EqOnUOut p p′ u ou ≡ *Eq* (*Atom* (*Obs u ou*) *p*) (*Atom* (*Obs u ou*) *p′*)

lemma *EqOnUOut-eqOnUOut*[*simp*]:
assumes *env p* = *i* **and** *env p′* = *i′*
shows *sem* (*EqOnUOut p p′ u ou*) *env* = *eqOnUOut i i′ u ou*
using *assms* **unfolding** *EqOnUOut-def eqOnUOut-def* **by** *simp*

definition *EqOnGL* ::
pvar ⇒ *pvar* ⇒ (*′U*,*′C*,*′Out*) *aprop dfmla*
where
EqOnGL p p′ ≡ *Scon* {*EqOnUOut p p′ u ou* | *u ou*. (*u*,*ou*) ∈ *GL* × *UNIV*}

lemma *finite-EqOnGL*:
finite {*EqOnUOut p p′ u ou* | *u ou*. (*u*,*ou*) ∈ *GL* × *UNIV*} (**is** *finite ?K*)
proof−
have *1*: *?K* = (λ (*u*,*ou*). *EqOnUOut p p′ u ou*) ‘ (*GL* × *UNIV*) **by** *auto*
show *?thesis* **unfolding** *1* **apply**(*rule finite-imageI*)
by (*metis finite-Out finite-SigmaI finite-GL*)
qed

lemma *EqOnGL-eqOnGL*[*simp*]:
assumes *env p* = *i* **and** *env p′* = *i′*
shows *sem* (*EqOnGL p p′*) *env* = *eqOnGL i i′*
using *assms finite-EqOnGL*
unfolding *EqOnGL-def eqOnGL-def sem-Scon*[*OF finite-EqOnGL*] *image-def*
by *simp* (*metis* (*hide-lams*, *no-types*) *EqOnUOut-eqOnUOut*)

lemma *FV-EqOnGL*: *FV* (*EqOnGL p p′*) ⊆ {*p*,*p′*} (**is** *?L* ⊆ *?R*)
proof−
have *?L* = ⋃ {*FV* (*EqOnUOut p p′ u ou*) | *u ou*. (*u*,*ou*) ∈ *GL* × *UNIV*}
unfolding *EqOnGL-def FV-Scon*[*OF finite-EqOnGL*] **by** *auto*
also have ... ⊆ *?R* **unfolding** *EqOnUOut-def der-Op-defs* **by** *auto*
finally show *?thesis* **.**
qed

definition *p0* = *getFresh* {}
definition *p1* = *getFresh* {*p0*}

lemma *p0p1*[*simp*]: *p0* ≠ *p1* **unfolding** *p1-def*
by (*metis Diff-cancel getFresh infinite-imp-nonempty infinite-remove insertI1*)

definition *nonintDfmla* :: (′*U*,′*C*,′*Out*) *aprop dfmla* **where**
nonintDfmla ≡
Fall2 p0 p1 (*Imp* (*Alw* (*EqButGH p0 p1*)) (*Alw* (*EqOnGL p0 p1*)))

lemma *sem-nonintDfmla*: *sem nonintDfmla env* = *nonintSfmla*
unfolding *nonintDfmla-def nonintSfmla-def* **by** *simp*

lemma *wff-nonintDfmla*[*simp*]: *wff nonintDfmla*
unfolding *nonintDfmla-def Fall2-def Fall-def* **by** *simp*

lemma *closed-nonintDfmla*[*simp*]: *FV nonintDfmla* = {}
unfolding *nonintDfmla-def Fall2-def Fall-def der-Op-defs*
using *FV-EqButGH FV-EqOnGL* **by** *fastforce*

In the end, we obtain that the semantics of the closed (syntactic) formula nonintDfmla expresses noninterference faithfully:

theorem *semClosed-nonintDfmla*: *semClosed nonintDfmla* = *nonint*
unfolding *nonintSfmla-iff-nonint*[*symmetric*]
apply(*subst sem-nonintDfmla*[*symmetric*]) **apply**(*rule semClosed-Nil*) **by** *auto*

end-of-context GM-sec-model-finite