Constructive Cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar

April 19, 2020

Abstract

Inspired by Abstract Cryptography [6], we extend CryptHOL [1, 4], a framework for formalizing game-based proofs, with an abstract model of Random Systems [7] and provide proof rules about their composition and equality. This foundation facilitates the formalization of Constructive Cryptography [5] proofs, where the security of a cryptographic scheme is realized as a special form of construction in which a complex random system is built from simpler ones. This is a first step towards a fully-featured compositional framework, similar to Universal Composability framework [2], that supports formalization of simulation-based proofs [3].

Contents

theory *More-CryptHOL* **imports**
CryptHOL.CryptHOL
begin

lemma *is-empty-image* [*simp*]: *Set.is-empty* (*f* ‘ *A*) = *Set.is-empty A*
by(*auto simp add*: *Set.is-empty-def*)

lemma *inj-on-map-sum* [*simp*]:
⟦ *inj-on f A*; *inj-on g B* ⟧ ⟹ *inj-on* (*map-sum f g*) (*A* <+> *B*)
proof(*rule inj-onI*, *goal-cases*)
case (*1 x y*)
then show *?case* **by**(*cases x*; *cases y*; *auto simp add*: *inj-on-def*)
qed

lemma *inv-into-map-sum*:
inv-into (*A* <+> *B*) (*map-sum f g*) *x* = *map-sum* (*inv-into A f*) (*inv-into B g*)
x
if *x* ∈ *f* ‘ *A* <+> *g* ‘ *B inj-on f A inj-on g B*
using *that* **by**(*cases rule*: *PlusE*[*consumes 1*])(*auto simp add*: *inv-into-f-eq f-inv-into-f*)

lemma *Pair-fst-Unity*: (*fst x*, ()) = *x*
by(*cases x*) *simp*

fun *rsuml* :: (′*a* + ′*b*) + ′*c* ⇒ ′*a* + (′*b* + ′*c*) **where**
rsuml (*Inl* (*Inl a*)) = *Inl a*
| *rsuml* (*Inl* (*Inr b*)) = *Inr* (*Inl b*)
| *rsuml* (*Inr c*) = *Inr* (*Inr c*)

fun *lsumr* :: ′*a* + (′*b* + ′*c*) ⇒ (′*a* + ′*b*) + ′*c* **where**
lsumr (*Inl a*) = *Inl* (*Inl a*)
| *lsumr* (*Inr* (*Inl b*)) = *Inl* (*Inr b*)
| *lsumr* (*Inr* (*Inr c*)) = *Inr c*

lemma *rsuml-lsumr* [*simp*]: *rsuml* (*lsumr x*) = *x*
by(*cases x rule*: *lsumr.cases*) *simp-all*

lemma *lsumr-rsuml* [*simp*]: *lsumr* (*rsuml x*) = *x*
by(*cases x rule*: *rsuml.cases*) *simp-all*

definition *rprodl* :: (′*a* × ′*b*) × ′*c* ⇒ ′*a* × (′*b* × ′*c*) **where** *rprodl* = (λ((*a*, *b*),
c). (*a*, (*b*, *c*)))

lemma *rprodl-simps* [*simp*]: *rprodl* ((*a*, *b*), *c*) = (*a*, (*b*, *c*))
by(*simp add*: *rprodl-def*)

lemma *rprodl-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-prod* (*rel-prod A B*) *C* ===> *rel-prod A* (*rel-prod B C*)) *rprodl rprodl*
unfolding *rprodl-def* **by** *transfer-prover*

definition *lprodr* :: $'a \times ('b \times 'c) \Rightarrow ('a \times 'b) \times 'c$ **where** *lprodr* = (λ(*a*, *b*, *c*). ((*a*, *b*), *c*))

lemma *lprodr-simps* [*simp*]: *lprodr* (*a*, *b*, *c*) = ((*a*, *b*), *c*)
by(*simp add*: *lprodr-def*)

lemma *lprodr-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-prod A* (*rel-prod B C*) ===> *rel-prod* (*rel-prod A B*) *C*) *lprodr lprodr*
unfolding *lprodr-def* **by** *transfer-prover*

lemma *lprodr-inverse* [*simp*]: *rprodl* (*lprodr x*) = *x*
by(*cases x*) *auto*

lemma *rprodl-inverse* [*simp*]: *lprodr* (*rprodl x*) = *x*
by(*cases x*) *auto*

lemma *rel-fun-comp*:
$\bigwedge$*f g h*. *rel-fun A B* (*f* $\circ$ *g*) *h* = *rel-fun A* (λ*x*. *B* (*f x*)) *g h*
$\bigwedge$*f g h*. *rel-fun A B f* (*g* $\circ$ *h*) = *rel-fun A* (λ*x y*. *B x* (*g y*)) *f h*
by(*auto simp add*: *rel-fun-def*)

lemma *rel-fun-map-fun1*: *rel-fun* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *A f g* $\Longrightarrow$ *rel-fun* (=) *A* (*map-fun h id f*) *g*
by(*auto simp add*: *rel-fun-def Grp-def*)

lemma *rel-fun-map-fun2*: *rel-fun* (*eq-on* (*range h*)) *A f g* $\Longrightarrow$ *rel-fun* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *A f* (*map-fun h id g*)
by(*auto simp add*: *rel-fun-def Grp-def eq-onp-def*)

lemma *map-fun2-id*: *map-fun f g x* = *g* $\circ$ *map-fun f id x*
by(*simp add*: *map-fun-def o-assoc*)

lemma *rel-fun-refl-eq-onp*:
($\bigwedge$*z*. *z* $\in$ *f* ‘ *X* $\Longrightarrow$ *A z z*) $\Longrightarrow$ *rel-fun* (*eq-on X*) *A f f*
by(*auto simp add*: *rel-fun-def eq-onp-def*)

lemma *map-fun-id2-in*: *map-fun g h f* = *map-fun g id* (*h* $\circ$ *f*)
by(*simp add*: *map-fun-def*)

lemma *Domainp-rel-fun-le*: *Domainp* (*rel-fun A B*) $\leq$ *pred-fun* (*Domainp A*) (*Domainp B*)
by(*auto dest*: *rel-funD*)

lemma *eq-onE*: $[\![$ *eq-on X a b*; $[\![$ *b* $\in$ *X*; *a* = *b* $]\!] \Longrightarrow$ *thesis* $]\!] \Longrightarrow$ *thesis* **by** *auto*

lemma *Domainp-eq-on* [*simp*]: *Domainp* (*eq-on X*) = (λ*x*. *x* ∈ *X*)
by *auto*

declare *eq-on-def* [*simp del*]

lemma *pred-prod-mono'* [*mono*]:
pred-prod A B xy ⟶ *pred-prod A' B' xy*
if ⋀*x*. *A x* ⟶ *A' x* ⋀*y*. *B y* ⟶ *B' y*
using *that* **by**(*cases xy*) *auto*

fun *rel-witness-prod* :: ('*a* × '*b*) × ('*c* × '*d*) ⇒ (('*a* × '*c*) × ('*b* × '*d*)) **where**
rel-witness-prod ((*a*, *b*), (*c*, *d*)) = ((*a*, *c*), (*b*, *d*))

consts *relcompp-witness* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*b* ⇒ '*c* ⇒ *bool*) ⇒ '*a* × '*c* ⇒ '*b*
specification (*relcompp-witness*)
relcompp-witness1: (*A OO B*) (*fst xy*) (*snd xy*) ⟹ *A* (*fst xy*) (*relcompp-witness A B xy*)
relcompp-witness2: (*A OO B*) (*fst xy*) (*snd xy*) ⟹ *B* (*relcompp-witness A B xy*) (*snd xy*)
apply(*fold all-conj-distrib*)
apply(*rule choice allI*)+
by(*auto intro*: *choice allI*)

lemmas *relcompp-witness*[*of* - - (*x*, *y*) **for** *x y*, *simplified*] = *relcompp-witness1 relcompp-witness2*

hide-fact (**open**) *relcompp-witness1 relcompp-witness2*

lemma *relcompp-witness-eq* [*simp*]: *relcompp-witness* (=) (=) (*x*, *x*) = *x*
using *relcompp-witness*(*1*)[*of* (=) (=) *x x*] **by**(*simp add*: *eq-OO*)

fun *rel-witness-option* :: '*a option* × '*b option* ⇒ ('*a* × '*b*) *option* **where**
rel-witness-option (*Some x*, *Some y*) = *Some* (*x*, *y*)
| *rel-witness-option* (*None*, *None*) = *None*
| *rel-witness-option* - = *None* — Just to make the definition complete

lemma *rel-witness-option*:
shows *set-rel-witness-option*: ⟦ *rel-option A x y*; (*a*, *b*) ∈ *set-option* (*rel-witness-option* (*x*, *y*)) ⟧ ⟹ *A a b*
and *map1-rel-witness-option*: *rel-option A x y* ⟹ *map-option fst* (*rel-witness-option* (*x*, *y*)) = *x*
and *map2-rel-witness-option*: *rel-option A x y* ⟹ *map-option snd* (*rel-witness-option* (*x*, *y*)) = *y*
by(*cases* (*x*, *y*) *rule*: *rel-witness-option.cases*; *simp*; *fail*)+

lemma *rel-witness-option1*:
assumes *rel-option A x y*
shows *rel-option* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) *x* (*rel-witness-option* (*x*, *y*))

using *map1-rel-witness-option*[*OF assms*, *symmetric*]
unfolding *option.rel-eq*[*symmetric*] *option.rel-map*
by(*rule option.rel-mono-strong*)(*auto intro*: *set-rel-witness-option*[*OF assms*])

lemma *rel-witness-option2*:
assumes *rel-option A x y*
shows *rel-option* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (*rel-witness-option* (*x*, *y*)) *y*
using *map2-rel-witness-option*[*OF assms*]
unfolding *option.rel-eq*[*symmetric*] *option.rel-map*
by(*rule option.rel-mono-strong*)(*auto intro*: *set-rel-witness-option*[*OF assms*])

definition *rel-witness-fun* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*b* ⇒ '*c* ⇒ *bool*) ⇒ ('*a* ⇒ '*d*) × ('*c* ⇒ '*e*) ⇒ ('*b* ⇒ '*d* × '*e*) **where**
rel-witness-fun A A' = (λ(*f*, *g*) *b*. (*f* (*THE a*. *A a b*), *g* (*THE c*. *A' b c*)))

lemma
assumes *fg*: *rel-fun* (*A OO A'*) *B f g*
and *A*: *left-unique A right-total A*
and *A'*: *right-unique A' left-total A'*
shows *rel-witness-fun1*: *rel-fun A* (λ*x* (*x'*, *y*). *x* = *x'* ∧ *B x' y*) *f* (*rel-witness-fun A A'* (*f*, *g*))
and *rel-witness-fun2*: *rel-fun A'* (λ(*x*, *y'*) *y*. *y* = *y'* ∧ *B x y'*) (*rel-witness-fun A A'* (*f*, *g*)) *g*
proof (*goal-cases*)
case *1*
have *A x y* ⟹ *f x* = *f* (*THE a*. *A a y*) ∧ *B* (*f* (*THE a*. *A a y*)) (*g* (*The* (*A' y*))) **for** *x y*
by(*rule left-totalE*[*OF A'*(*2*)]; *erule meta-allE*[*of - y*]; *erule exE*; *frule* (*1*) *fg*[*THEN rel-funD*, *OF relcomppI*])
(*auto intro*!: *arg-cong*[**where** *f*=*f*] *arg-cong*[**where** *f*=*g*] *rel-funI the-equality the-equality*[*symmetric*] *dest*: *left-uniqueD*[*OF A*(*1*)] *right-uniqueD*[*OF A'*(*1*)] *elim*!: *arg-cong2*[**where** *f*=*B*, *THEN iffD2*, *rotated* −*1*])

with *1* **show** *?case* **by**(*clarsimp simp add*: *rel-fun-def rel-witness-fun-def*)
next
case *2*
have *A' x y* ⟹ *g y* = *g* (*The* (*A' x*)) ∧ *B* (*f* (*THE a*. *A a x*)) (*g* (*The* (*A' x*))) **for** *x y*
by(*rule right-totalE*[*OF A*(*2*), *of x*]; *frule* (*1*) *fg*[*THEN rel-funD*, *OF relcomppI*])
(*auto intro*!: *arg-cong*[**where** *f*=*f*] *arg-cong*[**where** *f*=*g*] *rel-funI the-equality the-equality*[*symmetric*] *dest*: *left-uniqueD*[*OF A*(*1*)] *right-uniqueD*[*OF A'*(*1*)] *elim*!: *arg-cong2*[**where** *f*=*B*, *THEN iffD2*, *rotated* −*1*])

with *2* **show** *?case* **by**(*clarsimp simp add*: *rel-fun-def rel-witness-fun-def*)
qed

lemma *rel-witness-fun-eq* [*simp*]: *rel-witness-fun* (=) (=) (*f*, *g*) = (λ*x*. (*f x*, *g x*))

by(*simp add*: *rel-witness-fun-def*)

consts *rel-witness-pmf* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a pmf* × '*b pmf* ⇒ ('*a* × '*b*) *pmf*
specification (*rel-witness-pmf*)
set-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) ⟹ *set-pmf* (*rel-witness-pmf A xy*) ⊆ {(*a*, *b*). *A a b*}
map1-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) ⟹ *map-pmf fst* (*rel-witness-pmf A xy*) = *fst xy*
map2-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) ⟹ *map-pmf snd* (*rel-witness-pmf A xy*) = *snd xy*
apply(*fold all-conj-distrib imp-conjR*)
apply(*rule choice allI*)+
apply(*unfold pmf.in-rel*)
by *blast*

lemmas *set-rel-witness-pmf* = *set-rel-witness-pmf'*[*of* - (*x*, *y*) **for** *x y*, *simplified*]
lemmas *map1-rel-witness-pmf* = *map1-rel-witness-pmf'*[*of* - (*x*, *y*) **for** *x y*, *simplified*]
lemmas *map2-rel-witness-pmf* = *map2-rel-witness-pmf'*[*of* - (*x*, *y*) **for** *x y*, *simplified*]
lemmas *rel-witness-pmf* = *set-rel-witness-pmf map1-rel-witness-pmf map2-rel-witness-pmf*

lemma *rel-witness-pmf1*:
assumes *rel-pmf A p q*
shows *rel-pmf* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) *p* (*rel-witness-pmf A* (*p*, *q*))
using *map1-rel-witness-pmf*[*OF assms*, *symmetric*]
unfolding *pmf.rel-eq*[*symmetric*] *pmf.rel-map*
by(*rule pmf.rel-mono-strong*)(*auto dest*: *set-rel-witness-pmf*[*OF assms*, *THEN subsetD*])

lemma *rel-witness-pmf2*:
assumes *rel-pmf A p q*
shows *rel-pmf* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (*rel-witness-pmf A* (*p*, *q*)) *q*
using *map2-rel-witness-pmf*[*OF assms*]
unfolding *pmf.rel-eq*[*symmetric*] *pmf.rel-map*
by(*rule pmf.rel-mono-strong*)(*auto dest*: *set-rel-witness-pmf*[*OF assms*, *THEN subsetD*])

definition *rel-witness-spmf* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a spmf* × '*b spmf* ⇒ ('*a* × '*b*) *spmf* **where**
rel-witness-spmf A = *map-pmf rel-witness-option* ∘ *rel-witness-pmf* (*rel-option A*)

lemma assumes *rel-spmf A p q*
shows *rel-witness-spmf1*: *rel-spmf* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) *p* (*rel-witness-spmf A* (*p*, *q*))
and *rel-witness-spmf2*: *rel-spmf* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (*rel-witness-spmf A* (*p*, *q*)) *q*

by(*auto simp add*: *pmf.rel-map rel-witness-spmf-def intro*: *pmf.rel-mono-strong*[*OF rel-witness-pmf1*[*OF assms*]] *rel-witness-option1 pmf.rel-mono-strong*[*OF rel-witness-pmf2*[*OF assms*]] *rel-witness-option2*)

primrec (*transfer*) *enforce-option* :: ('*a* ⇒ *bool*) ⇒ '*a option* ⇒ '*a option* **where**
  enforce-option P (*Some x*) = (*if P x then Some x else None*)
| *enforce-option P None* = *None*

lemma *set-enforce-option* [*simp*]: *set-option* (*enforce-option P x*) = {*a* ∈ *set-option x*. *P a*}
  by(*cases x*) *auto*

lemma *enforce-map-option*: *enforce-option P* (*map-option f x*) = *map-option f* (*enforce-option* (*P* ∘ *f*) *x*)
  by(*cases x*) *auto*

lemma *enforce-bind-option* [*simp*]:
  enforce-option P (*Option.bind x f*) = *Option.bind x* (*enforce-option P* ∘ *f*)
  by(*cases x*) *auto*

lemma *enforce-option-alt-def*:
  enforce-option P x = *Option.bind x* (λ*a*. *Option.bind* (*assert-option* (*P a*)) (λ-:: *unit*. *Some a*))
  by(*cases x*) *simp-all*

lemma *enforce-option-eq-None-iff* [*simp*]:
  enforce-option P x = *None* ⟷ (∀ *a*. *x* = *Some a* ⟶ ¬ *P a*)
  by(*cases x*) *auto*

lemma *enforce-option-eq-Some-iff* [*simp*]:
  enforce-option P x = *Some y* ⟷ *x* = *Some y* ∧ *P y*
  by(*cases x*) *auto*

lemma *Some-eq-enforce-option-iff* [*simp*]:
  Some y = *enforce-option P x* ⟷ *x* = *Some y* ∧ *P y*
  by(*cases x*) *auto*

lemma *enforce-option-top* [*simp*]: *enforce-option* ⊤ = *id*
  by(*rule ext*; *rename-tac x*; *case-tac x*; *simp*)

lemma *enforce-option-K-True* [*simp*]: *enforce-option* (λ-. *True*) *x* = *x*
  by(*cases x*) *simp-all*

lemma *enforce-option-bot* [*simp*]: *enforce-option* ⊥ = (λ-. *None*)
  by(*simp add*: *fun-eq-iff*)

lemma *enforce-option-K-False* [*simp*]: *enforce-option* (λ-. *False*) *x* = *None*
  by *simp*

lemma *enforce-pred-id-option*: *pred-option P x* $\Longrightarrow$ *enforce-option P x* = *x*
by(*cases x*) *auto*

lemma *rel-fun-refl*: $[\![A \leq (=); (=) \leq B]\!] \Longrightarrow (=) \leq$ *rel-fun A B*
by(*subst fun.rel-eq*[*symmetric*])(*rule fun-mono*)

lemma *rel-fun-mono-strong*:
$[\![$ *rel-fun A B f g*; $A' \leq A$; $\bigwedge x\ y.\ [\![x \in f$ ‘ {*x*. *Domainp A' x*}; $y \in g$ ‘ {*x*. *Rangep A' x*}; *B x y* $]\!] \Longrightarrow B'\ x\ y\ ]\!] \Longrightarrow$ *rel-fun A' B' f g*
by(*auto simp add*: *rel-fun-def*) *fastforce*

lemma *rel-fun-refl-strong*:
assumes $A \leq (=)$ $\bigwedge x.\ x \in f$ ‘ {*x*. *Domainp A x*} $\Longrightarrow B\ x\ x$
shows *rel-fun A B f f*
proof −
have *rel-fun* (=) (=) *f f* **by**(*simp add*: *rel-fun-eq*)
then show *?thesis* **using** *assms*(*1*)
by(*rule rel-fun-mono-strong*) (*auto intro*: *assms*(*2*))
qed

lemma *Grp-iff*: *BNF-Def.Grp B g x y* $\longleftrightarrow$ $y = g\ x \wedge x \in B$ **by**(*simp add*: *Grp-def*)

lemma *Rangep-Grp*: *Rangep* (*BNF-Def.Grp A f*) = ($\lambda x.\ x \in f$ ‘ *A*)
by(*auto simp add*: *fun-eq-iff Grp-iff*)

lemma *Domainp-Grp*: *Domainp* (*BNF-Def.Grp A f*) = ($\lambda x.\ x \in A$)
by(*auto simp add*: *Grp-iff fun-eq-iff*)

lemma *rel-fun-Grp*:
rel-fun (*BNF-Def.Grp UNIV h*)$^{-1-1}$ (*BNF-Def.Grp A g*) = *BNF-Def.Grp* {*f*. *f* ‘ *range h* $\subseteq$ *A*} (*map-fun h g*)
by(*auto simp add*: *rel-fun-def fun-eq-iff Grp-iff*)

lemma *wf-strict-prefix*: *wfP strict-prefix*
proof −
from *wf* **have** *wf* (*inv-image* {(*x*, *y*). *x* < *y*} *length*) **by**(*rule wf-inv-image*)
moreover have {(*x*, *y*). *strict-prefix x y*} $\subseteq$ *inv-image* {(*x*, *y*). *x* < *y*} *length* **by**(*auto intro*: *prefix-length-less*)
ultimately show *?thesis* **unfolding** *wfP-def* **by**(*rule wf-subset*)
qed

lemma *strict-prefix-setD*:
strict-prefix xs ys $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
by(*auto simp add*: *strict-prefix-def prefix-def*)

lemma *weight-assert-spmf* [*simp*]: *weight-spmf* (*assert-spmf b*) = *indicator* {*True*} *b*
by(*simp split*: *split-indicator*)

definition *enforce-spmf* :: ('*a* ⇒ *bool*) ⇒ '*a spmf* ⇒ '*a spmf* **where**
enforce-spmf P = *map-pmf* (*enforce-option P*)

lemma *enforce-spmf-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*A* ===> (=)) ===> *rel-spmf A* ===> *rel-spmf A*) *enforce-spmf enforce-spmf*
unfolding *enforce-spmf-def* **by** *transfer-prover*

lemma *enforce-return-spmf* [*simp*]:
enforce-spmf P (*return-spmf x*) = (*if P x then return-spmf x else return-pmf None*)
by(*simp add*: *enforce-spmf-def*)

lemma *enforce-return-pmf-None* [*simp*]:
enforce-spmf P (*return-pmf None*) = *return-pmf None*
by(*simp add*: *enforce-spmf-def*)

lemma *enforce-map-spmf*:
enforce-spmf P (*map-spmf f p*) = *map-spmf f* (*enforce-spmf* (*P* ∘ *f*) *p*)
by(*simp add*: *enforce-spmf-def pmf.map-comp o-def enforce-map-option*)

lemma *enforce-bind-spmf* [*simp*]:
enforce-spmf P (*bind-spmf p f*) = *bind-spmf p* (*enforce-spmf P* ∘ *f*)
by(*auto simp add*: *enforce-spmf-def bind-spmf-def map-bind-pmf intro*!: *bind-pmf-cong split*: *option.split*)

lemma *set-enforce-spmf* [*simp*]: *set-spmf* (*enforce-spmf P p*) = {*a* ∈ *set-spmf p*. *P a*}
by(*auto simp add*: *enforce-spmf-def in-set-spmf*)

lemma *enforce-spmf-alt-def*:
enforce-spmf P p = *bind-spmf p* (λ*a*. *bind-spmf* (*assert-spmf* (*P a*)) (λ- :: *unit*. *return-spmf a*))
by(*auto simp add*: *enforce-spmf-def assert-spmf-def map-pmf-def bind-spmf-def bind-return-pmf intro*!: *bind-pmf-cong split*: *option.split*)

lemma *bind-enforce-spmf* [*simp*]:
bind-spmf (*enforce-spmf P p*) *f* = *bind-spmf p* (λ*x*. *if P x then f x else return-pmf None*)
by(*auto simp add*: *enforce-spmf-alt-def assert-spmf-def intro*!: *bind-spmf-cong*)

lemma *weight-enforce-spmf*:
weight-spmf (*enforce-spmf P p*) = *weight-spmf p* − *measure* (*measure-spmf p*) {*x*. ¬ *P x*} (**is** *?lhs* = *?rhs*)
proof −

have *?lhs* = *LINT x*|*measure-spmf p*. *indicator* {*x*. *P x*} *x*
by(*auto simp add*: *enforce-spmf-alt-def weight-bind-spmf o-def simp del*: *Bochner-Integration.integral-indicat*
intro!: *Bochner-Integration.integral-cong split*: *split-indicator*)
also have ... = *?rhs*
by(*subst measure-spmf.finite-measure-Diff*[*symmetric*])(*auto simp add*: *space-measure-spmf*
intro!: *arg-cong2*[**where** *f*=*measure*])
finally show *?thesis* **.**
qed

lemma *lossless-enforce-spmf* [*simp*]:
lossless-spmf (*enforce-spmf P p*) ⟷ *lossless-spmf p* ∧ *set-spmf p* ⊆ {*x*. *P x*}
by(*auto simp add*: *enforce-spmf-alt-def*)

lemma *enforce-spmf-top* [*simp*]: *enforce-spmf* ⊤ = *id*
by(*simp add*: *enforce-spmf-def*)

lemma *enforce-spmf-K-True* [*simp*]: *enforce-spmf* (λ-. *True*) *p* = *p*
using *enforce-spmf-top*[*THEN fun-cong*, *of p*] **by**(*simp add*: *top-fun-def*)

lemma *enforce-spmf-bot* [*simp*]: *enforce-spmf* ⊥ = (λ-. *return-pmf None*)
by(*simp add*: *enforce-spmf-def fun-eq-iff*)

lemma *enforce-spmf-K-False* [*simp*]: *enforce-spmf* (λ-. *False*) *p* = *return-pmf None*
using *enforce-spmf-bot*[*THEN fun-cong*, *of p*] **by**(*simp add*: *bot-fun-def*)

lemma *enforce-pred-id-spmf*: *enforce-spmf P p* = *p* **if** *pred-spmf P p*
proof −
have *enforce-spmf P p* = *map-pmf id p* **using** *that*
by(*auto simp add*: *enforce-spmf-def enforce-pred-id-option simp del*: *map-pmf-id*
intro!: *pmf.map-cong-pred*[*OF refl*] *elim*!: *pmf-pred-mono-strong*)
then show *?thesis* **by** *simp*
qed

lemma *map-the-spmf-of-pmf* [*simp*]: *map-pmf the* (*spmf-of-pmf p*) = *p*
by(*simp add*: *spmf-of-pmf-def pmf.map-comp o-def*)

lemma *bind-bind-conv-pair-spmf*:
bind-spmf p (λ*x*. *bind-spmf q* (*f x*)) = *bind-spmf* (*pair-spmf p q*) (λ(*x*, *y*). *f x y*)
by(*simp add*: *pair-spmf-alt-def*)

lemma *cond-pmf-of-set*:
assumes *fin*: *finite A* **and** *nonempty*: *A* ∩ *B* ≠ {}
shows *cond-pmf* (*pmf-of-set A*) *B* = *pmf-of-set* (*A* ∩ *B*) (**is** *?lhs* = *?rhs*)
proof(*rule pmf-eqI*)
from *nonempty* **have** *A*: *A* ≠ {} **by** *auto*
show *pmf ?lhs x* = *pmf ?rhs x* **for** *x*
by(*subst pmf-cond*; *clarsimp simp add*: *fin A nonempty measure-pmf-of-set split*:
split-indicator)
qed

lemma *cond-spmf-spmf-of-set*:
cond-spmf (*spmf-of-set A*) *B* = *spmf-of-set* (*A* ∩ *B*) **if** *finite A*
by(*rule spmf-eqI*)(*auto simp add*: *spmf-of-set measure-spmf-of-set that split*: *split-indicator*)

lemma *pair-pmf-of-set*:
assumes *A*: *finite A A* ≠ {}
and *B*: *finite B B* ≠ {}
shows *pair-pmf* (*pmf-of-set A*) (*pmf-of-set B*) = *pmf-of-set* (*A* × *B*)
by(*rule pmf-eqI*)(*clarsimp simp add*: *pmf-pair assms split*: *split-indicator*)

lemma *pair-spmf-of-set*:
pair-spmf (*spmf-of-set A*) (*spmf-of-set B*) = *spmf-of-set* (*A* × *B*)
by(*rule spmf-eqI*)(*clarsimp simp add*: *spmf-of-set card-cartesian-product split*: *split-indicator*)

lemma *emeasure-cond-pmf*:
fixes *p A*
defines *q* ≡ *cond-pmf p A*
assumes *set-pmf p* ∩ *A* ≠ {}
shows *emeasure* (*measure-pmf q*) *B* = *emeasure* (*measure-pmf p*) (*A* ∩ *B*) / *emeasure* (*measure-pmf p*) *A*
proof −
note [*transfer-rule*] = *cond-pmf.transfer*[*OF assms*(*2*), *folded q-def*]
interpret *pmf-as-measure* **.**
show *?thesis* **by** *transfer simp*
qed

lemma *measure-cond-pmf*:
measure (*measure-pmf* (*cond-pmf p A*)) *B* = *measure* (*measure-pmf p*) (*A* ∩ *B*) / *measure* (*measure-pmf p*) *A*
if *set-pmf p* ∩ *A* ≠ {}
using *emeasure-cond-pmf*[*OF that*, *of B*] *that*
by(*auto simp add*: *measure-pmf.emeasure-eq-measure measure-pmf-posI divide-ennreal*)

lemma *emeasure-measure-pmf-zero-iff*: *emeasure* (*measure-pmf p*) *s* = *0* ⟷ *set-pmf p* ∩ *s* = {} (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* ⟷ (*AE x in measure-pmf p*. *x* ∉ *s*)
by(*subst AE-iff-measurable*)(*auto*)
also have ... = *?rhs* **by**(*auto simp add*: *AE-measure-pmf-iff*)
finally show *?thesis* **.**
qed

lemma *emeasure-cond-spmf*:
emeasure (*measure-spmf* (*cond-spmf p A*)) *B* = *emeasure* (*measure-spmf p*) (*A*

$\cap$ B) / *emeasure* (*measure-spmf p*) A
apply(*clarsimp simp add*: *cond-spmf-def emeasure-measure-spmf-conv-measure-pmf emeasure-measure-pmf-zero-iff set-pmf-Int-Some split*!: *if-split*)
apply *blast*
apply(*subst* (*asm*) *emeasure-cond-pmf*)
by(*auto simp add*: *set-pmf-Int-Some image-Int*)

lemma *measure-cond-spmf*:
measure (*measure-spmf* (*cond-spmf p A*)) B = *measure* (*measure-spmf p*) ($A \cap B$) / *measure* (*measure-spmf p*) A
apply(*clarsimp simp add*: *cond-spmf-def measure-measure-spmf-conv-measure-pmf measure-pmf-zero-iff set-pmf-Int-Some split*!: *if-split*)
apply(*subst* (*asm*) *measure-cond-pmf*)
by(*auto simp add*: *image-Int set-pmf-Int-Some*)

lemma *lossless-cond-spmf* [*simp*]: *lossless-spmf* (*cond-spmf p A*) $\longleftrightarrow$ *set-spmf p* $\cap$ $A \neq \{\}$
by(*clarsimp simp add*: *cond-spmf-def lossless-iff-set-pmf-None set-pmf-Int-Some*)

lemma *measure-spmf-eq-density*: *measure-spmf p* = *density* (*count-space UNIV*) (*spmf p*)
by(*rule measure-eqI*)(*simp-all add*: *emeasure-density nn-integral-spmf*[*symmetric*] *nn-integral-count-space-indicator*)

lemma *integral-measure-spmf*:
fixes f :: $'a \Rightarrow 'b$::{*banach*, *second-countable-topology*}
assumes A: *finite A*
shows ($\bigwedge a.\ a \in$ *set-spmf M* $\Longrightarrow f\ a \neq 0 \Longrightarrow a \in A$) $\Longrightarrow$ (*LINT x*|*measure-spmf M*. $f\ x$) = ($\sum a \in A.$ *spmf M a* $*_R$ $f\ a$)
unfolding *measure-spmf-eq-density*
apply (*simp add*: *integral-density*)
apply (*subst lebesgue-integral-count-space-finite-support*)
by (*auto intro*!: *finite-subset*[*OF* - ‹*finite A*›] *sum.mono-neutral-left simp*: *spmf-eq-0-set-spmf*)

lemma *image-set-spmf-eq*:
f ‘ *set-spmf p* = g ‘ *set-spmf q* **if** *ASSUMPTION* (*map-spmf f p* = *map-spmf g q*)
using *that*[*unfolded ASSUMPTION-def*, *THEN arg-cong*[**where** *f*=*set-spmf*]]
by *simp*

lemma *map-spmf-const*: *map-spmf* (λ-. x) p = *scale-spmf* (*weight-spmf p*) (*return-spmf x*)
by(*simp add*: *map-spmf-conv-bind-spmf bind-spmf-const*)

lemma *cond-return-pmf* [*simp*]: *cond-pmf* (*return-pmf x*) A = *return-pmf x* **if** $x \in A$
using *that* **by**(*intro pmf-eqI*)(*auto simp add*: *pmf-cond split*: *split-indicator*)

lemma *cond-return-spmf* [*simp*]: *cond-spmf* (*return-spmf x*) *A* = (*if x* ∈ *A then return-spmf x else return-pmf None*)
by(*simp add*: *cond-spmf-def*)

lemma *measure-range-Some-eq-weight*:
measure (*measure-pmf p*) (*range Some*) = *weight-spmf p*
by (*simp add*: *measure-measure-spmf-conv-measure-pmf space-measure-spmf*)

lemma *restrict-spmf-eq-return-pmf-None* [*simp*]:
restrict-spmf p A = *return-pmf None* ⟷ *set-spmf p* ∩ *A* = {}
by(*auto 4 3 simp add*: *restrict-spmf-def map-pmf-eq-return-pmf-iff bind-UNION in-set-spmf bind-eq-None-conv option.the-def dest*: *bspec split*: *if-split-asm option.split-asm*)

lemma *integrable-scale-measure* [*simp*]:
⟦ *integrable M f*; *r* < ⊤ ⟧ ⟹ *integrable* (*scale-measure r M*) *f*
for *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
by(*auto simp add*: *integrable-iff-bounded nn-integral-scale-measure ennreal-mult-less-top*)

lemma *integral-scale-measure*:
assumes *integrable M f r* < ⊤
shows *integral*^{L} (*scale-measure r M*) *f* = *enn2real r* ∗ *integral*^{L} *M f*
using *assms*
apply(*subst* (*1 2*) *real-lebesgue-integral-def*)
apply(*simp-all add*: *nn-integral-scale-measure ennreal-enn2real-if*)
by(*auto simp add*: *ennreal-mult-less-top ennreal-less-top-iff ennreal-mult-eq-top-iff enn2real-mult right-diff-distrib elim*!: *integrableE*)

definition *mk-lossless* :: *'a spmf* ⇒ *'a spmf* **where**
mk-lossless p = *scale-spmf* (*inverse* (*weight-spmf p*)) *p*

lemma *mk-lossless-idem* [*simp*]: *mk-lossless* (*mk-lossless p*) = *mk-lossless p*
by(*simp add*: *mk-lossless-def weight-scale-spmf min-def max-def inverse-eq-divide*)

lemma *mk-lossless-return* [*simp*]: *mk-lossless* (*return-pmf x*) = *return-pmf x*
by(*cases x*)(*simp-all add*: *mk-lossless-def*)

lemma *mk-lossless-map* [*simp*]: *mk-lossless* (*map-spmf f p*) = *map-spmf f* (*mk-lossless p*)
by(*simp add*: *mk-lossless-def map-scale-spmf*)

lemma *spmf-mk-lossless* [*simp*]: *spmf* (*mk-lossless p*) *x* = *spmf p x* / *weight-spmf p*
by(*simp add*: *mk-lossless-def spmf-scale-spmf inverse-eq-divide max-def*)

lemma *set-spmf-mk-lossless* [*simp*]: *set-spmf* (*mk-lossless p*) = *set-spmf p*
by(*simp add*: *mk-lossless-def set-scale-spmf measure-spmf-zero-iff zero-less-measure-iff*)

lemma *mk-lossless-lossless* [*simp*]: *lossless-spmf p* $\Longrightarrow$ *mk-lossless p* = *p*
by(*simp add*: *mk-lossless-def lossless-weight-spmfD*)

lemma *mk-lossless-eq-return-pmf-None* [*simp*]: *mk-lossless p* = *return-pmf None* $\longleftrightarrow$ *p* = *return-pmf None*
proof −
have *aux*: *weight-spmf p* = *0* $\Longrightarrow$ *spmf p i* = *0* **for** *i*
by(*rule antisym*, *rule order-trans*[*OF spmf-le-weight*]) (*auto intro*!: *order-trans*[*OF spmf-le-weight*])

have[*simp*]: *spmf* (*scale-spmf* (*inverse* (*weight-spmf p*)) *p*) = *spmf* (*return-pmf None*) $\Longrightarrow$ *spmf p i* = *0* **for** *i*
by(*drule fun-cong*[**where** *x*=*i*]) (*auto simp add*: *aux spmf-scale-spmf max-def*)

show *?thesis* **by**(*auto simp add*: *mk-lossless-def intro*: *spmf-eqI*)
qed

lemma *return-pmf-None-eq-mk-lossless* [*simp*]: *return-pmf None* = *mk-lossless p* $\longleftrightarrow$ *p* = *return-pmf None*
by(*metis mk-lossless-eq-return-pmf-None*)

lemma *mk-lossless-spmf-of-set* [*simp*]: *mk-lossless* (*spmf-of-set A*) = *spmf-of-set A*
by(*simp add*: *spmf-of-set-def del*: *spmf-of-pmf-pmf-of-set*)

lemma *weight-mk-lossless*: *weight-spmf* (*mk-lossless p*) = (*if p* = *return-pmf None then 0 else 1*)
by(*simp add*: *mk-lossless-def weight-scale-spmf min-def max-def inverse-eq-divide weight-spmf-eq-0*)

lemma *mk-lossless-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-spmf A* ===> *rel-spmf A*) *mk-lossless mk-lossless*
by(*simp add*: *mk-lossless-def rel-fun-def rel-spmf-weightD rel-spmf-scaleI*)

lemma *rel-spmf-mk-losslessI*:
rel-spmf A p q $\Longrightarrow$ *rel-spmf A* (*mk-lossless p*) (*mk-lossless q*)
by(*rule mk-lossless-parametric*[*THEN rel-funD*])

lemma *rel-spmf-restrict-spmfI*:
rel-spmf (*λx y*. (*x* ∈ *A* ∧ *y* ∈ *B* ∧ *R x y*) ∨ *x* ∉ *A* ∧ *y* ∉ *B*) *p q*
$\Longrightarrow$ *rel-spmf R* (*restrict-spmf p A*) (*restrict-spmf q B*)
by(*auto simp add*: *restrict-spmf-def pmf.rel-map elim*!: *option.rel-cases pmf.rel-mono-strong*)

lemma *cond-spmf-alt*: *cond-spmf p A* = *mk-lossless* (*restrict-spmf p A*)
proof(*cases set-spmf p* ∩ *A* = {})
case *True*
then show *?thesis* **by**(*simp add*: *cond-spmf-def measure-spmf-zero-iff*)
next
case *False*

show *?thesis*
by(*rule spmf-eqI*)(*simp add*: *False cond-spmf-def pmf-cond set-pmf-Int-Some image-iff measure-measure-spmf-conv-measure-pmf* [*symmetric*] *spmf-scale-spmf max-def inverse-eq-divide*)
qed

lemma *cond-spmf-bind*:
cond-spmf (*bind-spmf p f*) *A* = *mk-lossless* (*p* ⤜ (λ*x*. *f x* ↾ *A*))
by(*simp add*: *cond-spmf-alt restrict-bind-spmf scale-bind-spmf*)

lemma *cond-spmf-UNIV* [*simp*]: *cond-spmf p UNIV* = *mk-lossless p*
by(*clarsimp simp add*: *cond-spmf-alt*)

lemma *cond-pmf-singleton*:
cond-pmf p A = *return-pmf x* **if** *set-pmf p* ∩ *A* = {*x*}
proof −
have[*simp*]: *set-pmf p* ∩ *A* = {*x*} ⟹ *x* ∈ *A* ⟹ *measure-pmf.prob p A* = *pmf p x*
by(*auto simp add*: *measure-pmf-single*[*symmetric*] *AE-measure-pmf-iff intro*!: *measure-pmf.finite-measure-eq-AE*)

have *pmf* (*cond-pmf p A*) *i* = *pmf* (*return-pmf x*) *i* **for** *i*
using *that* **by**(*auto simp add*: *pmf-cond measure-pmf-zero-iff pmf-eq-0-set-pmf split*: *split-indicator*)

then show *?thesis* **by**(*rule pmf-eqI*)
qed

definition *cond-spmf-fst* :: ('*a* × '*b*) *spmf* ⇒ '*a* ⇒ '*b spmf* **where**
cond-spmf-fst p a = *map-spmf snd* (*cond-spmf p* ({*a*} × *UNIV*))

lemma *cond-spmf-fst-return-spmf* [*simp*]:
cond-spmf-fst (*return-spmf* (*x*, *y*)) *x* = *return-spmf y*
by(*simp add*: *cond-spmf-fst-def*)

lemma *cond-spmf-fst-map-Pair* [*simp*]: *cond-spmf-fst* (*map-spmf* (*Pair x*) *p*) *x* = *mk-lossless p*
by(*clarsimp simp add*: *cond-spmf-fst-def spmf.map-comp o-def*)

lemma *cond-spmf-fst-map-Pair'* [*simp*]: *cond-spmf-fst* (*map-spmf* (λ*y*. (*x*, *f y*)) *p*) *x* = *map-spmf f* (*mk-lossless p*)
by(*subst spmf.map-comp*[**where** *f*=*Pair x*, *symmetric*, *unfolded o-def*]) *simp*

lemma *cond-spmf-fst-eq-return-None* [*simp*]: *cond-spmf-fst p x* = *return-pmf None* ⟷ *x* ∉ *fst* ‘ *set-spmf p*
by(*auto 4 4 simp add*: *cond-spmf-fst-def map-pmf-eq-return-pmf-iff in-set-spmf*[*symmetric*] *dest*: *bspec*[**where** *x*=*Some* -] *intro*: *ccontr rev-image-eqI*)

lemma *cond-spmf-fst-map-Pair1*:
cond-spmf-fst (*map-spmf* (λ*x*. (*f x*, *g x*)) *p*) (*f x*) = *return-spmf* (*g* (*inv-into* (*set-spmf p*) *f* (*f x*)))
if *x* ∈ *set-spmf p inj-on f* (*set-spmf p*)
proof −
let *?foo*=λ*y*. *map-option* (λ*x*. (*f x*, *g x*)) −‘ *Some* ‘ ({*f y*} × *UNIV*)
have[*simp*]: *y* ∈ *set-spmf p* ⟹ *f x* = *f y* ⟹ *set-pmf p* ∩ (*?foo y*) ≠ {} **for** *y*
by(*auto simp add*: *vimage-def image-def in-set-spmf*)

have[*simp*]: *y* ∈ *set-spmf p* ⟹ *f x* = *f y* ⟹ *map-spmf snd* (*map-spmf* (λ*x*. (*f x*, *g x*)) (*cond-pmf p* (*?foo y*))) = *return-spmf* (*g x*) **for** *y*
using *that* **by**(*subst cond-pmf-singleton*[**where** *x*=*Some x*]) (*auto simp add*: *in-set-spmf elim*: *inj-onD*)

show *?thesis*
using *that*
by(*auto simp add*: *cond-spmf-fst-def cond-spmf-def*)
(*erule notE*, *subst cond-map-pmf*, *simp-all*)
qed

lemma *lossless-cond-spmf-fst* [*simp*]: *lossless-spmf* (*cond-spmf-fst p x*) ⟷ *x* ∈ *fst* ‘ *set-spmf p*
by(*auto simp add*: *cond-spmf-fst-def intro*: *rev-image-eqI*)

lemma *cond-spmf-fst-inverse*:
bind-spmf (*map-spmf fst p*) (λ*x*. *map-spmf* (*Pair x*) (*cond-spmf-fst p x*)) = *p*
(**is** *?lhs* = *?rhs*)
proof(*rule spmf-eqI*)
fix *i* :: *'a* × *'b*
have ∗: ({*x*} × *UNIV* ∩ (*Pair x* ∘ *snd*) −‘ {*i*}) = (*if x* = *fst i then* {*i*} *else* {}) **for** *x* **by**(*cases i*)*auto*
have *spmf ?lhs i* = *LINT x*|*measure-spmf* (*map-spmf fst p*). *spmf* (*map-spmf* (*Pair x* ∘ *snd*) (*cond-spmf p* ({*x*} × *UNIV*))) *i*
by(*auto simp add*: *spmf-bind spmf.map-comp*[*symmetric*] *cond-spmf-fst-def intro*!: *integral-cong-AE*)
also have ... = *LINT x*|*measure-spmf* (*map-spmf fst p*). *measure* (*measure-spmf* (*cond-spmf p* ({*x*} × *UNIV*))) ((*Pair x* ∘ *snd*) −‘ {*i*})
by(*rule integral-cong-AE*)(*auto simp add*: *spmf-map*)
also have ... = *LINT x*|*measure-spmf* (*map-spmf fst p*). *measure* (*measure-spmf p*) ({*x*} × *UNIV* ∩ (*Pair x* ∘ *snd*) −‘ {*i*}) /
measure (*measure-spmf p*) ({*x*} × *UNIV*)
by(*rule integral-cong-AE*; *clarsimp simp add*: *measure-cond-spmf*)
also have ... = *spmf* (*map-spmf fst p*) (*fst i*) ∗ *spmf p i* / *measure* (*measure-spmf p*) ({*fst i*} × *UNIV*)
by(*simp add*: ∗ *if-distrib*[**where** *f*=*measure* (*measure-spmf* -)] *cong*: *if-cong*)
(*subst integral-measure-spmf*[**where** *A*={*fst i*}]; *auto split*: *if-split-asm simp add*: *spmf-conv-measure-spmf*)
also have ... = *spmf p i*
by(*clarsimp simp add*: *spmf-map vimage-fst*)(*metis* (*no-types*, *lifting*) *Int-insert-left-if1*

in-set-spmf-iff-spmf insertI1 insert-UNIV insert-absorb insert-not-empty measure-spmf-zero-iff mem-Sigma-iff prod.collapse)
finally show *spmf ?lhs i = spmf ?rhs i* **.**
qed

fun *rel-witness-generat* :: ('*a*, '*c*, '*e*) *generat* × ('*b*, '*d*, '*f*) *generat* ⇒ ('*a* × '*b*, '*c* × '*d*, '*e* × '*f*) *generat* **where**
rel-witness-generat (*Pure x*, *Pure y*) = *Pure* (*x*, *y*)
| *rel-witness-generat* (*IO out c*, *IO out' c'*) = *IO* (*out*, *out'*) (*c*, *c'*)

lemma *rel-witness-generat*:
assumes *rel-generat A C R x y*
shows *pures-rel-witness-generat*: *generat-pures* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*a*, *b*). *A a b*}
and *outs-rel-witness-generat*: *generat-outs* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*c*, *d*). *C c d*}
and *conts-rel-witness-generat*: *generat-conts* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*e*, *f*). *R e f*}
and *map1-rel-witness-generat*: *map-generat fst fst fst* (*rel-witness-generat* (*x*, *y*)) = *x*
and *map2-rel-witness-generat*: *map-generat snd snd snd* (*rel-witness-generat* (*x*, *y*)) = *y*
using *assms* **by**(*cases*; *simp*; *fail*)+

lemmas *set-rel-witness-generat* = *pures-rel-witness-generat outs-rel-witness-generat conts-rel-witness-generat*

lemma *rel-witness-generat1*:
assumes *rel-generat A C R x y*
shows *rel-generat* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) (λ*c* (*c'*, *d*). *c* = *c'* ∧ *C c' d*) (λ*r* (*r'*, *s*). *r* = *r'* ∧ *R r' s*) *x* (*rel-witness-generat* (*x*, *y*))
using *map1-rel-witness-generat*[*OF assms*, *symmetric*]
unfolding *generat.rel-eq*[*symmetric*] *generat.rel-map*
by(*rule generat.rel-mono-strong*)(*auto dest*: *set-rel-witness-generat*[*OF assms*, *THEN subsetD*])

lemma *rel-witness-generat2*:
assumes *rel-generat A C R x y*
shows *rel-generat* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (λ(*c*, *d'*) *d*. *d* = *d'* ∧ *C c d'*) (λ(*r*, *s'*) *s*. *s* = *s'* ∧ *R r s'*) (*rel-witness-generat* (*x*, *y*)) *y*
using *map2-rel-witness-generat*[*OF assms*]
unfolding *generat.rel-eq*[*symmetric*] *generat.rel-map*
by(*rule generat.rel-mono-strong*)(*auto dest*: *set-rel-witness-generat*[*OF assms*, *THEN subsetD*])

lemma *rel-gpv''-map-gpv1*:
rel-gpv'' A C R (map-gpv f g gpv) gpv' = rel-gpv'' (λa. A (f a)) (λc. C (g c)) R gpv gpv' (**is** *?lhs = ?rhs*)
proof
show *?rhs* **if** *?lhs* **using** *that*
apply(*coinduction arbitrary*: *gpv gpv'*)
apply(*drule rel-gpv''D*)
apply(*simp add*: *gpv.map-sel spmf-rel-map*)
apply(*erule rel-spmf-mono*)
by(*auto simp add*: *generat.rel-map rel-fun-comp elim*!: *generat.rel-mono-strong rel-fun-mono*)
show *?lhs* **if** *?rhs* **using** *that*
apply(*coinduction arbitrary*: *gpv gpv'*)
apply(*drule rel-gpv''D*)
apply(*simp add*: *gpv.map-sel spmf-rel-map*)
apply(*erule rel-spmf-mono*)
by(*auto simp add*: *generat.rel-map rel-fun-comp elim*!: *generat.rel-mono-strong rel-fun-mono*)
qed

lemma *rel-gpv''-map-gpv2*:
rel-gpv'' A C R gpv (map-gpv f g gpv') = rel-gpv'' (λa b. A a (f b)) (λc d. C c (g d)) R gpv gpv'
using *rel-gpv''-map-gpv1*[*of conversep A conversep C conversep R f g gpv' gpv*]
apply(*rewrite* **in** ⌑ = - *conversep-iff*[*symmetric*])
apply(*rewrite* **in** - = ⌑ *conversep-iff*[*symmetric*])
apply(*simp only*: *rel-gpv''-conversep*)
apply(*simp only*: *rel-gpv''-conversep*[*symmetric*])
apply(*simp only*: *conversep-iff*[*abs-def*])
done

lemmas *rel-gpv''-map-gpv = rel-gpv''-map-gpv1*[*abs-def*] *rel-gpv''-map-gpv2*

lemma *rel-gpv''-map-gpv'* [*simp*]:
shows ⋀*f g h gpv*. *NO-MATCH id f* ∨ *NO-MATCH id g*
⟹ *rel-gpv'' A C R (map-gpv' f g h gpv) = rel-gpv'' (λa. A (f a)) (λc. C (g c)) R (map-gpv' id id h gpv)*
and ⋀*f g h gpv gpv'*. *NO-MATCH id f* ∨ *NO-MATCH id g*
⟹ *rel-gpv'' A C R gpv (map-gpv' f g h gpv') = rel-gpv'' (λa b. A a (f b)) (λc d. C c (g d)) R gpv (map-gpv' id id h gpv')*
proof (*goal-cases*)
case (*1 f g h gpv*)
then show *?case* **using** *map-gpv'-comp*[*of f g id id id h gpv*, *symmetric*] **by**(*simp add*: *rel-gpv''-map-gpv*[*unfolded map-gpv-conv-map-gpv'*])
next
case (*2 f g h gpv gpv'*)
then show *?case* **using** *map-gpv'-comp*[*of f g id id id h gpv'*, *symmetric*] **by**(*simp add*: *rel-gpv''-map-gpv*[*unfolded map-gpv-conv-map-gpv'*])

qed

lemmas *rel-gpv-map-gpv'* = *rel-gpv''-map-gpv'*[**where** *R*=(=), *folded rel-gpv-conv-rel-gpv''*]

definition *rel-witness-gpv* :: ('*a* ⇒ '*d* ⇒ *bool*) ⇒ ('*b* ⇒ '*e* ⇒ *bool*) ⇒ ('*c* ⇒ '*g* ⇒ *bool*) ⇒ ('*g* ⇒ '*f* ⇒ *bool*) ⇒ ('*a*, '*b*, '*c*) *gpv* × ('*d*, '*e*, '*f*) *gpv* ⇒ ('*a* × '*d*, '*b* × '*e*, '*g*) *gpv* **where**
rel-witness-gpv A C R R' = *corec-gpv* (
map-spmf (*map-generat id id* (λ(*rpv*, *rpv'*). (*Inr* ∘ *rel-witness-fun R R'* (*rpv*, *rpv'*))) ∘ *rel-witness-generat*) ∘
rel-witness-spmf (*rel-generat A C* (*rel-fun* (*R OO R'*) (*rel-gpv'' A C* (*R OO R'*)))) ∘ *map-prod the-gpv the-gpv*)

lemma *rel-witness-gpv-sel* [*simp*]:
the-gpv (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) =
map-spmf (*map-generat id id* (λ(*rpv*, *rpv'*). (*rel-witness-gpv A C R R'* ∘ *rel-witness-fun R R'* (*rpv*, *rpv'*))) ∘ *rel-witness-generat*)
(*rel-witness-spmf* (*rel-generat A C* (*rel-fun* (*R OO R'*) (*rel-gpv'' A C* (*R OO R'*)))) (*the-gpv gpv*, *the-gpv gpv'*))
unfolding *rel-witness-gpv-def*
by(*auto simp add*: *spmf.map-comp generat.map-comp o-def intro*!: *map-spmf-cong generat.map-cong*)

lemma assumes *rel-gpv'' A C* (*R OO R'*) *gpv gpv'*
and *R*: *left-unique R right-total R*
and *R'*: *right-unique R' left-total R'*
shows *rel-witness-gpv1*: *rel-gpv''* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) (λ*c* (*c'*, *d*). *c* = *c'* ∧ *C c' d*) *R gpv* (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) (**is** *?thesis1*)
and *rel-witness-gpv2*: *rel-gpv''* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (λ(*c*, *d'*) *d*. *d* = *d'* ∧ *C c d'*) *R'* (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) *gpv'* (**is** *?thesis2*)
proof −
show *?thesis1* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *gpv gpv'*)
case *rel-gpv''*
from *this*[*THEN rel-gpv''D*] **show** *?case*
by(*auto simp add*: *spmf-rel-map generat.rel-map rel-fun-comp elim*!: *rel-fun-mono*[*OF rel-witness-fun1*[*OF - R R'*]]
rel-spmf-mono[*OF rel-witness-spmf1*] *generat.rel-mono*[*THEN predicate2D*, *rotated* −*1*, *OF rel-witness-generat1*])
qed
show *?thesis2* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *gpv gpv'*)
case *rel-gpv''*
from *this*[*THEN rel-gpv''D*] **show** *?case*
by(*simp add*: *spmf-rel-map*)
(*erule rel-spmf-mono*[*OF rel-witness-spmf2*]
, *auto simp add*: *generat.rel-map rel-fun-comp elim*!: *rel-fun-mono*[*OF rel-witness-fun2*[*OF - R R'*]]
generat.rel-mono[*THEN predicate2D*, *rotated* −*1*, *OF rel-witness-generat2*])

qed
qed

lemma *rel-gpv''-neg-distr*:
assumes *R*: *left-unique R right-total R*
and *R'*: *right-unique R' left-total R'*
shows *rel-gpv'' (A OO A') (C OO C') (R OO R') ≤ rel-gpv'' A C R OO rel-gpv'' A' C' R'*
proof(*rule predicate2I relcomppI*)+
fix *gpv gpv''*
assume ∗: *rel-gpv'' (A OO A') (C OO C') (R OO R') gpv gpv''*
let *?gpv' = map-gpv (relcompp-witness A A') (relcompp-witness C C') (rel-witness-gpv (A OO A') (C OO C') R R' (gpv, gpv''))*
show *rel-gpv'' A C R gpv ?gpv'* **using** *rel-witness-gpv1*[*OF* ∗ *R R'*] **unfolding** *rel-gpv''-map-gpv*
by(*rule rel-gpv''-mono*[*THEN predicate2D*, *rotated −1*]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)
show *rel-gpv'' A' C' R' ?gpv' gpv''* **using** *rel-witness-gpv2*[*OF* ∗ *R R'*] **unfolding** *rel-gpv''-map-gpv*
by(*rule rel-gpv''-mono*[*THEN predicate2D*, *rotated −1*]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)
qed

lemma *rel-gpv''-mono'* [*mono*]:
assumes ⋀*x y*. *A x y* ⟶ *A' x y*
and ⋀*x y*. *C x y* ⟶ *C' x y*
and ⋀*x y*. *R' x y* ⟶ *R x y*
shows *rel-gpv'' A C R gpv gpv'* ⟶ *rel-gpv'' A' C' R' gpv gpv'*
using *rel-gpv''-mono*[*of A A' C C' R' R*] *assms* **by**(*blast*)

context includes *I.lifting* **begin**

lift-definition *I-uniform* :: *'out set* ⇒ *'in set* ⇒ (*'out*, *'in*) *I* **is** λ*A B x*. *if x* ∈ *A then B else* {} **.**

lemma *outs-I-uniform* [*simp*]: *outs-I* (*I-uniform A B*) = (*if B* = {} *then* {} *else A*)
by *transfer simp*

lemma *responses-I-uniform* [*simp*]: *responses-I* (*I-uniform A B*) *x* = (*if x* ∈ *A then B else* {})
by *transfer simp*

lemma *I-uniform-UNIV* [*simp*]: *I-uniform UNIV UNIV* = *I-full*
by *transfer simp*

lifting-update *I.lifting*
lifting-forget *I.lifting*

end

lemma $\mathcal{I}$-*eqI*: $[\![$ *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$; $\bigwedge x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}' \Longrightarrow$ *responses-*$\mathcal{I}$ $\mathcal{I}$ x = *responses-*$\mathcal{I}$ $\mathcal{I}'$ x $]\!] \Longrightarrow \mathcal{I} = \mathcal{I}'$
including $\mathcal{I}$.*lifting* **by** *transfer auto*

instantiation $\mathcal{I}$ :: (*type*, *type*) *order* **begin**

definition *less-eq-*$\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*a*, '*b*) $\mathcal{I}$ ⇒ *bool*
where *le-*$\mathcal{I}$*-def*: *less-eq-*$\mathcal{I}$ $\mathcal{I}$ $\mathcal{I}' \longleftrightarrow$ *outs-*$\mathcal{I}$ $\mathcal{I} \subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}' \wedge (\forall x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$. *responses-*$\mathcal{I}$ $\mathcal{I}'$ $x \subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ x)

definition *less-*$\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*a*, '*b*) $\mathcal{I}$ ⇒ *bool*
where *less-*$\mathcal{I}$ = *mk-less* (≤)

instance
proof
show $\mathcal{I} < \mathcal{I}' \longleftrightarrow \mathcal{I} \leq \mathcal{I}' \wedge \neg\, \mathcal{I}' \leq \mathcal{I}$ **for** $\mathcal{I}$ $\mathcal{I}'$ :: ('*a*, '*b*) $\mathcal{I}$ **by**(*simp add*: *less-*$\mathcal{I}$*-def mk-less-def*)
show $\mathcal{I} \leq \mathcal{I}$ **for** $\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ **by**(*simp add*: *le-*$\mathcal{I}$*-def*)
show $\mathcal{I} \leq \mathcal{I}''$ **if** $\mathcal{I} \leq \mathcal{I}'$ $\mathcal{I}' \leq \mathcal{I}''$ **for** $\mathcal{I}$ $\mathcal{I}'$ $\mathcal{I}''$ :: ('*a*, '*b*) $\mathcal{I}$ **using** *that*
by(*fastforce simp add*: *le-*$\mathcal{I}$*-def*)
show $\mathcal{I} = \mathcal{I}'$ **if** $\mathcal{I} \leq \mathcal{I}'$ $\mathcal{I}' \leq \mathcal{I}$ **for** $\mathcal{I}$ $\mathcal{I}'$ :: ('*a*, '*b*) $\mathcal{I}$ **using** *that*
by(*auto simp add*: *le-*$\mathcal{I}$*-def intro*!: $\mathcal{I}$*-eqI*)
qed
end

instantiation $\mathcal{I}$ :: (*type*, *type*) *order-bot* **begin**
definition *bot-*$\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ **where** *bot-*$\mathcal{I}$ = $\mathcal{I}$*-uniform* {} *UNIV*
instance by *standard*(*auto simp add*: *bot-*$\mathcal{I}$*-def le-*$\mathcal{I}$*-def*)
end

lemma *outs-*$\mathcal{I}$*-bot* [*simp*]: *outs-*$\mathcal{I}$ *bot* = {}
by(*simp add*: *bot-*$\mathcal{I}$*-def*)

lemma *respones-*$\mathcal{I}$*-bot* [*simp*]: *responses-*$\mathcal{I}$ *bot* x = {}
by(*simp add*: *bot-*$\mathcal{I}$*-def*)

lemma *outs-*$\mathcal{I}$*-mono*: $\mathcal{I} \leq \mathcal{I}' \Longrightarrow$ *outs-*$\mathcal{I}$ $\mathcal{I} \subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}'$
by(*simp add*: *le-*$\mathcal{I}$*-def*)

lemma *responses-*$\mathcal{I}$*-mono*: $[\![$ $\mathcal{I} \leq \mathcal{I}'$; $x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!] \Longrightarrow$ *responses-*$\mathcal{I}$ $\mathcal{I}'$ $x \subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ x
by(*simp add*: *le-*$\mathcal{I}$*-def*)

lemma $\mathcal{I}$*-uniform-empty* [*simp*]: $\mathcal{I}$*-uniform* {} A = *bot*
unfolding *bot-*$\mathcal{I}$*-def* **including** $\mathcal{I}$.*lifting* **by** *transfer simp*

lemma *WT-gpv-*$\mathcal{I}$*-mono*: $[\![$ $\mathcal{I} \vdash g$ *gpv* $\surd$; $\mathcal{I} \leq \mathcal{I}'$ $]\!] \Longrightarrow \mathcal{I}' \vdash g$ *gpv* $\surd$

by(*erule WT-gpv-mono*; *rule outs-$\mathcal{I}$-mono responses-$\mathcal{I}$-mono*)

lemma *results-gpv-mono*:
assumes *le*: $\mathcal{I}' \leq \mathcal{I}$ **and** *WT*: $\mathcal{I}' \vdash g$ *gpv* $\surd$
shows *results-gpv* $\mathcal{I}$ *gpv* $\subseteq$ *results-gpv* $\mathcal{I}'$ *gpv*
proof(*rule subsetI*, *goal-cases*)
case (*1 x*)
show *?case* **using** *1 WT* **by**(*induction*)(*auto 4 3 intro*: *results-gpv.intros responses-$\mathcal{I}$-mono*[*OF le*, *THEN subsetD*] *intro*: *WT-gpvD*)
qed

lemma *$\mathcal{I}$-uniform-mono*:
$\mathcal{I}$-uniform A B $\leq$ *$\mathcal{I}$-uniform C D* **if** $A \subseteq C$ $D \subseteq B$ $D = \{\} \longrightarrow B = \{\}$
unfolding *le-$\mathcal{I}$-def* **using** *that* **by** *auto*

context begin
qualified inductive *outsp-gpv* :: ('*out*, '*in*) $\mathcal{I} \Rightarrow$ '*out* $\Rightarrow$ ('*a*, '*out*, '*in*) *gpv* $\Rightarrow$ *bool*
for $\mathcal{I}$ *x* **where**
IO: *IO x c* $\in$ *set-spmf* (*the-gpv gpv*) $\Longrightarrow$ *outsp-gpv* $\mathcal{I}$ *x gpv*
| *Cont*: $[\![$ *IO out rpv* $\in$ *set-spmf* (*the-gpv gpv*); *input* $\in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *out*; *outsp-gpv* $\mathcal{I}$ *x* (*rpv input*) $]\!]$
$\Longrightarrow$ *outsp-gpv* $\mathcal{I}$ *x gpv*

definition *outs-gpv* :: ('*out*, '*in*) $\mathcal{I} \Rightarrow$ ('*a*, '*out*, '*in*) *gpv* $\Rightarrow$ '*out set*
where *outs-gpv* $\mathcal{I}$ *gpv* $\equiv \{x.$ *outsp-gpv* $\mathcal{I}$ *x gpv*$\}$

lemma *outsp-gpv-outs-gpv-eq* [*pred-set-conv*]: *outsp-gpv* $\mathcal{I}$ *x* $= (\lambda gpv.\ x \in$ *outs-gpv* $\mathcal{I}$ *gpv*)
by(*simp add*: *outs-gpv-def*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path outs-gpv*)›

lemmas *intros* [*intro?*] = *outsp-gpv.intros*[*to-set*]
and *IO* = *IO*[*to-set*]
and *Cont* = *Cont*[*to-set*]
and *induct* [*consumes 1*, *case-names IO Cont*, *induct set*: *outs-gpv*] = *outsp-gpv.induct*[*to-set*]
and *cases* [*consumes 1*, *case-names IO Cont*, *cases set*: *outs-gpv*] = *outsp-gpv.cases*[*to-set*]
and *simps* = *outsp-gpv.simps*[*to-set*]
end

inductive-simps *outs-gpv-GPV* [*to-set*, *simp*]: *outsp-gpv* $\mathcal{I}$ *x* (*GPV gpv*)

end

lemma *outs-gpv-Done* [*iff*]: *outs-gpv* $\mathcal{I}$ (*Done x*) $= \{\}$
by(*auto simp add*: *Done.ctr*)

lemma *outs-gpv-Fail* [*iff*]: *outs-gpv* $\mathcal{I}$ *Fail* = {}
by(*auto simp add*: *Fail-def*)

lemma *outs-gpv-Pause* [*simp*]:
outs-gpv $\mathcal{I}$ (*Pause out c*) = *insert out* ($\bigcup$ *input*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *outs-gpv* $\mathcal{I}$ (*c input*))
by(*auto simp add*: *Pause.ctr*)

lemma *outs-gpv-lift-spmf* [*iff*]: *outs-gpv* $\mathcal{I}$ (*lift-spmf p*) = {}
by(*auto simp add*: *lift-spmf.ctr*)

lemma *outs-gpv-assert-gpv* [*simp*]: *outs-gpv* $\mathcal{I}$ (*assert-gpv b*) = {}
by(*cases b*)*auto*

lemma *outs-gpv-bind-gpv* [*simp*]:
outs-gpv $\mathcal{I}$ (*gpv* $\gg\!=$ *f*) = *outs-gpv* $\mathcal{I}$ *gpv* ∪ ($\bigcup$ *x*∈*results-gpv* $\mathcal{I}$ *gpv*. *outs-gpv* $\mathcal{I}$ (*f x*))
(**is** *?lhs* = *?rhs*)
proof(*intro Set.set-eqI iffI*)
fix *x*
assume *x* ∈ *?lhs*
then show *x* ∈ *?rhs*
proof(*induction gpv'*≡*gpv* $\gg\!=$ *f arbitrary*: *gpv*)
case *IO* **thus** *?case*
proof(*clarsimp split*: *if-split-asm elim*!: *is-PureE not-is-PureE*, *goal-cases*)
case (*1 generat*)
then show *?case* **by**(*cases generat*)(*auto intro*: *results-gpv.Pure outs-gpv.intros*)
qed
next
case (*Cont out rpv input*)
thus *?case*
proof(*clarsimp split*: *if-split-asm*, *goal-cases*)
case (*1 generat*)
then show *?case* **by**(*cases generat*)(*auto 4 3 split*: *if-split-asm intro*: *results-gpv.intros outs-gpv.intros*)
qed
qed
next
fix *x*
assume *x* ∈ *?rhs*
then consider (*out*) *x* ∈ *outs-gpv* $\mathcal{I}$ *gpv* | (*result*) *y* **where** *y* ∈ *results-gpv* $\mathcal{I}$ *gpv x* ∈ *outs-gpv* $\mathcal{I}$ (*f y*) **by** *auto*
then show *x* ∈ *?lhs*
proof *cases*
case *out* **then show** *?thesis*
by(*induction*) (*auto 4 4 intro*: *outs-gpv.IO outs-gpv.Cont rev-bexI*)
next
case *result* **then show** *?thesis*
by *induction* ((*erule outs-gpv.cases* | *rule outs-gpv.Cont*),

auto 4 4 intro: *outs-gpv.intros rev-bexI elim*: *outs-gpv.cases*)+
qed
qed

lemma *outs-gpv-ℐ-full*: *outs-gpv ℐ-full* = *outs′-gpv*
proof(*intro ext Set.set-eqI iffI*)
show *x* ∈ *outs′-gpv gpv* **if** *x* ∈ *outs-gpv ℐ-full gpv* **for** *x gpv*
using *that* **by** *induction*(*auto intro*: *outs′-gpvI*)
show *x* ∈ *outs-gpv ℐ-full gpv* **if** *x* ∈ *outs′-gpv gpv* **for** *x gpv*
using *that* **by** *induction*(*auto intro*: *outs-gpv.intros elim*!: *generat.set-cases*)
qed

lemma *outs′-bind-gpv* [*simp*]:
outs′-gpv (*bind-gpv gpv f*) = *outs′-gpv gpv* ∪ (⋃ *x*∈*results′-gpv gpv*. *outs′-gpv* (*f x*))
unfolding *outs-gpv-ℐ-full*[*symmetric*] *results-gpv-ℐ-full*[*symmetric*] **by** *simp*

lemma *outs-gpv-map-gpv-id* [*simp*]: *outs-gpv ℐ* (*map-gpv f id gpv*) = *outs-gpv ℐ gpv*
by(*auto simp add*: *map-gpv-conv-bind id-def*)

lemma *outs-gpv-map-gpv-id′* [*simp*]: *outs-gpv ℐ* (*map-gpv f* (λ*x*. *x*) *gpv*) = *outs-gpv ℐ gpv*
by(*auto simp add*: *map-gpv-conv-bind id-def*)

lemma *outs′-gpv-bind-option* [*simp*]:
outs′-gpv (*monad.bind-option Fail x f*) = (⋃ *y*∈*set-option x*. *outs′-gpv* (*f y*))
by(*cases x*) *simp-all*

lemma *WT-gpv-outs-gpv*:
assumes *ℐ* ⊢*g gpv* √
shows *outs-gpv ℐ gpv* ⊆ *outs-ℐ ℐ*
proof
show *x* ∈ *outs-ℐ ℐ* **if** *x* ∈ *outs-gpv ℐ gpv* **for** *x* **using** *that assms*
by(*induction*)(*blast intro*: *WT-gpv-OutD WT-gpv-ContD*)+
qed

context includes *ℐ.lifting* **begin**

lift-definition *map-ℐ* :: (*′out′* ⇒ *′out*) ⇒ (*′in* ⇒ *′in′*) ⇒ (*′out*, *′in*) *ℐ* ⇒ (*′out′*, *′in′*) *ℐ*
is λ*f g resp x*. *g* ‘ *resp* (*f x*) **.**

lemma *outs-ℐ-map-ℐ* [*simp*]:
outs-ℐ (*map-ℐ f g ℐ*) = *f* −‘ *outs-ℐ ℐ*
by *transfer simp*

lemma *responses-ℐ-map-ℐ* [*simp*]:
responses-ℐ (*map-ℐ f g ℐ*) *x* = *g* ‘ *responses-ℐ ℐ* (*f x*)

by *transfer simp*

lemma *map-ℐ-ℐ-uniform* [*simp*]:
map-ℐ f g (*ℐ-uniform A B*) = *ℐ-uniform* (*f* −‘ *A*) (*g* ‘ *B*)
by *transfer*(*auto simp add*: *fun-eq-iff*)

lemma *map-ℐ-id* [*simp*]: *map-ℐ id id ℐ* = *ℐ*
by *transfer simp*

lemma *map-ℐ-id0*: *map-ℐ id id* = *id*
by(*simp add*: *fun-eq-iff*)

lemma *map-ℐ-comp* [*simp*]: *map-ℐ f g* (*map-ℐ f′ g′ ℐ*) = *map-ℐ* (*f′* ∘ *f*) (*g* ∘ *g′*) *ℐ*
by *transfer auto*

lemma *map-ℐ-cong*: *map-ℐ f g ℐ* = *map-ℐ f′ g′ ℐ′*
if *ℐ* = *ℐ′* **and** *f*: *f* = *f′* **and** ⋀*x y*. ⟦ *x* ∈ *outs-ℐ ℐ′*; *y* ∈ *responses-ℐ ℐ′ x* ⟧ ⟹ *g y* = *g′ y*
unfolding *that*(*1*,*2*) **using** *that*(*3*−)
by *transfer*(*auto simp add*: *fun-eq-iff intro*!: *image-cong*)

lifting-update *ℐ.lifting*
lifting-forget *ℐ.lifting*
end

functor *map-ℐ* **by**(*simp-all add*: *fun-eq-iff*)

lemma *WT-gpv-map-gpv′*: *ℐ* ⊢*g map-gpv′ f g h gpv* √ **if** *map-ℐ g h ℐ* ⊢*g gpv* √
using *that* **by**(*coinduction arbitrary*: *gpv*)(*auto 4 4 dest*: *WT-gpvD*)

lemma *WT-gpv-map-gpv*: *ℐ* ⊢*g map-gpv f g gpv* √ **if** *map-ℐ g id ℐ* ⊢*g gpv* √
unfolding *map-gpv-conv-map-gpv′* **using** *that* **by**(*rule WT-gpv-map-gpv′*)

lemma *results-gpv-map-gpv′* [*simp*]:
results-gpv ℐ (*map-gpv′ f g h gpv*) = *f* ‘ (*results-gpv* (*map-ℐ g h ℐ*) *gpv*)
proof(*intro Set.set-eqI iffI*; (*elim imageE*; *hypsubst*)*?*)
show *x* ∈ *f* ‘ *results-gpv* (*map-ℐ g h ℐ*) *gpv* **if** *x* ∈ *results-gpv ℐ* (*map-gpv′ f g h gpv*) **for** *x* **using** *that*
by(*induction gpv*′≡*map-gpv′ f g h gpv arbitrary*: *gpv*)(*fastforce intro*: *results-gpv.intros rev-image-eqI*)+
show *f x* ∈ *results-gpv ℐ* (*map-gpv′ f g h gpv*) **if** *x* ∈ *results-gpv* (*map-ℐ g h ℐ*) *gpv* **for** *x* **using** *that*
by(*induction*)(*fastforce intro*: *results-gpv.intros*)+
qed

lemma *map-ℐ-plus-ℐ* [*simp*]:
map-ℐ (*map-sum f1 f2*) (*map-sum g1 g2*) (*ℐ1* ⊕*ℐ ℐ2*) = *map-ℐ f1 g1 ℐ1* ⊕*ℐ map-ℐ f2 g2 ℐ2*

proof(*rule* $\mathcal{I}$*-eqI*[*OF Set.set-eqI*], *goal-cases*)
case (*1 x*)
then show *?case* **by**(*cases x*) *auto*
qed (*auto simp add*: *image-image*)

lemma *le-plus-*$\mathcal{I}$*-iff* [*simp*]:
$\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \leq \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \longleftrightarrow \mathcal{I}1 \leq \mathcal{I}1' \wedge \mathcal{I}2 \leq \mathcal{I}2'$
by(*auto 4 4 simp add*: *le-*$\mathcal{I}$*-def dest*: *bspec*[**where** *x*=*Inl* -] *bspec*[**where** *x*=*Inr* -])

inductive *pred-gpv'* :: ('*a* ⇒ *bool*) ⇒ ('*out* ⇒ *bool*) ⇒ '*in set* ⇒ ('*a*, '*out*, '*in*) *gpv* ⇒ *bool* **for** *P Q X gpv* **where**
pred-gpv' P Q X gpv
if $\bigwedge x.\ x \in$ *results-gpv* ($\mathcal{I}$*-uniform UNIV X*) *gpv* $\Longrightarrow$ *P x* $\bigwedge$*out*. *out* ∈ *outs-gpv* ($\mathcal{I}$*-uniform UNIV X*) *gpv* $\Longrightarrow$ *Q out*

lemma *pred-gpv-conv-pred-gpv'*: *pred-gpv P Q* = *pred-gpv' P Q UNIV*
by(*auto simp add*: *fun-eq-iff pred-gpv-def pred-gpv'.simps results-gpv-*$\mathcal{I}$*-full outs-gpv-*$\mathcal{I}$*-full*)

lemma *rel-gpv''-Grp*: **includes** *lifting-syntax* **shows**
rel-gpv'' (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$ =
BNF-Def.Grp {*x*. *results-gpv* ($\mathcal{I}$*-uniform UNIV* (*range h*)) *x* ⊆ *A* ∧ *outs-gpv* ($\mathcal{I}$*-uniform UNIV* (*range h*)) *x* ⊆ *B*} (*map-gpv' f g h*)
(**is** *?lhs* = *?rhs*)
proof(*intro ext GrpI iffI CollectI conjI subsetI*)
let *?*$\mathcal{I}$ = $\mathcal{I}$*-uniform UNIV* (*range h*)
fix *gpv gpv'*
assume ∗: *?lhs gpv gpv'*
then show *map-gpv' f g h gpv* = *gpv'*
by(*coinduction arbitrary*: *gpv gpv'*)
(*drule rel-gpv''D*
, *auto 4 5 simp add*: *spmf-rel-map generat.rel-map elim*!: *rel-spmf-mono generat.rel-mono-strong GrpE intro*!: *GrpI dest*: *rel-funD*)
show *x* ∈ *A* **if** *x* ∈ *results-gpv ?*$\mathcal{I}$ *gpv* **for** *x* **using** *that* ∗
proof(*induction arbitrary*: *gpv'*)
case (*Pure gpv*)
have *pred-spmf* (*Domainp* (*rel-generat* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) ((*BNF-Def.Grp UNIV h*)$^{-1-1}$ ===> *rel-gpv''* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$))) (*the-gpv gpv*)
using *Pure.prems*[*THEN rel-gpv''D*] **unfolding** *spmf-Domainp-rel*[*symmetric*]
by(*rule DomainPI*)
with *Pure.hyps* **show** *?case* **by**(*simp add*: *generat.Domainp-rel pred-spmf-def pred-generat-def Domainp-Grp*)
next
case (*IO out c gpv input*)
have *pred-spmf* (*Domainp* (*rel-generat* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*)

```
((BNF-Def.Grp UNIV h)^-1-1 ===> rel-gpv'' (BNF-Def.Grp A f) (BNF-Def.Grp
B g) (BNF-Def.Grp UNIV h)^-1-1))) (the-gpv gpv)
      using IO.prems[THEN rel-gpv''D] unfolding spmf-Domainp-rel[symmetric]
by(rule DomainPI)
    with IO.hyps show ?case
     by(auto simp add: generat.Domainp-rel pred-spmf-def pred-generat-def Grp-iff
dest: rel-funD intro: IO.IH dest!: bspec)
  qed
  show x ∈ B if x ∈ outs-gpv ?I gpv for x using that *
  proof(induction arbitrary: gpv')
    case (IO c gpv)
   have pred-spmf (Domainp (rel-generat (BNF-Def.Grp A f) (BNF-Def.Grp B g)
((BNF-Def.Grp UNIV h)^-1-1 ===> rel-gpv'' (BNF-Def.Grp A f) (BNF-Def.Grp
B g) (BNF-Def.Grp UNIV h)^-1-1))) (the-gpv gpv)
      using IO.prems[THEN rel-gpv''D] unfolding spmf-Domainp-rel[symmetric]
by(rule DomainPI)
     with IO.hyps show ?case by(simp add: generat.Domainp-rel pred-spmf-def
pred-generat-def Domainp-Grp)
  next
    case (Cont out rpv gpv input)
   have pred-spmf (Domainp (rel-generat (BNF-Def.Grp A f) (BNF-Def.Grp B g)
((BNF-Def.Grp UNIV h)^-1-1 ===> rel-gpv'' (BNF-Def.Grp A f) (BNF-Def.Grp
B g) (BNF-Def.Grp UNIV h)^-1-1))) (the-gpv gpv)
    using Cont.prems[THEN rel-gpv''D] unfolding spmf-Domainp-rel[symmetric]
by(rule DomainPI)
    with Cont.hyps show ?case
     by(auto simp add: generat.Domainp-rel pred-spmf-def pred-generat-def Grp-iff
dest: rel-funD intro: Cont.IH dest!: bspec)
  qed
next
  fix gpv gpv'
  assume ?rhs gpv gpv'
  then have gpv': gpv' = map-gpv' f g h gpv
    and *: results-gpv (I-uniform UNIV (range h)) gpv ⊆ A outs-gpv (I-uniform
UNIV (range h)) gpv ⊆ B by(auto simp add: Grp-iff)
  show ?lhs gpv gpv' using * unfolding gpv'
    by(coinduction arbitrary: gpv)
     (fastforce simp add: spmf-rel-map generat.rel-map Grp-iff intro!: rel-spmf-reflI
generat.rel-refl-strong rel-funI elim!: generat.set-cases intro: results-gpv.intros outs-gpv.intros)
qed

lemma rel-gpv''-map-gpv'1:
  rel-gpv'' A C (BNF-Def.Grp UNIV h)^-1-1 gpv gpv' ⟹ rel-gpv'' A C (=)
(map-gpv' id id h gpv) gpv'
  apply(coinduction arbitrary: gpv gpv')
  apply(drule rel-gpv''D)
  apply(simp add: spmf-rel-map)
  apply(erule rel-spmf-mono)
  apply(simp add: generat.rel-map)
```

apply(*erule generat.rel-mono-strong*; *simp?*)
apply(*subst map-fun2-id*)
by(*auto simp add*: *rel-fun-comp intro*!: *rel-fun-map-fun1 elim*: *rel-fun-mono*)

lemma *rel-gpv''-map-gpv'2*:
rel-gpv'' A C (*eq-on* (*range h*)) *gpv gpv'* ⟹ *rel-gpv'' A C* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *gpv* (*map-gpv' id id h gpv'*)
apply(*coinduction arbitrary*: *gpv gpv'*)
apply(*drule rel-gpv''D*)
apply(*simp add*: *spmf-rel-map*)
apply(*erule rel-spmf-mono-strong*)
apply(*simp add*: *generat.rel-map*)
apply(*erule generat.rel-mono-strong*; *simp?*)
apply(*subst map-fun-id2-in*)
apply(*rule rel-fun-map-fun2*)
by (*auto simp add*: *rel-fun-comp elim*: *rel-fun-mono*)

context
fixes *A* :: *'a* ⇒ *'d* ⇒ *bool*
and *C* :: *'c* ⇒ *'g* ⇒ *bool*
and *R* :: *'b* ⇒ *'e* ⇒ *bool*
begin

private lemma *f11*: *Pure x* ∈ *set-spmf* (*the-gpv gpv*) ⟹
Domainp (*rel-generat A C* (*rel-fun R* (*rel-gpv'' A C R*))) (*Pure x*) ⟹ *Domainp A x*
by (*auto simp add*: *pred-generat-def elim*:*bspec dest*: *generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])

private lemma *f21*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟹
rel-generat A C (*rel-fun R* (*rel-gpv'' A C R*)) (*IO out c*) *ba* ⟹ *Domainp C out*
by (*auto simp add*: *pred-generat-def elim*:*bspec dest*: *generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])

private lemma *f12*:
assumes *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *input* ∈ *responses-*$\mathcal{I}$ (*$\mathcal{I}$-uniform UNIV* {*x*. *Domainp R x*}) *out*
and *x* ∈ *results-gpv* (*$\mathcal{I}$-uniform UNIV* {*x*. *Domainp R x*}) (*c input*)
and *Domainp* (*rel-gpv'' A C R*) *gpv*
shows *Domainp* (*rel-gpv'' A C R*) (*c input*)
proof −
obtain *b1* **where** *o1*:*rel-gpv'' A C R gpv b1* **using** *assms*(*4*) **by** *clarsimp*
obtain *b2* **where** *o2*:*rel-generat A C* (*rel-fun R* (*rel-gpv'' A C R*)) (*IO out c*) *b2*
using *assms*(*1*) *o1*[*THEN rel-gpv''D*, *THEN spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-spmf-def* **by** − (*drule* (*1*) *bspec*, *auto*)

have *Ball* (*generat-conts* (*IO out c*)) (*Domainp* (*rel-fun R* (*rel-gpv'' A C R*)))

using *o2*[*THEN generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-generat-def* **by** *simp*

with *assms*(*2*) **show** *?thesis*
apply −
apply(*drule bspec*)
apply *simp*
apply *clarify*
apply(*drule Domainp-rel-fun-le*[*THEN predicate1D*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
by *simp*
qed

private lemma *f22*:
assumes *IO out' rpv* ∈ *set-spmf* (*the-gpv gpv*)
and *input* ∈ *responses-*$\mathcal{I}$ (*$\mathcal{I}$-uniform UNIV* {*x*. *Domainp R x*}) *out'*
and *out* ∈ *outs-gpv* (*$\mathcal{I}$-uniform UNIV* {*x*. *Domainp R x*}) (*rpv input*)
and *Domainp* (*rel-gpv'' A C R*) *gpv*
shows *Domainp* (*rel-gpv'' A C R*) (*rpv input*)
proof −
obtain *b1* **where** *o1*:*rel-gpv'' A C R gpv b1* **using** *assms*(*4*) **by** *auto*
obtain *b2* **where** *o2*:*rel-generat A C* (*rel-fun R* (*rel-gpv'' A C R*)) (*IO out' rpv*) *b2*
using *assms*(*1*) *o1*[*THEN rel-gpv''D*, *THEN spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-spmf-def* **by** − (*drule* (*1*) *bspec*, *auto*)

have *Ball* (*generat-conts* (*IO out' rpv*)) (*Domainp* (*rel-fun R* (*rel-gpv'' A C R*)))
using *o2*[*THEN generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-generat-def* **by** *simp*

with *assms*(*2*) **show** *?thesis*
apply −
apply(*drule bspec*)
apply *simp*
apply *clarify*
apply(*drule Domainp-rel-fun-le*[*THEN predicate1D*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
by *simp*
qed

lemma *Domainp-rel-gpv''-le*:
Domainp (*rel-gpv'' A C R*) ≤ *pred-gpv'* (*Domainp A*) (*Domainp C*) {*x*. *Domainp R x*}
proof(*rule predicate1I pred-gpv'.intros*)+

show *Domainp A x* **if** *x* ∈ *results-gpv* (*$\mathcal{I}$-uniform UNIV* {*x*. *Domainp R x*}) *gpv*

Domainp (*rel-gpv''* *A* *C* *R*) *gpv* **for** *x gpv* **using** *that*
proof(*induction*)
case (*Pure gpv*)
then show *?case*
by (*clarify*) (*drule rel-gpv''D*
, *auto simp add*: *f11 pred-spmf-def dest*: *spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
qed (*simp add*: *f12*)
show *Domainp C out* **if** *out* ∈ *outs-gpv* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) *gpv Domainp* (*rel-gpv''* *A* *C* *R*) *gpv* **for** *out gpv* **using** *that*
proof(*induction*)
case (*IO c gpv*)
then show *?case*
by (*clarify*) (*drule rel-gpv''D*
, *auto simp add*: *f21 pred-spmf-def dest*!: *bspec spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
qed (*simp add*: *f22*)
qed

end

lemma *map-gpv'-id12*: *map-gpv'* *f* *g* *h* *gpv* = *map-gpv'* *id* *id* *h* (*map-gpv* *f* *g* *gpv*)
unfolding *map-gpv-conv-map-gpv'* *map-gpv'-comp* **by** *simp*

lemma *rel-gpv''-refl*: ⟦ (=) ≤ *A*; (=) ≤ *C*; *R* ≤ (=) ⟧ ⟹ (=) ≤ *rel-gpv''* *A* *C* *R*
by(*subst rel-gpv''-eq*[*symmetric*])(*rule rel-gpv''-mono*)

context
fixes *A* *A'* :: *'a* ⇒ *'b* ⇒ *bool*
and *C* *C'* :: *'c* ⇒ *'d* ⇒ *bool*
and *R* *R'* :: *'e* ⇒ *'f* ⇒ *bool*

begin

private abbreviation *foo* **where**
foo ≡ (λ *fx fy gpvx gpvy g1 g2*.
∀ *x y*. *x* ∈ *fx* (*ℐ-uniform UNIV* (*Collect* (*Domainp R'*))) *gpvx* ⟶
y ∈ *fy* (*ℐ-uniform UNIV* (*Collect* (*Rangep R'*))) *gpvy* ⟶ *g1 x y* ⟶ *g2 x y*)

private lemma *f1*: *foo results-gpv results-gpv gpv gpv'* *A* *A'* ⟹
x ∈ *set-spmf* (*the-gpv gpv*) ⟹ *y* ∈ *set-spmf* (*the-gpv gpv'*) ⟹
a ∈ *generat-conts x* ⟹ *b* ∈ *generat-conts y* ⟹ *R'* *a'* *α* ⟹ *R'* *β* *b'* ⟹
foo results-gpv results-gpv (*a a'*) (*b b'*) *A* *A'*
by (*fastforce elim*: *generat.set-cases intro*: *results-gpv.IO*)

private lemma *f2*: *foo outs-gpv outs-gpv gpv gpv'* *C* *C'* ⟹

```
    x ∈ set-spmf (the-gpv gpv) ⟹ y ∈ set-spmf (the-gpv gpv') ⟹
    a ∈ generat-conts x ⟹ b ∈ generat-conts y ⟹ R' a' α ⟹ R' β b' ⟹
   foo outs-gpv outs-gpv (a a') (b b') C C'
  by (fastforce elim: generat.set-cases intro: outs-gpv.Cont)

lemma rel-gpv''-mono-strong:
  ⟦ rel-gpv'' A C R gpv gpv';
     ⋀x y. ⟦ x ∈ results-gpv (ℐ-uniform UNIV {x. Domainp R' x}) gpv; y ∈
results-gpv (ℐ-uniform UNIV {x. Rangep R' x}) gpv'; A x y ⟧ ⟹ A' x y;
    ⋀x y. ⟦ x ∈ outs-gpv (ℐ-uniform UNIV {x. Domainp R' x}) gpv; y ∈ outs-gpv
(ℐ-uniform UNIV {x. Rangep R' x}) gpv'; C x y ⟧ ⟹ C' x y;
    R' ≤ R ⟧
  ⟹ rel-gpv'' A' C' R' gpv gpv'
  apply(coinduction arbitrary: gpv gpv')
  apply(drule rel-gpv''D)
  apply(erule rel-spmf-mono-strong)
  apply(erule generat.rel-mono-strong)
   apply(erule generat.set-cases)+
   apply(erule allE, rotate-tac −1)
   apply(erule allE)
   apply(erule impE)
    apply(rule results-gpv.Pure)
    apply simp
   apply(erule impE)
    apply(rule results-gpv.Pure)
    apply simp
   apply simp
  apply(erule generat.set-cases)+
  apply(rotate-tac 1)
  apply(erule allE, rotate-tac −1)
  apply(erule allE)
  apply(erule impE)
   apply(rule outs-gpv.IO)
   apply simp
  apply(erule impE)
   apply(rule outs-gpv.IO)
   apply simp
  apply simp
  apply(erule (1) rel-fun-mono-strong)
  by (fastforce simp add: f1[simplified] f2[simplified])

end

lemma rel-gpv''-refl-strong:
  assumes ⋀x. x ∈ results-gpv (ℐ-uniform UNIV {x. Domainp R x}) gpv ⟹ A
x x
    and ⋀x. x ∈ outs-gpv (ℐ-uniform UNIV {x. Domainp R x}) gpv ⟹ C x x
    and R ≤ (=)
  shows rel-gpv'' A C R gpv gpv
```

proof −
have *rel-gpv''* (=) (=) (=) *gpv gpv* **unfolding** *rel-gpv''-eq* **by** *simp*
then show *?thesis* **using** - - *assms*(*3*) **by**(*rule rel-gpv''-mono-strong*)(*auto intro*: *assms*(*1−2*))
qed

lemma *rel-gpv''-refl-eq-on*:
⟦ ⋀*x*. *x* ∈ *results-gpv* (*I-uniform UNIV X*) *gpv* ⟹ *A x x*; ⋀*out*. *out* ∈ *outs-gpv* (*I-uniform UNIV X*) *gpv* ⟹ *B out out* ⟧
⟹ *rel-gpv'' A B* (*eq-on X*) *gpv gpv*
by(*rule rel-gpv''-refl-strong*) (*auto elim*: *eq-onE*)

lemma *pred-gpv'-mono'* [*mono*]:
pred-gpv' A C R gpv ⟶ *pred-gpv' A' C' R gpv*
if ⋀*x*. *A x* ⟶ *A' x* ⋀*x*. *C x* ⟶ *C' x*
using *that* **unfolding** *pred-gpv'.simps*
by *auto*

primcorec *enforce-I-gpv* :: ('*out*, '*in*) *I* ⇒ ('*a*, '*out*, '*in*) *gpv* ⇒ ('*a*, '*out*, '*in*) *gpv* **where**
enforce-I-gpv I gpv = *GPV*
(*map-spmf* (*map-generat id id* ((∘) (*enforce-I-gpv I*)))
(*map-spmf* (λ*generat*. *case generat of Pure x* ⇒ *Pure x* | *IO out rpv* ⇒ *IO out* (λ*input*. *if input* ∈ *responses-I I out then rpv input else Fail*))
(*enforce-spmf* (*pred-generat* ⊤ (λ*x*. *x* ∈ *outs-I I*) ⊤) (*the-gpv gpv*))))

lemma *enforce-I-gpv-Done* [*simp*]: *enforce-I-gpv I* (*Done x*) = *Done x*
by(*rule gpv.expand*) *simp*

lemma *enforce-I-gpv-Fail* [*simp*]: *enforce-I-gpv I Fail* = *Fail*
by(*rule gpv.expand*) *simp*

lemma *enforce-I-gpv-Pause* [*simp*]:
enforce-I-gpv I (*Pause out rpv*) =
(*if out* ∈ *outs-I I then Pause out* (λ*input*. *if input* ∈ *responses-I I out then enforce-I-gpv I* (*rpv input*) *else Fail*) *else Fail*)
by(*rule gpv.expand*)(*simp add*: *fun-eq-iff*)

lemma *enforce-I-gpv-lift-spmf* [*simp*]: *enforce-I-gpv I* (*lift-spmf p*) = *lift-spmf p*
by(*rule gpv.expand*)(*simp add*: *enforce-map-spmf spmf.map-comp o-def*)

lemma *enforce-I-gpv-bind-gpv* [*simp*]:
enforce-I-gpv I (*bind-gpv gpv f*) = *bind-gpv* (*enforce-I-gpv I gpv*) (*enforce-I-gpv I* ∘ *f*)
by(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
(*auto 4 3 simp add*: *bind-gpv.sel spmf-rel-map bind-map-spmf o-def pred-generat-def elim*!: *generat.set-cases intro*!: *generat.rel-refl-strong rel-spmf-bind-reflI rel-spmf-reflI*

rel-funI split!: *if-splits generat.split-asm*)

lemma *enforce-ℐ-gpv-parametric′*:
includes *lifting-syntax*
notes [*transfer-rule*] = *corec-gpv-parametric′ the-gpv-parametric′ Fail-parametric′*
assumes [*transfer-rule*]: *bi-unique C bi-unique R*
shows (*rel-ℐ C R ===> rel-gpv′′ A C R ===> rel-gpv′′ A C R*) *enforce-ℐ-gpv enforce-ℐ-gpv*
unfolding *enforce-ℐ-gpv-def top-fun-def* **by**(*transfer-prover*)

lemma *enforce-ℐ-gpv-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
bi-unique C ⟹ (*rel-ℐ C* (=) ===> *rel-gpv A C* ===> *rel-gpv A C*) *enforce-ℐ-gpv enforce-ℐ-gpv*
unfolding *rel-gpv-conv-rel-gpv′′* **by**(*rule enforce-ℐ-gpv-parametric′*[*OF - bi-unique-eq*])

lemma *WT-enforce-ℐ-gpv* [*simp*]: *ℐ* ⊢*g enforce-ℐ-gpv ℐ gpv* √
by(*coinduction arbitrary*: *gpv*)(*auto split*: *generat.split-asm*)

lemma *WT-gpv-parametric′*: **includes** *lifting-syntax* **shows**
bi-unique C ⟹ (*rel-ℐ C R ===> rel-gpv′′ A C R ===>* (=)) *WT-gpv WT-gpv*
proof(*rule rel-funI iffI*)+
note [*transfer-rule*] = *the-gpv-parametric′*
show ∗: *ℐ* ⊢*g gpv* √ **if** [*transfer-rule*]: *rel-ℐ C R ℐ ℐ′ bi-unique C*
and ∗: *ℐ′* ⊢*g gpv′* √ *rel-gpv′′ A C R gpv gpv′* **for** *ℐ ℐ′ gpv gpv′ A C R*
using ∗
proof(*coinduction arbitrary*: *gpv gpv′*)
case (*WT-gpv out c gpv gpv′*)
note [*transfer-rule*] = *WT-gpv*(*2*)
have *rel-set* (*rel-generat A C* (*R ===> rel-gpv′′ A C R*)) (*set-spmf* (*the-gpv gpv*)) (*set-spmf* (*the-gpv gpv′*))
by *transfer-prover*
from *rel-setD1*[*OF this WT-gpv*(*3*)] **obtain** *out′ c′*
where [*transfer-rule*]: *C out out′* (*R ===> rel-gpv′′ A C R*) *c c′*
and *out′*: *IO out′ c′* ∈ *set-spmf* (*the-gpv gpv′*)
by(*auto elim*: *generat.rel-cases*)
have *out* ∈ *outs-ℐ ℐ* ⟷ *out′* ∈ *outs-ℐ ℐ′* **by** *transfer-prover*
with *WT-gpvD*(*1*)[*OF WT-gpv*(*1*) *out′*] **have** *?out* **by** *simp*
moreover have *?cont*
proof(*standard*; *goal-cases cont*)
case (*cont input*)
have *rel-set R* (*responses-ℐ ℐ out*) (*responses-ℐ ℐ′ out′*) **by** *transfer-prover*
from *rel-setD1*[*OF this cont*] **obtain** *input′* **where** [*transfer-rule*]: *R input input′*
and *input′*: *input′* ∈ *responses-ℐ ℐ′ out′* **by** *blast*
have *rel-gpv′′ A C R* (*c input*) (*c′ input′*) **by** *transfer-prover*
with *WT-gpvD*(*2*)[*OF WT-gpv*(*1*) *out′ input′*] **show** *?case* **by** *auto*
qed
ultimately show *?case* **..**
qed

show $\mathcal{I}' \vdash g\ gpv' \surd$ **if** *rel-*$\mathcal{I}$ $C\ R\ \mathcal{I}\ \mathcal{I}'$ *bi-unique* $C\ \mathcal{I} \vdash g\ gpv \surd$ *rel-gpv''* $A\ C\ R\ gpv\ gpv'$
for $\mathcal{I}\ \mathcal{I}'\ gpv\ gpv'$
using $*[of\ conversep\ C\ conversep\ R\ \mathcal{I}'\ \mathcal{I}\ gpv\ conversep\ A\ gpv']$ *that*
by(*simp add*: *rel-gpv''-conversep*)
qed

lemma *WT-gpv-map-gpv-id* [*simp*]: $\mathcal{I} \vdash g\ map\text{-}gpv\ f\ id\ gpv \surd \longleftrightarrow \mathcal{I} \vdash g\ gpv \surd$
using *WT-gpv-parametric'*[*of BNF-Def.Grp UNIV id* (=) *BNF-Def.Grp UNIV f*, *folded rel-gpv-conv-rel-gpv''*]
unfolding *gpv.rel-Grp* **unfolding** *eq-alt*[*symmetric*] *relator-eq*
by(*auto simp add*: *rel-fun-def Grp-def bi-unique-eq*)

locale *raw-converter-invariant* =
fixes $\mathcal{I}$ :: ('*call*, '*ret*) $\mathcal{I}$
and $\mathcal{I}'$ :: ('*call'*, '*ret'*) $\mathcal{I}$
and *callee* :: '*s* ⇒ '*call* ⇒ ('*ret* × '*s*, '*call'*, '*ret'*) *gpv*
and *I* :: '*s* ⇒ *bool*
assumes *results-callee*: $\bigwedge s\ x.\ [\![\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow results\text{-}gpv\ \mathcal{I}'\ (callee\ s\ x) \subseteq responses\text{-}\mathcal{I}\ \mathcal{I}\ x \times \{s.\ I\ s\}$
and *WT-callee*: $\bigwedge x\ s.\ [\![\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow \mathcal{I}' \vdash g\ callee\ s\ x \surd$
begin

context begin
private lemma *aux*:
set-spmf (*inline1 callee gpv s*) ⊆ {*Inr* (*out*, *callee'*, *rpv'*) | *out callee' rpv'*.
∃ *call*∈*outs-*$\mathcal{I}$ $\mathcal{I}$. ∃ *s*. *I s* ∧ (∀ *x* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out*. *callee' x* ∈ *sub-gpvs* $\mathcal{I}'$ (*callee s call*))} ∪
{*Inl* (*x*, *s'*) | *x s'*. *x* ∈ *results-gpv* $\mathcal{I}$ *gpv* ∧ *I s'*}
(**is** *?concl* (*inline1 callee*) *gpv s* **is** - ⊆ *?rhs1* ∪ *?rhs2 gpv*)
if $\mathcal{I} \vdash g\ gpv \surd$ *I s*
using *that*
proof(*induction arbitrary*: *gpv s rule*: *inline1-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1'*)
{ fix *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
from *step.prems*(*1*) *IO* **have** *out*: *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*rule WT-gpvD*)
{ fix *x s'*
assume *Pure*: *Pure* (*x*, *s'*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
then have (*x*, *s'*) ∈ *results-gpv* $\mathcal{I}'$ (*callee s out*) **by**(*rule results-gpv.Pure*)
with *out step.prems*(*2*) **have** *x* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out I s'* **by**(*auto dest*: *results-callee*)
from *step.prems*(*1*) *IO this*(*1*) **have** $\mathcal{I} \vdash g\ c\ x \surd$ **by**(*rule WT-gpvD*)
hence *?concl inline1'* (*c x*) *s'* **using** ‹*I s'*› **by**(*rule step.IH*)
also have ... ⊆ *?rhs1* ∪ *?rhs2 gpv* **using** ‹*x* ∈ -› *IO* **by**(*auto intro*: *results-gpv.intros*)

```
    also note calculation
  } moreover {
    fix out' c'
    assume IO out' c' ∈ set-spmf (the-gpv (callee s out))
    hence ∀x∈responses-I I I' out'. c' x ∈ sub-gpvs I' (callee s out)
      by(auto intro: sub-gpvs.base)
   then have ∃call∈outs-I I. ∃s. I s ∧ (∀x∈responses-I I I' out'. c' x ∈ sub-gpvs
I' (callee s call))
      using out step.prems(2) by blast
  } moreover note calculation }
  then show ?case using step.prems
    by(auto 4 3 del: subsetI simp add: bind-UNION intro!: UN-least split: gen-
erat.split intro: results-gpv.intros)
 qed

lemma inline1-in-sub-gpvs-callee:
 assumes Inr (out, callee', rpv') ∈ set-spmf (inline1 callee gpv s)
   and WT: I ⊢g gpv √
   and s: I s
 shows ∃call∈outs-I I. ∃s. I s ∧ (∀x ∈ responses-I I I' out. callee' x ∈ sub-gpvs
I' (callee s call))
 using aux[OF WT s] assms(1) by fastforce

lemma inline1-Inl-results-gpv:
 assumes Inl (x, s') ∈ set-spmf (inline1 callee gpv s)
   and WT: I ⊢g gpv √
   and s: I s
 shows x ∈ results-gpv I gpv ∧ I s'
 using aux[OF WT s] assms(1) by fastforce
end

lemma inline1-in-sub-gpvs:
 assumes Inr (out, callee', rpv') ∈ set-spmf (inline1 callee gpv s)
   and (x, s') ∈ results-gpv I' (callee' input)
   and input ∈ responses-I I' out
   and I ⊢g gpv √
   and I s
 shows rpv' x ∈ sub-gpvs I gpv ∧ I s'
proof −
 from ‹I ⊢g gpv √› ‹I s›
 have set-spmf (inline1 callee gpv s) ⊆ {Inr (out, callee', rpv') | out callee' rpv'.
  ∀input ∈ responses-I I' out. ∀(x, s')∈results-gpv I' (callee' input). I s' ∧ rpv'
x ∈ sub-gpvs I gpv}
   ∪ {Inl (x, s') | x s'. I s'} (is ?concl (inline1 callee) gpv s is - ⊆ ?rhs gpv s)
 proof(induction arbitrary: gpv s rule: inline1-fixp-induct)
   case adm show ?case by(intro cont-intro ccpo-class.admissible-leI)
   case bottom show ?case by simp
   case (step inline1')
   { fix out c
```

```
    assume IO: IO out c ∈ set-spmf (the-gpv gpv)
    from step.prems(1) IO have out: out ∈ outs-I I by(rule WT-gpvD)
    { fix x s'
      assume Pure: Pure (x, s') ∈ set-spmf (the-gpv (callee s out))
      then have (x, s') ∈ results-gpv I' (callee s out) by(rule results-gpv.Pure)
        with out step.prems(2) have x ∈ responses-I I out I s' by(auto dest:
results-callee)
      from step.prems(1) IO this(1) have I ⊢g c x √ by(rule WT-gpvD)
      hence ?concl inline1' (c x) s' using ‹I s'› by(rule step.IH)
      also have ... ⊆ ?rhs gpv s' using IO Pure ‹I s›
        by(fastforce intro: sub-gpvs.cont dest: WT-gpv-OutD[OF step.prems(1)]
results-callee[THEN subsetD, OF - - results-gpv.Pure])
      finally have set-spmf (inline1' (c x) s') ⊆ ... .
    } moreover {
      fix out' c' input x s'
      assume IO out' c' ∈ set-spmf (the-gpv (callee s out))
        and input ∈ responses-I I' out' and (x, s') ∈ results-gpv I' (c' input)
      then have c x ∈ sub-gpvs I gpv I s' using IO ‹I s›
          by(auto intro!: sub-gpvs.base dest: WT-gpv-OutD[OF step.prems(1)]
results-callee[THEN subsetD, OF - - results-gpv.IO])
    } moreover note calculation }
    then show ?case using step.prems(2)
      by(auto simp add: bind-UNION intro!: UN-least split: generat.split del:
subsetI)
  qed
  with assms show ?thesis by fastforce
 qed

lemma WT-gpv-inline1:
 assumes Inr (out, rpv, rpv') ∈ set-spmf (inline1 callee gpv s)
   and I ⊢g gpv √
   and I s
 shows out ∈ outs-I I' (is ?thesis1)
   and input ∈ responses-I I' out ⟹ I' ⊢g rpv input √ (is PROP ?thesis2)
   and ⟦ input ∈ responses-I I' out; (x, s') ∈ results-gpv I' (rpv input) ⟧ ⟹ I
⊢g rpv' x √ ∧ I s' (is PROP ?thesis3)
proof −
 from ‹I ⊢g gpv √› ‹I s›
 have set-spmf (inline1 callee gpv s) ⊆ {Inr (out, rpv, rpv') | out rpv rpv'. out
∈ outs-I I'} ∪ {Inl (x, s')| x s'. I s'}
 proof(induction arbitrary: gpv s rule: inline1-fixp-induct)
   { case adm show ?case by(intro cont-intro ccpo-class.admissible-leI) }
   { case bottom show ?case by simp }
   case (step inline1')
   { fix out c
     assume IO: IO out c ∈ set-spmf (the-gpv gpv)
     from step.prems(1) IO have out: out ∈ outs-I I by(rule WT-gpvD)
     { fix x s'
       assume Pure: Pure (x, s') ∈ set-spmf (the-gpv (callee s out))
```

```
      then have ∗: (x, s') ∈ results-gpv I' (callee s out) by(rule results-gpv.Pure)
        with out step.prems(2) have x ∈ responses-I I out I s' by(auto dest:
results-callee)
       from step.prems(1) IO this(1) have I ⊢g c x √ by(rule WT-gpvD)
       note this ‹I s'›
      } moreover {
       fix out' c'
       from out step.prems(2) have I' ⊢g callee s out √ by(rule WT-callee)
       moreover assume IO out' c' ∈ set-spmf (the-gpv (callee s out))
       ultimately have out' ∈ outs-I I' by(rule WT-gpvD)
      } moreover note calculation }
      then show ?case using step.prems(2)
     by(auto del: subsetI simp add: bind-UNION intro!: UN-least split: generat.split
intro!: step.IH[THEN order-trans])
   qed
   then show ?thesis1 using assms by auto

   assume input ∈ responses-I I' out
   with inline1-in-sub-gpvs-callee[OF ‹Inr - ∈ -› ‹I ⊢g gpv √› ‹I s›]
   obtain out' s where out' ∈ outs-I I
     and ∗: rpv input ∈ sub-gpvs I' (callee s out') and I s by blast
   from ‹out' ∈ -› ‹I s› have I' ⊢g callee s out' √ by(rule WT-callee)
   then show I' ⊢g rpv input √ using ∗ by(rule WT-sub-gpvsD)

   assume (x, s') ∈ results-gpv I' (rpv input)
   with ‹Inr - ∈ -› have rpv' x ∈ sub-gpvs I gpv ∧ I s'
     using ‹input ∈ -› ‹I ⊢g gpv √› assms(3) ‹I s› by−(rule inline1-in-sub-gpvs)
   with ‹I ⊢g gpv √› show I ⊢g rpv' x √ ∧ I s' by(blast intro: WT-sub-gpvsD)
  qed

lemma WT-gpv-inline-invar:
  assumes I ⊢g gpv √
    and I s
  shows I' ⊢g inline callee gpv s √
  using assms
proof(coinduction arbitrary: gpv s rule: WT-gpv-coinduct-bind)
  case (WT-gpv out c gpv)
  from ‹IO out c ∈ -› obtain callee' rpv'
    where Inr: Inr (out, callee', rpv') ∈ set-spmf (inline1 callee gpv s)
      and c: c = (λinput. callee' input ⤜ (λ(x, s). inline callee (rpv' x) s))
    by(clarsimp simp add: inline-sel split: sum.split-asm)
  from Inr ‹I ⊢g gpv √› ‹I s› have ?out by(rule WT-gpv-inline1)
  moreover have ?cont TYPE('ret × 's) (is ∀ input∈-. - ∨ - ∨ ?case' input)
  proof(rule ballI disjI2)+
    fix input
    assume input ∈ responses-I I' out
    with Inr ‹I ⊢g gpv √ › ‹I s› have I' ⊢g callee' input √
      and ⋀x s'. (x, s') ∈ results-gpv I' (callee' input) ⟹ I ⊢g rpv' x √ ∧ I s'
      by(blast dest: WT-gpv-inline1)+
```

then show *?case' input* **by**(*subst c*)(*auto 4 5*)
qed
ultimately show *?case TYPE('ret × 's)* **..**
qed

end

lemma *WT-gpv-inline*:
assumes $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *results-gpv* $\mathcal{I}'$ (*callee s x*) $\subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* $\times$ *UNIV*
and $\bigwedge x\ s.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow \mathcal{I}' \vdash g$ *callee s x* $\surd$
and $\mathcal{I} \vdash g$ *gpv* $\surd$
shows $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$
proof −
interpret *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ *callee* λ*-. True*
using *assms* **by**(*unfold-locales*)*auto*
show *?thesis* **by**(*rule WT-gpv-inline-invar*)(*use assms* **in** *auto*)
qed

lemma *results-gpv-sub-gvps*: *gpv' ∈ sub-gpvs* $\mathcal{I}$ *gpv* $\Longrightarrow$ *results-gpv* $\mathcal{I}$ *gpv' ⊆ results-gpv* $\mathcal{I}$ *gpv*
by(*induction rule*: *sub-gpvs.induct*)(*auto intro*: *results-gpv.IO*)

lemma *in-results-gpv-sub-gvps*: ⟦ *x ∈ results-gpv* $\mathcal{I}$ *gpv'*; *gpv' ∈ sub-gpvs* $\mathcal{I}$ *gpv* ⟧ $\Longrightarrow$ *x ∈ results-gpv* $\mathcal{I}$ *gpv*
using *results-gpv-sub-gvps*[*of gpv'* $\mathcal{I}$ *gpv*] **by** *blast*

context *raw-converter-invariant* **begin**
lemma *results-gpv-inline-aux*:
assumes (*x*, *s'*) *∈ results-gpv* $\mathcal{I}'$ (*inline-aux callee y*)
shows ⟦ *y* = *Inl* (*gpv*, *s*); $\mathcal{I} \vdash g$ *gpv* $\surd$; *I s* ⟧ $\Longrightarrow$ *x ∈ results-gpv* $\mathcal{I}$ *gpv* ∧ *I s'*
and ⟦ *y* = *Inr* (*rpv*, *callee'*); ∀(*z*, *s'*) *∈ results-gpv* $\mathcal{I}'$ *callee'*. $\mathcal{I} \vdash g$ *rpv z* $\surd$ ∧ *I s'* ⟧
$\Longrightarrow$ ∃(*z*, *s''*) *∈ results-gpv* $\mathcal{I}'$ *callee'*. *x ∈ results-gpv* $\mathcal{I}$ (*rpv z*) ∧ *I s''* ∧ *I s'*
using *assms*
proof(*induction gvp'*≡*inline-aux callee y arbitrary*: *y gpv s rpv callee'*)
case *Pure* **case** *1*
with *Pure* **show** *?case*
by(*auto simp add*: *inline-aux.sel split*: *sum.split-asm dest*: *inline1-Inl-results-gpv*)
next
case *Pure* **case** *2*
with *Pure* **show** *?case*
by(*clarsimp simp add*: *inline-aux.sel split*: *sum.split-asm*)
(*fastforce split*: *generat.split-asm dest*: *inline1-Inl-results-gpv intro*: *results-gpv.Pure*)+
next
case (*IO out c input*) **case** *1*
with *IO*(*1*) **obtain** *rpv rpv'* **where** *inline1*: *Inr* (*out*, *rpv*, *rpv'*) *∈ set-spmf* (*inline1 callee gpv s*)
and *c*: *c* = (*λinput. inline-aux callee* (*Inr* (*rpv'*, *rpv input*)))

by(*auto simp add*: *inline-aux.sel split*: *sum.split-asm*)
from *inline1*[*THEN inline1-in-sub-gpvs*, *OF* - ‹*input* ∈ *responses-I* $\mathcal{I}'$ *out*› - ‹*I s*›] ‹$\mathcal{I}$ ⊢*g gpv* √›
have ∀(*z*, *s'*)∈*results-gpv* $\mathcal{I}'$ (*rpv input*). $\mathcal{I}$ ⊢*g rpv' z* √ ∧ *I s'*
by(*auto intro*: *WT-sub-gpvsD*)
from *IO*(*5*)[*unfolded c*, *OF refl refl this*] **obtain** *input' s''*
where *input'*: (*input'*, *s''*) ∈ *results-gpv* $\mathcal{I}'$ (*rpv input*)
and *x*: *x* ∈ *results-gpv* $\mathcal{I}$ (*rpv' input'*) **and** *s''*: *I s'' I s'*
by *auto*
from *inline1*[*THEN inline1-in-sub-gpvs*, *OF input'* ‹*input* ∈ *responses-I* $\mathcal{I}'$ *out*› ‹$\mathcal{I}$ ⊢*g gpv* √› ‹*I s*›] *s'' x*
show *?case* **by**(*auto intro*: *in-results-gpv-sub-gvps*)
next
case (*IO out c input*) **case** *2*
from *IO*(*1*) *2*(*1*) **consider** (*Pure*) *input' s'' rpv' rpv''*
where *Pure* (*input'*, *s''*) ∈ *set-spmf* (*the-gpv callee'*) *Inr* (*out*, *rpv'*, *rpv''*) ∈ *set-spmf* (*inline1 callee* (*rpv input'*) *s''*)
c = (λ*input*. *inline-aux callee* (*Inr* (*rpv''*, *rpv' input*)))
| (*Cont*) *rpv'* **where** *IO out rpv'* ∈ *set-spmf* (*the-gpv callee'*) *c* = (λ*input*. *inline-aux callee* (*Inr* (*rpv*, *rpv' input*)))
by(*auto simp add*: *inline-aux.sel split*: *sum.split-asm*; *rename-tac generat*; *case-tac generat*; *clarsimp*)
then show *?case*
proof *cases*
case *Pure*
have *res*: (*input'*, *s''*) ∈ *results-gpv* $\mathcal{I}'$ *callee'* **using** *Pure*(*1*) **by**(*rule results-gpv.Pure*)
with *2* **have** *WT*: $\mathcal{I}$ ⊢*g rpv input'* √ *I s''* **by** *auto*
have ∀(*z*, *s'*)∈*results-gpv* $\mathcal{I}'$ (*rpv' input*). $\mathcal{I}$ ⊢*g rpv'' z* √ ∧ *I s'*
using *inline1-in-sub-gpvs*[*OF Pure*(*2*) - ‹*input* ∈ -› *WT*] *WT* **by**(*auto intro*: *WT-sub-gpvsD*)
from *IO*(*5*)[*unfolded Pure*(*3*), *OF refl refl this*] **obtain** *z s'''*
where *z*: (*z*, *s'''*) ∈ *results-gpv* $\mathcal{I}'$ (*rpv' input*)
and *x*: *x* ∈ *results-gpv* $\mathcal{I}$ (*rpv'' z*) **and** *s'*: *I s''' I s'* **by** *auto*
have *x* ∈ *results-gpv* $\mathcal{I}$ (*rpv input'*) **using** *x inline1-in-sub-gpvs*[*OF Pure*(*2*) *z* ‹*input* ∈ -› *WT*]
by(*auto intro*: *in-results-gpv-sub-gvps*)
then show *?thesis* **using** *res WT s'* **by** *auto*
next
case *Cont*
have ∀(*z*, *s'*)∈*results-gpv* $\mathcal{I}'$ (*rpv' input*). $\mathcal{I}$ ⊢*g rpv z* √ ∧ *I s'*
using *Cont* 2 ‹*input* ∈ *responses-I* $\mathcal{I}'$ *out*› **by**(*auto intro*: *results-gpv.IO*)
from *IO*(*5*)[*unfolded Cont*, *OF refl refl this*] **obtain** *z s''*
where (*z*, *s''*) ∈ *results-gpv* $\mathcal{I}'$ (*rpv' input*) *x* ∈ *results-gpv* $\mathcal{I}$ (*rpv z*) *I s'' I s'* **by** *auto*
then show *?thesis* **using** *Cont*(*1*) ‹*input* ∈ -› **by**(*auto intro*: *results-gpv.IO*)
qed
qed

lemma *results-gpv-inline*:

$[\![(x, s') \in \mathit{results\text{-}gpv}\ \mathcal{I}'\ (\mathit{inline}\ \mathit{callee}\ \mathit{gpv}\ s); \mathcal{I} \vdash g\ \mathit{gpv}\ \surd; I\ s]\!] \Longrightarrow x \in \mathit{results\text{-}gpv}\ \mathcal{I}\ \mathit{gpv} \wedge I\ s'$
unfolding *inline-def* **by**(*rule results-gpv-inline-aux*(*1*)[*OF - refl*])

end

lemma *inline-map-gpv*:
inline callee (*map-gpv f g gpv*) *s* = *map-gpv* (*apfst f*) *id* (*inline* (λ*s x*. *callee s* (*g x*)) *gpv s*)
unfolding *apfst-def*
by(*rule inline-parametric*
[**where** *S*=*BNF-Def.Grp UNIV id* **and** *C*=*BNF-Def.Grp UNIV g* **and** *C'*=*BNF-Def.Grp UNIV id* **and** *A*=*BNF-Def.Grp UNIV f*,
THEN rel-funD, *THEN rel-funD*, *THEN rel-funD*,
unfolded gpv.rel-Grp prod.rel-Grp, *simplified*, *folded eq-alt*, *unfolded Grp-def*, *simplified*])
(*auto simp add*: *rel-fun-def relator-eq*)

lemma *I-full-le-plus-I*: *I-full* ≤ *plus-I I1 I2* **if** *I-full* ≤ *I1 I-full* ≤ *I2*
using *that* **by**(*auto simp add*: *le-I-def top-unique*)

lemma *plus-I-mono*: *plus-I I1 I2* ≤ *plus-I I1' I2'* **if** *I1* ≤ *I1' I2* ≤ *I2'*
using *that* **by**(*fastforce simp add*: *le-I-def*)

primcorec (*transfer*) *left-gpv* :: ('*a*, '*out*, '*in*) *gpv* ⇒ ('*a*, '*out* + '*out'*, '*in* + '*in'*) *gpv* **where**
the-gpv (*left-gpv gpv*) =
map-spmf (*map-generat id Inl* (λ*rpv input*. *case input of Inl input'* ⇒ *left-gpv* (*rpv input'*) | - ⇒ *Fail*)) (*the-gpv gpv*)

abbreviation *left-rpv* :: ('*a*, '*out*, '*in*) *rpv* ⇒ ('*a*, '*out* + '*out'*, '*in* + '*in'*) *rpv* **where**
left-rpv rpv ≡ λ*input*. *case input of Inl input'* ⇒ *left-gpv* (*rpv input'*) | - ⇒ *Fail*

primcorec (*transfer*) *right-gpv* :: ('*a*, '*out*, '*in*) *gpv* ⇒ ('*a*, '*out'* + '*out*, '*in'* + '*in*) *gpv* **where**
the-gpv (*right-gpv gpv*) =
map-spmf (*map-generat id Inr* (λ*rpv input*. *case input of Inr input'* ⇒ *right-gpv* (*rpv input'*) | - ⇒ *Fail*)) (*the-gpv gpv*)

abbreviation *right-rpv* :: ('*a*, '*out*, '*in*) *rpv* ⇒ ('*a*, '*out'* + '*out*, '*in'* + '*in*) *rpv* **where**
right-rpv rpv ≡ λ*input*. *case input of Inr input'* ⇒ *right-gpv* (*rpv input'*) | - ⇒ *Fail*

context
includes *lifting-syntax*
notes [*transfer-rule*] = *corec-gpv-parametric' Fail-parametric' the-gpv-parametric'*
begin

lemmas *left-gpv-parametric* = *left-gpv.transfer*

lemma *left-gpv-parametric'*:
(*rel-gpv'' A C R* ===> *rel-gpv'' A* (*rel-sum C C'*) (*rel-sum R R'*)) *left-gpv left-gpv*
unfolding *left-gpv-def* **by** *transfer-prover*

lemmas *right-gpv-parametric* = *right-gpv.transfer*

lemma *right-gpv-parametric'*:
(*rel-gpv'' A C' R'* ===> *rel-gpv'' A* (*rel-sum C C'*) (*rel-sum R R'*)) *right-gpv right-gpv*
unfolding *right-gpv-def* **by** *transfer-prover*

end

lemma *left-gpv-Done* [*simp*]: *left-gpv* (*Done x*) = *Done x*
by(*rule gpv.expand*) *simp*

lemma *right-gpv-Done* [*simp*]: *right-gpv* (*Done x*) = *Done x*
by(*rule gpv.expand*) *simp*

lemma *left-gpv-Pause* [*simp*]:
left-gpv (*Pause x rpv*) = *Pause* (*Inl x*) (λ*input*. *case input of Inl input'* ⇒ *left-gpv* (*rpv input'*) | - ⇒ *Fail*)
by(*rule gpv.expand*) *simp*

lemma *right-gpv-Pause* [*simp*]:
right-gpv (*Pause x rpv*) = *Pause* (*Inr x*) (λ*input*. *case input of Inr input'* ⇒ *right-gpv* (*rpv input'*) | - ⇒ *Fail*)
by(*rule gpv.expand*) *simp*

lemma *left-gpv-map*: *left-gpv* (*map-gpv f g gpv*) = *map-gpv f* (*map-sum g h*) (*left-gpv gpv*)
using *left-gpv.transfer*[*of BNF-Def.Grp UNIV f BNF-Def.Grp UNIV g BNF-Def.Grp UNIV h*]
unfolding *sum.rel-Grp gpv.rel-Grp*
by(*auto simp add*: *rel-fun-def Grp-def*)

lemma *right-gpv-map*: *right-gpv* (*map-gpv f g gpv*) = *map-gpv f* (*map-sum h g*) (*right-gpv gpv*)
using *right-gpv.transfer*[*of BNF-Def.Grp UNIV f BNF-Def.Grp UNIV g BNF-Def.Grp UNIV h*]

unfolding *sum.rel-Grp gpv.rel-Grp*
by(*auto simp add*: *rel-fun-def Grp-def*)

lemma *results'-gpv-left-gpv* [*simp*]:
results'-gpv (*left-gpv gpv* :: ('*a*, '*out* + '*out'*, '*in* + '*in'*) *gpv*) = *results'-gpv gpv*
(**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv'*≡*left-gpv gpv* :: ('*a*, '*out* + '*out'*, '*in* + '*in'*) *gpv arbitrary*: *gpv*)
(*fastforce simp add*: *elim*!: *generat.set-cases intro*: *results'-gpvI split*: *sum.splits*)+
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that*
by(*induction*)
(*auto 4 3 elim*!: *generat.set-cases intro*: *results'-gpv-Pure rev-image-eqI results'-gpv-Cont*[**where** *input*=*Inl* -])
qed

lemma *results'-gpv-right-gpv* [*simp*]:
results'-gpv (*right-gpv gpv* :: ('*a*, '*out'* + '*out*, '*in'* + '*in*) *gpv*) = *results'-gpv gpv*
(**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv'*≡*right-gpv gpv* :: ('*a*, '*out'* + '*out*, '*in'* + '*in*) *gpv arbitrary*: *gpv*)
(*fastforce simp add*: *elim*!: *generat.set-cases intro*: *results'-gpvI split*: *sum.splits*)+
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that*
by(*induction*)
(*auto 4 3 elim*!: *generat.set-cases intro*: *results'-gpv-Pure rev-image-eqI results'-gpv-Cont*[**where** *input*=*Inr* -])
qed

lemma *left-gpv-Inl-transfer*: *rel-gpv''* (=) (λ*l r*. *l* = *Inl r*) (λ*l r*. *l* = *Inl r*) (*left-gpv gpv*) *gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto simp add*: *spmf-rel-map generat.rel-map del*: *rel-funI intro*!: *rel-spmf-reflI generat.rel-refl-strong rel-funI*)

lemma *right-gpv-Inr-transfer*: *rel-gpv''* (=) (λ*l r*. *l* = *Inr r*) (λ*l r*. *l* = *Inr r*) (*right-gpv gpv*) *gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto simp add*: *spmf-rel-map generat.rel-map del*: *rel-funI intro*!: *rel-spmf-reflI generat.rel-refl-strong rel-funI*)

lemma *exec-gpv-plus-oracle-left*: *exec-gpv* (*plus-oracle oracle1 oracle2*) (*left-gpv gpv*) *s* = *exec-gpv oracle1 gpv s*
unfolding *spmf-rel-eq*[*symmetric*] *prod.rel-eq*[*symmetric*]
by(*rule exec-gpv-parametric'*[**where** *A*=(=) **and** *S*=(=) **and** *CALL*=λ*l r*. *l* = *Inl r* **and** *R*=λ*l r*. *l* = *Inl r*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
(*auto intro*!: *rel-funI simp add*: *spmf-rel-map apfst-def map-prod-def rel-prod-conv*

intro: *rel-spmf-reflI left-gpv-Inl-transfer*)

lemma *exec-gpv-plus-oracle-right*: *exec-gpv* (*plus-oracle oracle1 oracle2*) (*right-gpv gpv*) *s* = *exec-gpv oracle2 gpv s*
unfolding *spmf-rel-eq*[*symmetric*] *prod.rel-eq*[*symmetric*]
by(*rule exec-gpv-parametric'*[**where** *A*=(=) **and** *S*=(=) **and** *CALL*=λ*l r*. *l* = *Inr r* **and** *R*=λ*l r*. *l* = *Inr r*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
(*auto intro*!: *rel-funI simp add*: *spmf-rel-map apfst-def map-prod-def rel-prod-conv intro*: *rel-spmf-reflI right-gpv-Inr-transfer*)

lemma *left-gpv-bind-gpv*: *left-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*left-gpv gpv*) (*left-gpv* ∘ *f*)
by(*coinduction arbitrary*:*gpv f rule*: *gpv.coinduct-strong*)
(*auto 4 4 simp add*: *bind-map-spmf spmf-rel-map intro*!: *rel-spmf-reflI rel-spmf-bindI*[*of* (=)] *generat.rel-refl rel-funI split*: *sum.splits*)

lemma *inline1-left-gpv*:
inline1 (λ*s q*. *left-gpv* (*callee s q*)) *gpv s* =
map-spmf (*map-sum id* (*map-prod Inl* (*map-prod left-rpv id*))) (*inline1 callee gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf inline1.mono inline1.mono inline1-def inline1-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1' inline1''*)
then show *?case*
by(*auto simp add*: *map-spmf-bind-spmf o-def bind-map-spmf intro*!: *ext bind-spmf-cong split*: *generat.split*)
qed

lemma *left-gpv-inline*: *left-gpv* (*inline callee gpv s*) = *inline* (λ*s q*. *left-gpv* (*callee s q*)) *gpv s*
by(*coinduction arbitrary*: *callee gpv s rule*: *gpv-coinduct-bind*)
(*fastforce simp add*: *inline-sel spmf-rel-map inline1-left-gpv left-gpv-bind-gpv o-def split-def intro*!: *rel-spmf-reflI split*: *sum.split intro*!: *rel-funI gpv.rel-refl-strong*)

lemma *right-gpv-bind-gpv*: *right-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*right-gpv gpv*) (*right-gpv* ∘ *f*)
by(*coinduction arbitrary*:*gpv f rule*: *gpv.coinduct-strong*)
(*auto 4 4 simp add*: *bind-map-spmf spmf-rel-map intro*!: *rel-spmf-reflI rel-spmf-bindI*[*of* (=)] *generat.rel-refl rel-funI split*: *sum.splits*)

lemma *inline1-right-gpv*:
inline1 (λ*s q*. *right-gpv* (*callee s q*)) *gpv s* =
map-spmf (*map-sum id* (*map-prod Inr* (*map-prod right-rpv id*))) (*inline1 callee gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf inline1.mono inline1.mono inline1-def inline1-def*,

unfolded lub-spmf-empty, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1' inline1''*)
then show *?case*
by(*auto simp add*: *map-spmf-bind-spmf o-def bind-map-spmf intro*!: *ext bind-spmf-cong split*: *generat.split*)
qed

lemma *right-gpv-inline*: *right-gpv* (*inline callee gpv s*) = *inline* (λ*s q*. *right-gpv* (*callee s q*)) *gpv s*
by(*coinduction arbitrary*: *callee gpv s rule*: *gpv-coinduct-bind*)
(*fastforce simp add*: *inline-sel spmf-rel-map inline1-right-gpv right-gpv-bind-gpv o-def split-def intro*!: *rel-spmf-reflI split*: *sum.split intro*!: *rel-funI gpv.rel-refl-strong*)

lemma *WT-gpv-left-gpv*: $\mathcal{I}1 \vdash g$ *gpv* $\surd \Longrightarrow \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash g$ *left-gpv gpv* $\surd$
by(*coinduction arbitrary*: *gpv*)(*auto 4 4 dest*: *WT-gpvD*)

lemma *WT-gpv-right-gpv*: $\mathcal{I}2 \vdash g$ *gpv* $\surd \Longrightarrow \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash g$ *right-gpv gpv* $\surd$
by(*coinduction arbitrary*: *gpv*)(*auto 4 4 dest*: *WT-gpvD*)

lemma *results-gpv-left-gpv* [*simp*]: *results-gpv* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) (*left-gpv gpv*) = *results-gpv* $\mathcal{I}1$ *gpv*
(**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv'*≡*left-gpv gpv* :: ('*a*, '*b* + '*c*, '*d* + '*e*) *gpv arbitrary*: *gpv rule*: *results-gpv.induct*)
(*fastforce intro*: *results-gpv.intros*)+
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that*
by(*induction*)(*fastforce intro*: *results-gpv.intros*)+
qed

lemma *results-gpv-right-gpv* [*simp*]: *results-gpv* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) (*right-gpv gpv*) = *results-gpv* $\mathcal{I}2$ *gpv*
(**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv'*≡*right-gpv gpv* :: ('*a*, '*b* + '*c*, '*d* + '*e*) *gpv arbitrary*: *gpv rule*: *results-gpv.induct*)
(*fastforce intro*: *results-gpv.intros*)+
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that*
by(*induction*)(*fastforce intro*: *results-gpv.intros*)+
qed

lemma *left-gpv-Fail* [*simp*]: *left-gpv Fail* = *Fail*
by(*rule gpv.expand*) *auto*

lemma *right-gpv-Fail* [*simp*]: *right-gpv Fail* = *Fail*

by(*rule gpv.expand*) *auto*

lemma *rsuml-lsumr-left-gpv-left-gpv*:*map-gpv′ id rsuml lsumr* (*left-gpv* (*left-gpv gpv*)) = *left-gpv gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *lsumr.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *rsuml-lsumr-left-gpv-right-gpv*: *map-gpv′ id rsuml lsumr* (*left-gpv* (*right-gpv gpv*)) = *right-gpv* (*left-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *lsumr.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *rsuml-lsumr-right-gpv*: *map-gpv′ id rsuml lsumr* (*right-gpv gpv*) = *right-gpv* (*right-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *lsumr.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *map-gpv′-map-gpv-swap*:
map-gpv′ f g h (*map-gpv f′ id gpv*) = *map-gpv* (*f* ∘ *f′*) *id* (*map-gpv′ id g h gpv*)
by(*simp add*: *map-gpv-conv-map-gpv′ map-gpv′-comp*)

lemma *lsumr-rsuml-left-gpv*: *map-gpv′ id lsumr rsuml* (*left-gpv gpv*) = *left-gpv* (*left-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split intro*: *exI*[**where** *x*=*Fail*])

lemma *lsumr-rsuml-right-gpv-left-gpv*:
map-gpv′ id lsumr rsuml (*right-gpv* (*left-gpv gpv*)) = *left-gpv* (*right-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split intro*: *exI*[**where** *x*=*Fail*])

lemma *lsumr-rsuml-right-gpv-right-gpv*:
map-gpv′ id lsumr rsuml (*right-gpv* (*right-gpv gpv*)) = *right-gpv gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *rsuml.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *in-set-spmf-extend-state-oracle* [*simp*]:
x ∈ *set-spmf* (*extend-state-oracle oracle s y*) ⟷
fst (*snd x*) = *fst s* ∧ (*fst x*, *snd* (*snd x*)) ∈ *set-spmf* (*oracle* (*snd s*) *y*)
by(*auto 4 4 simp add*: *extend-state-oracle-def split-beta intro*: *rev-image-eqI prod.expand*)

lemma *extend-state-oracle-plus-oracle*:

extend-state-oracle (*plus-oracle oracle1 oracle2*) = *plus-oracle* (*extend-state-oracle oracle1*) (*extend-state-oracle oracle2*)
proof ((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases s*; *cases q*) (*simp-all add*: *apfst-def spmf.map-comp o-def split-def*)
qed

definition *stateless-callee* :: ('*a* ⇒ ('*b*, '*out*, '*in*) *gpv*) ⇒ ('*s* ⇒ '*a* ⇒ ('*b* × '*s*, '*out*, '*in*) *gpv*) **where**
stateless-callee callee s = *map-gpv* (λ*b*. (*b*, *s*)) *id* ∘ *callee*

lemma *stateless-callee-parametric'*:
includes *lifting-syntax* **notes** [*transfer-rule*] = *map-gpv-parametric'* **shows**
((*A* ===> *rel-gpv''* *B C R*) ===> *S* ===> *A* ===> (*rel-gpv''* (*rel-prod B S*) *C R*))
stateless-callee stateless-callee
unfolding *stateless-callee-def* **by** *transfer-prover*

lemma *id-oralce-alt-def*: *id-oracle* = *stateless-callee* (λ*x*. *Pause x Done*)
by(*simp add*: *id-oracle-def fun-eq-iff stateless-callee-def*)

context
fixes *left* :: '*s1* ⇒ '*x1* ⇒ ('*y1* × '*s1*, '*call1*, '*ret1*) *gpv*
and *right* :: '*s2* ⇒ '*x2* ⇒ ('*y2* × '*s2*, '*call2*, '*ret2*) *gpv*
begin

fun *parallel-intercept* :: '*s1* × '*s2* ⇒ '*x1* + '*x2* ⇒ (('*y1* + '*y2*) × ('*s1* × '*s2*), '*call1* + '*call2*, '*ret1* + '*ret2*) *gpv*
where
parallel-intercept (*s1*, *s2*) (*Inl a*) = *left-gpv* (*map-gpv* (*map-prod Inl* (λ*s1'*. (*s1'*, *s2*))) *id* (*left s1 a*))
| *parallel-intercept* (*s1*, *s2*) (*Inr b*) = *right-gpv* (*map-gpv* (*map-prod Inr* (*Pair s1*)) *id* (*right s2 b*))

end

lemma *expectation-gpv-*$\mathcal{I}$*-mono*:
defines *expectation-gpv'* ≡ *expectation-gpv*
assumes *le*: $\mathcal{I} \leq \mathcal{I}'$
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *expectation-gpv fail* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv' fail* $\mathcal{I}'$ *f gpv*
using *WT*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case *step* [*unfolded expectation-gpv'-def*]: (*step expectation-gpv'*)

show *?case* **unfolding** *expectation-gpv′-def*
by(*subst expectation-gpv.simps*)
(*clarsimp intro*!: *add-mono nn-integral-mono-AE INF-mono split*: *generat.split*
, *auto intro*!: *bexI step add-mono nn-integral-mono-AE INF-mono split*: *generat.split dest*: *WT-gpvD*[*OF step.prems*] *intro*!: *step dest*: *responses-ℐ-mono*[*OF le*])
qed

lemma *pgen-lossless-gpv-mono*:
assumes ∗: *pgen-lossless-gpv fail ℐ gpv*
and *le*: $\mathcal{I} \leq \mathcal{I}'$
and *WT*: $\mathcal{I} \vdash g\ gpv\ \surd$
and *fail*: *fail* ≤ *1*
shows *pgen-lossless-gpv fail ℐ′ gpv*
unfolding *pgen-lossless-gpv-def*
proof(*rule antisym*)
from *WT le* **have** $\mathcal{I}' \vdash g\ gpv\ \surd$ **by**(*rule WT-gpv-ℐ-mono*)
from *expectation-gpv-const-le*[*OF this*, *of fail 1*] *fail*
show *expectation-gpv fail ℐ′* (λ-. *1*) *gpv* ≤ *1* **by**(*simp add*: *max-def split*: *if-split-asm*)
from *expectation-gpv-ℐ-mono*[*OF le WT*, *of fail* λ-. *1*] ∗
show *expectation-gpv fail ℐ′* (λ-. *1*) *gpv* ≥ *1* **by**(*simp add*: *pgen-lossless-gpv-def*)
qed

lemma *plossless-gpv-mono*:
⟦ *plossless-gpv ℐ gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g\ gpv\ \surd$ ⟧ ⟹ *plossless-gpv ℐ′ gpv*
by(*erule pgen-lossless-gpv-mono*; *simp*)

lemma *pfinite-gpv-mono*:
⟦ *pfinite-gpv ℐ gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g\ gpv\ \surd$ ⟧ ⟹ *pfinite-gpv ℐ′ gpv*
by(*erule pgen-lossless-gpv-mono*; *simp*)

lemma *pgen-lossless-gpv-parametric′*: **includes** *lifting-syntax* **shows**
((=) ===> *rel-ℐ C R* ===> *rel-gpv″ A C R* ===> (=)) *pgen-lossless-gpv pgen-lossless-gpv*
unfolding *pgen-lossless-gpv-def* **supply** *expectation-gpv-parametric′*[*transfer-rule*]
by *transfer-prover*

lemma *pgen-lossless-gpv-parametric*: **includes** *lifting-syntax* **shows**
((=) ===> *rel-ℐ C* (=) ===> *rel-gpv A C* ===> (=)) *pgen-lossless-gpv pgen-lossless-gpv*
using *pgen-lossless-gpv-parametric′*[*of C* (=) *A*] **by**(*simp add*: *rel-gpv-conv-rel-gpv″*)

lemma *pgen-lossless-gpv-map-gpv-id* [*simp*]:
pgen-lossless-gpv fail ℐ (*map-gpv f id gpv*) = *pgen-lossless-gpv fail ℐ gpv*
using *pgen-lossless-gpv-parametric*[*of BNF-Def.Grp UNIV id BNF-Def.Grp UNIV f*]
unfolding *gpv.rel-Grp*
by(*auto simp add*: *eq-alt*[*symmetric*] *rel-ℐ-eq rel-fun-def Grp-iff*)

context *raw-converter-invariant* **begin**

lemma *expectation-gpv-le-inline*:
defines *expectation-gpv2* ≡ *expectation-gpv 0 I′*
assumes *callee*: ⋀*s x*. ⟦ *x* ∈ *outs-I I*; *I s* ⟧ ⟹ *plossless-gpv I′* (*callee s x*)
and *WT-gpv*: *I* ⊢*g gpv* √
and *I*: *I s*
shows *expectation-gpv 0 I f gpv* ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*)
using *WT-gpv I*
proof(*induction arbitrary*: *gpv s rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv′*)
{ fix *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
with *step.prems* (*1*) **have** *out*: *out* ∈ *outs-I I* **by**(*rule WT-gpv-OutD*)
have (*INF r*∈*responses-I I out*. *expectation-gpv′* (*c r*)) = ∫⁺ *generat*. (*INF r*∈*responses-I I out*. *expectation-gpv′* (*c r*)) ∂*measure-spmf* (*the-gpv* (*callee s out*))
using *WT-callee*[*OF out*, *of s*] *callee*[*OF out*, *of s*] ‹*I s*›
by(*clarsimp simp add*: *measure-spmf.emeasure-eq-measure plossless-iff-colossless-pfinite colossless-gpv-lossless-spmfD lossless-weight-spmfD*)
also have ... ≤ ∫⁺ *generat*. (*case generat of Pure* (*x*, *s′*) ⇒
∫⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out′*, *callee′*, *rpv*) ⇒ *INF r′*∈*responses-I I′ out′*. *expectation-gpv 0 I′* (λ(*r*, *s′*). *expectation-gpv 0 I′* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s′*)) (*callee′ r′*))
∂*measure-spmf* (*inline1 callee* (*c x*) *s′*)
| *IO out′ rpv* ⇒ *INF r′*∈*responses-I I′ out′*. *expectation-gpv 0 I′* (λ(*r′*, *s′*). *expectation-gpv 0 I′* (λ(*x*, *s*). *f x*) (*inline callee* (*c r′*) *s′*)) (*rpv r′*))
∂*measure-spmf* (*the-gpv* (*callee s out*))
proof(*rule nn-integral-mono-AE*; *simp split*!: *generat.split*)
fix *x s′*
assume *Pure*: *Pure* (*x*, *s′*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
hence (*x*, *s′*) ∈ *results-gpv I′* (*callee s out*) **by**(*rule results-gpv.Pure*)
with *results-callee*[*OF out*, *of s*] ‹*I s*› **have** *x*: *x* ∈ *responses-I I out* **and** *I s′* **by** *blast*+
from *x* **have** (*INF r*∈*responses-I I out*. *expectation-gpv′* (*c r*)) ≤ *expectation-gpv′* (*c x*) **by**(*rule INF-lower*)
also have ... ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c x*) *s′*)
by(*rule step.IH*)(*rule WT-gpv-ContD*[*OF step.prems*(*1*) *IO x*] *step.prems* ‹*I s′*›|*assumption*)+
also have ... = ∫⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out′*, *callee′*, *rpv*) ⇒ *INF r′*∈*responses-I I′ out′*. *expectation-gpv 0 I′* (λ(*r*, *s′*). *expectation-gpv 0 I′* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s′*)) (*callee′ r′*))
∂*measure-spmf* (*inline1 callee* (*c x*) *s′*)
unfolding *expectation-gpv2-def*
by(*subst expectation-gpv.simps*)(*auto simp add*: *inline-sel split-def o-def*

intro!: *nn-integral-cong split*: *generat.split sum.split*)
finally show (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) ≤ ... **.**
next
fix *out' rpv*
assume *IO'*: *IO out' rpv* ∈ *set-spmf* (*the-gpv* (*callee s out*))
have (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) ≤ (*INF* (*r*, *s'*)∈(⋃ *r'*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*. *results-gpv* $\mathcal{I}'$ (*rpv r'*)). *expectation-gpv'* (*c r*))
using *IO' results-callee*[*OF out*, *of s*] ‹*I s*› **by**(*intro INF-mono*)(*auto intro*: *results-gpv.IO*)
also have ... = (*INF r'*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*. *INF* (*r*, *s'*)∈*results-gpv* $\mathcal{I}'$ (*rpv r'*). *expectation-gpv'* (*c r*))
by(*simp add*: *INF-UNION*)
also have ... ≤ (*INF r'*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*. *expectation-gpv 0* $\mathcal{I}'$ (λ(*r'*, *s'*). *expectation-gpv 0* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*c r'*) *s'*)) (*rpv r'*))
proof(*rule INF-mono*, *rule bexI*)
fix *r'*
assume *r'*: *r'* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*
have (*INF* (*r*, *s'*)∈*results-gpv* $\mathcal{I}'$ (*rpv r'*). *expectation-gpv'* (*c r*)) ≤ (*INF* (*r*, *s'*)∈*results-gpv* $\mathcal{I}'$ (*rpv r'*). *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c r*) *s'*))
using *IO IO' step.prems out results-callee*[*OF out*, *of s*] *r'*
by(*auto intro*!: *INF-mono rev-bexI step.IH dest*: *WT-gpv-ContD intro*: *results-gpv.IO*)
also have ... ≤ *expectation-gpv 0* $\mathcal{I}'$ (λ(*r'*, *s'*). *expectation-gpv 0* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*c r'*) *s'*)) (*rpv r'*)
unfolding *expectation-gpv2-def* **using** *plossless-gpv-ContD*[*OF callee*, *OF out* ‹*I s*› *IO' r'*] *WT-callee*[*OF out* ‹*I s*›] *IO' r'*
by(*intro plossless-INF-le-expectation-gpv*)(*auto intro*: *WT-gpv-ContD*)
finally show (*INF* (*r*, *s'*)∈*results-gpv* $\mathcal{I}'$ (*rpv r'*). *expectation-gpv'* (*c r*)) ≤ ... **.**
qed
finally show (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) ≤ ... **.**
qed
also note *calculation* **}**
then show *?case* **unfolding** *expectation-gpv2-def*
apply(*rewrite expectation-gpv.simps*)
apply(*rewrite inline-sel*)
apply(*simp add*: *o-def pmf-map-spmf-None*)
apply(*rewrite sum.case-distrib*[**where** *h*=*case-generat* - -])
apply(*simp cong del*: *sum.case-cong-weak*)
apply(*simp add*: *split-beta o-def cong del*: *sum.case-cong-weak*)
apply(*rewrite inline1.simps*)
apply(*rewrite measure-spmf-bind*)
apply(*rewrite nn-integral-bind*[**where** *B*=*measure-spmf* -])
apply *simp*
apply(*simp add*: *space-subprob-algebra*)
apply(*rule nn-integral-mono-AE*)
apply(*clarsimp split*!: *generat.split*)
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return*)

apply(*rewrite measure-spmf-bind*)
apply(*simp add*: *nn-integral-bind*[**where** *B*=*measure-spmf* -] *space-subprob-algebra*)
apply(*subst generat.case-distrib*[**where** *h*=*measure-spmf*])
apply(*subst generat.case-distrib*[**where** *h*=λ*x*. *nn-integral x* -])
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return split-def*)
done
qed

lemma *plossless-inline*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ [\![\ I\ s;\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
unfolding *pgen-lossless-gpv-def*
proof(*rule antisym*)
have *WT'*: $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$ **using** *WT I* **by**(*rule WT-gpv-inline-invar*)
from *expectation-gpv-const-le*[*OF WT'*, *of 0 1*]
show *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*) ≤ *1* **by**(*simp add*: *max-def*)

have *1* = *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *lossless* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have ... ≤ *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*)
by(*rule expectation-gpv-le-inline*[*unfolded split-def*]; *rule callee I WT*)
finally show *1* ≤ ... .
qed

end

lemma *expectation-left-gpv* [*simp*]:
expectation-gpv fail ($\mathcal{I}$ ⊕$_{\mathcal{I}}$ $\mathcal{I}'$) *f* (*left-gpv gpv*) = *expectation-gpv fail* $\mathcal{I}$ *f gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv' expectation-gpv''*)
show *?case*
by (*auto simp add*: *pmf-map-spmf-None o-def case-map-generat image-comp split*: *generat.split intro*!: *nn-integral-cong-AE INF-cong step.IH*)
qed

lemma *expectation-right-gpv* [*simp*]:
expectation-gpv fail ($\mathcal{I}$ ⊕$_{\mathcal{I}}$ $\mathcal{I}'$) *f* (*right-gpv gpv*) = *expectation-gpv fail* $\mathcal{I}'$ *f gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv' expectation-gpv''*)

show *?case*
by (*auto simp add*: *pmf-map-spmf-None o-def case-map-generat image-comp split*: *generat.split intro*!: *nn-integral-cong-AE INF-cong step.IH*)
qed

lemma *pgen-lossless-left-gpv* [*simp*]: *pgen-lossless-gpv fail* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*left-gpv gpv*) = *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma *pgen-lossless-right-gpv* [*simp*]: *pgen-lossless-gpv fail* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*right-gpv gpv*) = *pgen-lossless-gpv fail* $\mathcal{I}'$ *gpv*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma (**in** *raw-converter-invariant*) *expectation-gpv-le-inline-invariant*:
defines *expectation-gpv2* ≡ *expectation-gpv 0* $\mathcal{I}'$
assumes *callee*: $\bigwedge s\ x.\ [\![\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow \mathit{plossless\text{-}gpv}\ \mathcal{I}'\ (\mathit{callee}\ s\ x)$
and *WT-gpv*: $\mathcal{I} \vdash g\ \mathit{gpv}\ \surd$
and *I*: *I s*
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*)
using *WT-gpv I*
proof(*induction arbitrary*: *gpv s rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv'*)
{ fix *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
with *step.prems*(*1*) **have** *out*: *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*rule WT-gpv-OutD*)
have (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv' (c r)*) = $\int^+$ *generat*. (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv' (c r)*) *∂measure-spmf* (*the-gpv* (*callee s out*))
using *WT-callee*[*OF out*, *of s*] *callee*[*OF out*, *of s*] *step.prems*(*2*)
by(*clarsimp simp add*: *measure-spmf.emeasure-eq-measure plossless-iff-colossless-pfinite colossless-gpv-lossless-spmfD lossless-weight-spmfD*)
also have ... ≤ $\int^+$ *generat*. (*case generat of Pure* (*x*, *s'*) ⇒
$\int^+$ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out'*, *callee'*, *rpv*) ⇒ *INF r'*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*. *expectation-gpv 0* $\mathcal{I}'$ (λ(*r*, *s'*). *expectation-gpv 0* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s'*)) (*callee' r'*))
∂measure-spmf (*inline1 callee* (*c x*) *s'*)
| *IO out' rpv* ⇒ *INF r'*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*. *expectation-gpv 0* $\mathcal{I}'$ (λ(*r'*, *s'*). *expectation-gpv 0* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*c r'*) *s'*)) (*rpv r'*))
∂measure-spmf (*the-gpv* (*callee s out*))
proof(*rule nn-integral-mono-AE*; *simp split*!: *generat.split*)
fix *x s'*
assume *Pure*: *Pure* (*x*, *s'*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
hence (*x*, *s'*) ∈ *results-gpv* $\mathcal{I}'$ (*callee s out*) **by**(*rule results-gpv.Pure*)
with *results-callee*[*OF out step.prems*(*2*)] **have** *x*: *x* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* **and** *s'*: *I s'* **by** *blast*+
from *this*(*1*) **have** (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv' (c r)*) ≤

expectation-gpv' (*c x*) **by**(*rule INF-lower*)
also have ... ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c x*) *s'*)
by(*rule step.IH*)(*rule WT-gpv-ContD*[*OF step.prems*(*1*) *IO x*] *step.prems s'*|*assumption*)+
also have ... = ∫⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out'*, *callee'*, *rpv*) ⇒ *INF r'*∈*responses-I I' out'*. *expectation-gpv 0 I'* (λ(*r*, *s'*). *expectation-gpv 0 I'* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s'*)) (*callee' r'*))
∂measure-spmf (*inline1 callee* (*c x*) *s'*)
unfolding *expectation-gpv2-def*
by(*subst expectation-gpv.simps*)(*auto simp add*: *inline-sel split-def o-def intro*!: *nn-integral-cong split*: *generat.split sum.split*)
finally show (*INF r*∈*responses-I I out. expectation-gpv'* (*c r*)) ≤ ... **.**
next
fix *out' rpv*
assume *IO'*: *IO out' rpv* ∈ *set-spmf* (*the-gpv* (*callee s out*))
have (*INF r*∈*responses-I I out. expectation-gpv'* (*c r*)) ≤ (*INF* (*r*, *s'*)∈(⋃ *r'*∈*responses-I I' out'*. *results-gpv I'* (*rpv r'*)). *expectation-gpv'* (*c r*))
using *IO' results-callee*[*OF out step.prems*(*2*)] **by**(*intro INF-mono*)(*auto intro*: *results-gpv.IO*)
also have ... = (*INF r'*∈*responses-I I' out'*. *INF* (*r*, *s'*)∈*results-gpv I'* (*rpv r'*). *expectation-gpv'* (*c r*))
by(*simp add*: *INF-UNION*)
also have ... ≤ (*INF r'*∈*responses-I I' out'*. *expectation-gpv 0 I'* (λ(*r'*, *s'*). *expectation-gpv 0 I'* (λ(*x*, *s*). *f x*) (*inline callee* (*c r'*) *s'*)) (*rpv r'*))
proof(*rule INF-mono*, *rule bexI*)
fix *r'*
assume *r'*: *r'* ∈ *responses-I I' out'*
have (*INF* (*r*, *s'*)∈*results-gpv I'* (*rpv r'*). *expectation-gpv'* (*c r*)) ≤ (*INF* (*r*, *s'*)∈*results-gpv I'* (*rpv r'*). *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c r*) *s'*))
using *IO IO' step.prems out results-callee*[*OF out*, *of s*] *r'*
by(*auto intro*!: *INF-mono rev-bexI step.IH dest*: *WT-gpv-ContD intro*: *results-gpv.IO*)
also have ... ≤ *expectation-gpv 0 I'* (λ(*r'*, *s'*). *expectation-gpv 0 I'* (λ(*x*, *s*). *f x*) (*inline callee* (*c r'*) *s'*)) (*rpv r'*)
unfolding *expectation-gpv2-def* **using** *plossless-gpv-ContD*[*OF callee*, *OF out step.prems*(*2*) *IO' r'*] *WT-callee*[*OF out step.prems*(*2*)] *IO' r'*
by(*intro plossless-INF-le-expectation-gpv*)(*auto intro*: *WT-gpv-ContD*)
finally show (*INF* (*r*, *s'*)∈*results-gpv I'* (*rpv r'*). *expectation-gpv'* (*c r*)) ≤ ... **.**
qed
finally show (*INF r*∈*responses-I I out. expectation-gpv'* (*c r*)) ≤ ... **.**
qed
also note *calculation* **}**
then show *?case* **unfolding** *expectation-gpv2-def*
apply(*rewrite expectation-gpv.simps*)
apply(*rewrite inline-sel*)
apply(*simp add*: *o-def pmf-map-spmf-None*)

apply(*rewrite sum.case-distrib*[**where** *h*=*case-generat - -*])
apply(*simp cong del*: *sum.case-cong-weak*)
apply(*simp add*: *split-beta o-def cong del*: *sum.case-cong-weak*)
apply(*rewrite inline1.simps*)
apply(*rewrite measure-spmf-bind*)
apply(*rewrite nn-integral-bind*[**where** *B*=*measure-spmf -*])
apply *simp*
apply(*simp add*: *space-subprob-algebra*)
apply(*rule nn-integral-mono-AE*)
apply(*clarsimp split*!: *generat.split*)
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return*)
apply(*rewrite measure-spmf-bind*)
apply(*simp add*: *nn-integral-bind*[**where** *B*=*measure-spmf -*] *space-subprob-algebra*)
apply(*subst generat.case-distrib*[**where** *h*=*measure-spmf*])
apply(*subst generat.case-distrib*[**where** *h*=λx. *nn-integral x -*])
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return split-def*)
done
qed

lemma (**in** *raw-converter-invariant*) *plossless-inline-invariant*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ [\![\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* $]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
unfolding *pgen-lossless-gpv-def*
proof(*rule antisym*)
have *WT'*: $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$ **using** *WT I* **by**(*rule WT-gpv-inline-invar*)
from *expectation-gpv-const-le*[*OF WT'*, *of 0 1*]
show *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*) $\leq$ *1* **by**(*simp add*: *max-def*)

have *1* = *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *lossless* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have ... $\leq$ *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*)
by(*rule expectation-gpv-le-inline*[*unfolded split-def*]; *rule callee WT WT-callee I*)
finally show *1* $\leq$ ... **.**
qed

context *callee-invariant-on* **begin**

lemma *raw-converter-invariant*: *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ ($\lambda s\ x$. *lift-spmf* (*callee s x*)) *I*
by(*unfold-locales*)(*auto dest*: *callee-invariant WT-callee WT-calleeD*)

lemma (**in** *callee-invariant-on*) *plossless-exec-gpv*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ [\![\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* $]\!] \Longrightarrow$ *lossless-spmf* (*callee s x*)
and *I*: *I s*

shows *lossless-spmf* (*exec-gpv callee gpv s*)
proof −
interpret *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ $\lambda s\ x.$ *lift-spmf* (*callee s x*) *I* **for** $\mathcal{I}'$
by(*rule raw-converter-invariant*)
have *plossless-gpv* $\mathcal{I}$*-full* (*inline* ($\lambda s\ x.$ *lift-spmf* (*callee s x*)) *gpv s*)
using *lossless WT* **by**(*rule plossless-inline*)(*simp-all add*: *callee I*)
from *this*[*THEN plossless-gpv-lossless-spmfD*] **show** *?thesis*
unfolding *exec-gpv-conv-inline1* **by**(*simp add*: *inline-sel*)
qed

end

definition *extend-state-oracle2* :: ('*call*, '*ret*, '*s*) *callee* ⇒ ('*call*, '*ret*, '*s* × '*s*′) *callee* (-† [*1000*] *1000*)
where *extend-state-oracle2 callee* = (λ(*s*, *s*′) *x*. *map-spmf* (λ(*y*, *s*). (*y*, (*s*, *s*′))) (*callee s x*))

lemma *extend-state-oracle2-simps* [*simp*]:
extend-state-oracle2 callee (*s*, *s*′) *x* = *map-spmf* (λ(*y*, *s*). (*y*, (*s*, *s*′))) (*callee s x*)
by(*simp add*: *extend-state-oracle2-def*)

lemma *extend-state-oracle2-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) ===> *rel-prod S S*′ ===> *C* ===> *rel-spmf* (*rel-prod R* (*rel-prod S S*′)))
extend-state-oracle2 extend-state-oracle2
unfolding *extend-state-oracle2-def*[*abs-def*] **by** *transfer-prover*

lemma *callee-invariant-extend-state-oracle2-const* [*simp*]:
callee-invariant oracle† (λ(*s*, *s*′). *I s*′)
by *unfold-locales auto*

lemma *callee-invariant-extend-state-oracle2-const*′:
callee-invariant oracle† ($\lambda s.$ *I* (*snd s*))
by *unfold-locales auto*

lemma *extend-state-oracle2-plus-oracle*:
extend-state-oracle2 (*plus-oracle oracle1 oracle2*) = *plus-oracle* (*extend-state-oracle2 oracle1*) (*extend-state-oracle2 oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases s*; *cases q*) (*simp-all add*: *apfst-def spmf.map-comp o-def split-def*)
qed

lemma *parallel-oracle-conv-plus-oracle*:
parallel-oracle oracle1 oracle2 = *plus-oracle* (*oracle1*†) (†*oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)

then show *?case* **by** (*cases s*; *cases q*) (*auto simp add*: *spmf.map-comp apfst-def o-def split-def map-prod-def*)
qed

lemma *map-sum-parallel-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *map-sum f g* −−−> *map-spmf* (*map-prod* (*map-sum h k*) *id*)) (*parallel-oracle oracle1 oracle2*)
= *parallel-oracle* ((*id* −−−> *f* −−−> *map-spmf* (*map-prod h id*)) *oracle1*) ((*id* −−−> *g* −−−> *map-spmf* (*map-prod k id*)) *oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases s*; *cases q*) (*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed

lemma *map-sum-plus-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *map-sum f g* −−−> *map-spmf* (*map-prod* (*map-sum h k*) *id*)) (*plus-oracle oracle1 oracle2*)
= *plus-oracle* ((*id* −−−> *f* −−−> *map-spmf* (*map-prod h id*)) *oracle1*) ((*id* −−−> *g* −−−> *map-spmf* (*map-prod k id*)) *oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases q*) (*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed

lemma *map-rsuml-plus-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *rsuml* −−−> (*map-spmf* (*map-prod lsumr id*))) (*oracle1* $\oplus_O$ (*oracle2* $\oplus_O$ *oracle3*)) =
((*oracle1* $\oplus_O$ *oracle2*) $\oplus_O$ *oracle3*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case*
proof(*cases q*)
case (*Inl ql*)
then show *?thesis* **by**(*cases ql*)(*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed (*simp add*: *spmf.map-comp o-def apfst-def prod.map-comp id-def*)
qed

lemma *map-lsumr-plus-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *lsumr* −−−> (*map-spmf* (*map-prod rsuml id*))) ((*oracle1* $\oplus_O$ *oracle2*) $\oplus_O$ *oracle3*) =
(*oracle1* $\oplus_O$ (*oracle2* $\oplus_O$ *oracle3*))
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case*
proof(*cases q*)
case (*Inr qr*)

then show *?thesis* **by**(*cases qr*)(*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed (*simp add*: *spmf.map-comp o-def apfst-def prod.map-comp id-def*)
qed

context includes *lifting-syntax* **begin**

definition *lift-state-oracle*
:: (('*s* ⇒ '*a* ⇒ (('*b* × '*t*) × '*s*) *spmf*) ⇒ ('*s'* ⇒ '*a* ⇒ (('*b* × '*t*) × '*s'*) *spmf*))
⇒ ('*t* × '*s* ⇒ '*a* ⇒ ('*b* × '*t* × '*s*) *spmf*) ⇒ ('*t* × '*s'* ⇒ '*a* ⇒ ('*b* × '*t* × '*s'*) *spmf*) **where**
lift-state-oracle F oracle =
(λ(*t*, *s'*) *a*. *map-spmf rprodl* (*F* ((*Pair t* −−−> *id* −−−> *map-spmf lprodr*) *oracle*) *s'* *a*))

lemma *lift-state-oracle-simps* [*simp*]:
lift-state-oracle F oracle (*t*, *s'*) *a* = *map-spmf rprodl* (*F* ((*Pair t* −−−> *id* −−−> *map-spmf lprodr*) *oracle*) *s'* *a*)
by(*simp add*: *lift-state-oracle-def*)

lemma *lift-state-oracle-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(((*S* ===> *A* ===> *rel-spmf* (*rel-prod* (*rel-prod B T*) *S*)) ===> *S'* ===> *A* ===> *rel-spmf* (*rel-prod* (*rel-prod B T*) *S'*))
===> (*rel-prod T S* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-prod T S*)))
===> *rel-prod T S'* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-prod T S'*)))
lift-state-oracle lift-state-oracle
unfolding *lift-state-oracle-def map-fun-def o-def* **by** *transfer-prover*

lemma *lift-state-oracle-extend-state-oracle*:
includes *lifting-syntax*
assumes ⋀*B*. *Transfer.Rel* (((=) ===> (=) ===> *rel-spmf* (*rel-prod B* (=))) ===> (=) ===> (=) ===> *rel-spmf* (*rel-prod B* (=))) *G F*

shows *lift-state-oracle F* (*extend-state-oracle oracle*) = *extend-state-oracle* (*G oracle*)
unfolding *lift-state-oracle-def extend-state-oracle-def*
apply(*clarsimp simp add*: *fun-eq-iff map-fun-def o-def spmf.map-comp split-def rprodl-def*)
subgoal for *t s a*
apply(*rule sym*)
apply(*fold spmf-rel-eq*)
apply(*simp add*: *spmf-rel-map*)
apply(*rule rel-spmf-mono*)
apply(*rule assms*[*unfolded Rel-def*, **where** *B*=λ*x* (*y*, *z*). *x* = *y* ∧ *z* = *t*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
apply(*auto simp add*: *rel-fun-def spmf-rel-map intro*!: *rel-spmf-reflI*)
done
done

lemma *lift-state-oracle-compose*:
lift-state-oracle F (*lift-state-oracle G oracle*) = *lift-state-oracle* (*F* ∘ *G*) *oracle*
by(*simp add*: *lift-state-oracle-def map-fun-def o-def split-def spmf.map-comp*)

lemma *lift-state-oracle-id* [*simp*]: *lift-state-oracle id* = *id*
by(*simp add*: *fun-eq-iff spmf.map-comp o-def*)

lemma *rprodl-extend-state-oracle*: **includes** *lifting-syntax* **shows**
(*rprodl* −−−> *id* −−−> *map-spmf* (*map-prod id lprodr*)) (*extend-state-oracle*
(*extend-state-oracle oracle*)) =
extend-state-oracle oracle
by(*simp add*: *fun-eq-iff spmf.map-comp o-def split-def*)

end

lemma *interaction-bound-map-gpv'*:
assumes *surj h*
shows *interaction-bound consider* (*map-gpv' f g h gpv*) = *interaction-bound*
(*consider* ∘ *g*) *gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF lattice-partial-function-definition*
lattice-partial-function-definition interaction-bound.mono interaction-bound.mono
interaction-bound-def interaction-bound-def, *case-names adm bottom step*])
case (*step interaction-bound' interaction-bound'' gpv*)
have ∗: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *x* ∈ *UNIV* ⟹ *interaction-bound''*
(*c x*) ≤ (⨆ *x*. *interaction-bound''* (*c* (*h x*))) **for** *out c x*
using *assms*[*THEN surjD*, *of x*] **by** (*clarsimp intro*!: *SUP-upper*)

show *?case*
by (*auto simp add*: ∗ *step.IH image-comp split*: *generat.split*
intro!: *SUP-cong* [*OF refl*] *antisym SUP-upper SUP-least*)
qed *simp-all*

lemma *interaction-any-bounded-by-map-gpv'*:
assumes *interaction-any-bounded-by gpv n*
and *surj h*
shows *interaction-any-bounded-by* (*map-gpv' f g h gpv*) *n*
using *assms* **by**(*simp add*: *interaction-bounded-by.simps interaction-bound-map-gpv'*
o-def)

lemma *results'-gpv-map-gpv'*:
assumes *surj h*
shows *results'-gpv* (*map-gpv' f g h gpv*) = *f* ‘ *results'-gpv gpv* (**is** *?lhs* = *?rhs*)
proof −
have ∗:*IO z c* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *x* ∈ *results'-gpv* (*c input*) ⟹
f x ∈ *results'-gpv* (*map-gpv' f g h* (*c input*)) ⟹ *f x* ∈ *results'-gpv* (*map-gpv'*
f g h gpv) **for** *x z gpv c input*
using *surjD*[*OF assms*, *of input*] **by**(*fastforce intro*: *results'-gpvI elim*!: *gen-*
erat.set-cases intro: *rev-image-eqI simp add*: *map-fun-def o-def*)

show *?thesis*
proof(*intro Set.set-eqI iffI*; (*elim imageE*; *hypsubst*)*?*)
show $x \in$ *?rhs* **if** $x \in$ *?lhs* **for** x **using** *that*
by(*induction gpv'≡map-gpv' f g h gpv arbitrary*: *gpv*)(*fastforce elim*!: *generat.set-cases intro*: *results'-gpvI*)+
show $f\ x \in$ *?lhs* **if** $x \in$ *results'-gpv gpv* **for** x **using** *that*
by *induction* (*fastforce intro*: *results'-gpvI elim*!: *generat.set-cases intro*: *rev-image-eqI simp add*: *map-fun-def o-def*
, *clarsimp simp add*: ∗ *elim*!: *generat.set-cases*)
qed
qed

context fixes $B :: 'b \Rightarrow 'c\ set$ **and** $x :: 'a$ **begin**

primcorec *mk-lossless-gpv* :: $('a, 'b, 'c)\ gpv \Rightarrow ('a, 'b, 'c)\ gpv$ **where**
the-gpv (*mk-lossless-gpv gpv*) =
map-spmf (λ*generat*. *case generat of Pure x* ⇒ *Pure x*
| *IO out c* ⇒ *IO out* (λ*input*. *if input* ∈ *B out then mk-lossless-gpv* (*c input*) *else Done x*))
(*the-gpv gpv*)

end

lemma *WT-gpv-outs-gpvI*:
assumes *outs-gpv* $\mathcal{I}$ *gpv* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$
shows $\mathcal{I} \vdash g$ *gpv* $\surd$
using *assms* **by**(*coinduction arbitrary*: *gpv*)(*auto intro*: *outs-gpv.intros*)

lemma *WT-gpv-iff-outs-gpv*:
$\mathcal{I} \vdash g$ *gpv* $\surd \longleftrightarrow$ *outs-gpv* $\mathcal{I}$ *gpv* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$
by(*blast intro*: *WT-gpv-outs-gpvI dest*: *WT-gpv-outs-gpv*)

lemma *WT-gpv-mk-lossless-gpv*:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$
and *outs*: *outs-*$\mathcal{I}$ $\mathcal{I}'$ = *outs-*$\mathcal{I}$ $\mathcal{I}$
shows $\mathcal{I}' \vdash g$ *mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}$) *x gpv* $\surd$
using *assms*(*1*)
by(*coinduction arbitrary*: *gpv*)(*auto 4 3 split*: *generat.split-asm simp add*: *outs dest*: *WT-gpvD*)

lemma *expectation-gpv-mk-lossless-gpv*:
fixes $\mathcal{I}$ *y*
defines *rhs* ≡ *expectation-gpv 0* $\mathcal{I}$ (λ-. *y*)
assumes *WT*: $\mathcal{I}' \vdash g$ *gpv* $\surd$
and *outs*: *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$
shows *expectation-gpv 0* $\mathcal{I}'$ (λ-. *y*) *gpv* ≤ *rhs* (*mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}'$) *x gpv*)
using *WT*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)

case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case *step* [*unfolded rhs-def*]: (*step expectation-gpv'*)
show *?case* **using** *step.prems outs* **unfolding** *rhs-def*
apply(*subst expectation-gpv.simps*)
apply(*clarsimp intro*!: *nn-integral-mono-AE INF-mono split*!: *generat.split if-split*)
subgoal
by(*frule* (*1*) *WT-gpv-OutD*)(*auto simp add*: *in-outs-ℐ-iff-responses-ℐ intro*!: *bexI step.IH*[*unfolded rhs-def*] *dest*: *WT-gpv-ContD*)
apply(*frule* (*1*) *WT-gpv-OutD*; *clarsimp simp add*: *in-outs-ℐ-iff-responses-ℐ ex-in-conv*[*symmetric*])
subgoal for *out c input input'*
using *step.hyps*[*of c input'*] *expectation-gpv-const-le*[*of ℐ' c input' 0 y*]
by− (*drule* (*2*) *WT-gpv-ContD*, *fastforce intro*: *rev-bexI simp add*: *max-def*)
done
qed

lemma *plossless-gpv-mk-lossless-gpv*:
assumes *plossless-gpv ℐ gpv*
and *ℐ* ⊢*g gpv* √
and *outs-ℐ ℐ* = *outs-ℐ ℐ'*
shows *plossless-gpv ℐ'* (*mk-lossless-gpv* (*responses-ℐ ℐ*) *x gpv*)
using *assms expectation-gpv-mk-lossless-gpv*[*OF assms*(*2*), *of ℐ' 1 x*]
unfolding *pgen-lossless-gpv-def*
by −(*rule antisym*[*OF expectation-gpv-const-le*[*THEN order-trans*]]; *simp add*: *WT-gpv-mk-lossless-gpv*)

lemma (**in** *callee-invariant-on*) *exec-gpv-mk-lossless-gpv*:
assumes *ℐ* ⊢*g gpv* √
and *I s*
shows *exec-gpv callee* (*mk-lossless-gpv* (*responses-ℐ ℐ*) *x gpv*) *s* = *exec-gpv callee gpv s*
using *assms*
proof(*induction arbitrary*: *gpv s rule*: *exec-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv'*)
show *?case* **using** *step.prems WT-gpv-OutD*[*OF step.prems*(*1*)]
by(*clarsimp simp add*: *bind-map-spmf intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *generat.split if-split*)
(*force intro*!: *step.IH dest*: *WT-callee*[*THEN WT-calleeD*] *WT-gpv-OutD callee-invariant WT-gpv-ContD*)+
qed

lemma *in-results-gpv-restrict-gpvD*:
assumes *x* ∈ *results-gpv ℐ* (*restrict-gpv ℐ' gpv*)
shows *x* ∈ *results-gpv ℐ gpv*
using *assms*

apply(*induction gpv'≡restrict-gpv* $\mathcal{I}'$ *gpv arbitrary*: *gpv*)
apply(*clarsimp split*: *option.split-asm simp add*: *in-set-spmf*[*symmetric*])
subgoal for ... *y* **by**(*cases y*)(*auto intro*: *results-gpv.intros split*: *if-split-asm*)
apply(*clarsimp split*: *option.split-asm simp add*: *in-set-spmf*[*symmetric*])
subgoal for ... *y* **by**(*cases y*)(*auto intro*: *results-gpv.intros split*: *if-split-asm*)
done

lemma *results-gpv-restrict-gpv*:
results-gpv $\mathcal{I}$ (*restrict-gpv* $\mathcal{I}'$ *gpv*) ⊆ *results-gpv* $\mathcal{I}$ *gpv*
by(*blast intro*: *in-results-gpv-restrict-gpvD*)

lemma *in-results'-gpv-restrict-gpvD*:
$x \in$ *results'-gpv* (*restrict-gpv* $\mathcal{I}'$ *gpv*) ⟹ $x \in$ *results'-gpv gpv*
by(*rule in-results-gpv-restrict-gpvD*[**where** $\mathcal{I} = \mathcal{I}$*-full*, *unfolded results-gpv-*$\mathcal{I}$*-full*])

lemma *expectation-gpv-map-gpv'* [*simp*]:
expectation-gpv fail $\mathcal{I}$ *f* (*map-gpv' g h k gpv*) =
expectation-gpv fail (*map-*$\mathcal{I}$ *h k* $\mathcal{I}$) (*f* ∘ *g*) *gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exp1 exp2*)
have *pmf* (*the-gpv* (*map-gpv' g h k gpv*)) *None* = *pmf* (*the-gpv gpv*) *None*
by(*simp add*: *pmf-map-spmf-None*)
then show *?case*
by *simp*
(*auto simp add*: *nn-integral-measure-spmf step.IH image-comp*
split: *generat.split intro*!: *nn-integral-cong*)
qed

lemma *plossless-gpv-map-gpv'* [*simp*]:
pgen-lossless-gpv b $\mathcal{I}$ (*map-gpv' f g h gpv*) ⟷ *pgen-lossless-gpv b* (*map-*$\mathcal{I}$ *g h* $\mathcal{I}$) *gpv*
unfolding *pgen-lossless-gpv-def* **by**(*simp add*: *o-def*)

end
theory *Resource* **imports**
More-CryptHOL
begin

1 Resources

1.1 Type definition

codatatype ('*a*, '*b*) *resource*
= *Resource* (*run-resource*: '*a* ⇒ ('*b* × ('*a*, '*b*) *resource*) *spmf*)
for *map*: *map-resource'*

rel: *rel-resource′*

lemma *case-resource-conv-run-resource*: *case-resource f res* = *f* (*run-resource res*)
by(*fact resource.case-eq-if*)

1.2 Functor

context
fixes *a* :: *′a* ⇒ *′a′*
and *b* :: *′b* ⇒ *′b′*
begin

primcorec *map-resource* :: (*′a′*, *′b*) *resource* ⇒ (*′a*, *′b′*) *resource* **where**
run-resource (*map-resource res*) = *map-spmf* (*map-prod b map-resource*) ∘ (*run-resource res*) ∘ *a*

lemma *map-resource-sel* [*simp*]:
run-resource (*map-resource res*) *a′* = *map-spmf* (*map-prod b map-resource*) (*run-resource res* (*a a′*))
by *simp*

declare *map-resource.sel* [*simp del*]

lemma *map-resource-ctr* [*simp*, *code*]:
map-resource (*Resource f*) = *Resource* (*map-spmf* (*map-prod b map-resource*) ∘ *f* ∘ *a*)
by(*rule resource.expand*; *simp add*: *fun-eq-iff*)

end

lemma *map-resource-id1*: *map-resource id f res* = *map-resource′ f res*
by(*coinduction arbitrary*: *res*)(*auto simp add*: *rel-fun-def spmf-rel-map resource.map-sel intro*!: *rel-spmf-reflI*)

lemma *map-resource-id* [*simp*]: *map-resource id id res* = *res*
by(*simp add*: *map-resource-id1 resource.map-id*)

lemma *map-resource-compose* [*simp*]:
map-resource a b (*map-resource a′ b′ res*) = *map-resource* (*a′* ∘ *a*) (*b* ∘ *b′*) *res*
by(*coinduction arbitrary*: *res*)(*auto 4 3 intro*!: *rel-funI rel-spmf-reflI simp add*: *spmf-rel-map*)

functor *resource*: *map-resource* **by**(*simp-all add*: *o-def fun-eq-iff*)

1.3 Relator

coinductive *rel-resource* :: (*′a* ⇒ *′b* ⇒ *bool*) ⇒ (*′c* ⇒ *′d* ⇒ *bool*) ⇒ (*′a*, *′c*) *resource* ⇒ (*′b*, *′d*) *resource* ⇒ *bool*
for *A B* **where**
rel-resourceI:

rel-fun A (*rel-spmf* (*rel-prod B* (*rel-resource A B*))) (*run-resource res1*) (*run-resource res2*)
⟹ *rel-resource A B res1 res2*

lemma *rel-resource-coinduct* [*consumes 1*, *case-names rel-resource*, *coinduct pred*: *rel-resource*]:
assumes *X res1 res2*
and ⋀*res1 res2*. *X res1 res2* ⟹
rel-fun A (*rel-spmf* (*rel-prod B* (λ*res1 res2*. *X res1 res2* ∨ *rel-resource A B res1 res2*)))
(*run-resource res1*) (*run-resource res2*)
shows *rel-resource A B res1 res2*
using *assms*(*1*) **by**(*rule rel-resource.coinduct*)(*simp add*: *assms*(*2*))

lemma *rel-resource-simps* [*simp*, *code*]:
rel-resource A B (*Resource f*) (*Resource g*) ⟷ *rel-fun A* (*rel-spmf* (*rel-prod B* (*rel-resource A B*))) *f g*
by(*subst rel-resource.simps*) *simp*

lemma *rel-resourceD*:
rel-resource A B res1 res2 ⟹ *rel-fun A* (*rel-spmf* (*rel-prod B* (*rel-resource A B*))) (*run-resource res1*) (*run-resource res2*)
by(*blast elim*: *rel-resource.cases*)

lemma *rel-resource-eq1*: *rel-resource* (=) = *rel-resource′*
proof(*intro ext iffI*)
show *rel-resource′ B res1 res2* **if** *rel-resource* (=) *B res1 res2* **for** *B res1 res2*
using *that*
by(*coinduction arbitrary*: *res1 res2*)(*auto elim*: *rel-resource.cases*)
show *rel-resource* (=) *B res1 res2* **if** *rel-resource′ B res1 res2* **for** *B res1 res2*
using *that*
by(*coinduction arbitrary*: *res1 res2*)(*auto 4 4 elim*: *resource.rel-cases intro*: *spmf-rel-mono-strong simp add*: *rel-fun-def*)
qed

lemma *rel-resource-eq*: *rel-resource* (=) (=) = (=)
by(*simp add*: *rel-resource-eq1 resource.rel-eq*)

lemma *rel-resource-mono*:
assumes *A′* ≤ *A B* ≤ *B′*
shows *rel-resource A B* ≤ *rel-resource A′ B′*
proof
show *rel-resource A′ B′ res1 res2* **if** *rel-resource A B res1 res2* **for** *res1 res2*
using *that*
by(*coinduct*)(*auto dest*: *rel-resourceD elim*!: *rel-spmf-mono prod.rel-mono-strong rel-fun-mono intro*: *assms*[*THEN predicate2D*])
qed

lemma *rel-resource-conversep*: *rel-resource* A^{-1-1} B^{-1-1} = (*rel-resource A B*)$^{-1-1}$

proof(*intro ext iffI*; *simp*)
show *rel-resource A B res1 res2* **if** *rel-resource* A^{-1-1} B^{-1-1} *res2 res1*
for $A :: 'a1 \Rightarrow 'a2 \Rightarrow bool$ **and** $B :: 'c1 \Rightarrow 'c2 \Rightarrow bool$ **and** *res1 res2*
using *that* **by**(*coinduct*)
(*drule rel-resourceD*, *rewrite* **in** ⌑ *conversep-iff*[*symmetric*]
, *subst rel-fun-conversep*[*symmetric*], *subst spmf-rel-conversep*[*symmetric*],
erule rel-fun-mono
, *auto simp add*: *prod.rel-conversep*[*symmetric*] *rel-fun-def conversep-iff*[*abs-def*]
elim:*rel-spmf-mono prod.rel-mono-strong*)

from *this*[*of* A^{-1-1} B^{-1-1}]
show *rel-resource* A^{-1-1} B^{-1-1} *res2 res1* **if** *rel-resource A B res1 res2* **for** *res1*
res2 **using** *that* **by** *simp*
qed

lemma *rel-resource-map-resource'1*:
rel-resource A B (*map-resource'* f *res1*) *res2* = *rel-resource A* (λx. B (f x)) *res1*
res2
(**is** *?lhs* = *?rhs*)
proof
show *?rhs* **if** *?lhs* **using** *that*
by(*coinduction arbitrary*: *res1 res2*)
(*drule rel-resourceD*, *auto simp add*: *map-resource.sel map-resource-id1*[*symmetric*]
rel-fun-comp spmf-rel-map prod.rel-map[*abs-def*]
elim!: *rel-fun-mono rel-spmf-mono prod.rel-mono*[*THEN predicate2D*, *rotated*
−*1*])

show *?lhs* **if** *?rhs* **using** *that*
by(*coinduction arbitrary*: *res1 res2*)
(*drule rel-resourceD*, *auto simp add*: *map-resource.sel map-resource-id1*[*symmetric*]
rel-fun-comp spmf-rel-map prod.rel-map[*abs-def*]
elim!: *rel-fun-mono rel-spmf-mono prod.rel-mono*[*THEN predicate2D*, *rotated*
−*1*])
qed

lemma *rel-resource-map-resource'2*:
rel-resource A B res1 (*map-resource'* f *res2*) = *rel-resource A* (λx y. B x (f y))
res1 res2
using *rel-resource-map-resource'1*[*of conversep A conversep B f res2 res1*]
by(*rewrite* **in** ⌑ = - *conversep-iff*[*symmetric*]
, *rewrite* **in** - = ⌑ *conversep-iff*[*symmetric*])
(*simp only*: *rel-resource-conversep*[*symmetric*]
, *simp only*: *conversep-iff*[*abs-def*])

lemmas *resource-rel-map'* = *rel-resource-map-resource'1*[*abs-def*] *rel-resource-map-resource'2*

lemma *rel-resource-pos-distr*:
rel-resource A B OO rel-resource A' B' ≤ *rel-resource* (*A OO A'*) (*B OO B'*)
proof(*rule predicate2I*)

show *rel-resource* (*A OO A'*) (*B OO B'*) *res1 res3*
if (*rel-resource A B OO rel-resource A' B'*) *res1 res3*
for *res1 res3* **using** *that*
apply(*coinduction arbitrary*: *res1 res3*)
apply(*erule relcomppE*)
apply(*drule rel-resourceD*)+
apply(*rule rel-fun-mono*)
apply(*rule pos-fun-distr*[*THEN predicate2D*])
apply(*erule* (*1*) *relcomppI*)
apply *simp*
apply(*rule rel-spmf-mono*)
apply(*erule rel-spmf-pos-distr*[*THEN predicate2D*])
apply(*auto simp add*: *prod.rel-compp*[*symmetric*] *elim*: *prod.rel-mono*[*THEN predicate2D*, *rotated* −*1*])
done
qed

lemma *left-unique-rel-resource*:
⟦ *left-total A*; *left-unique B* ⟧ ⟹ *left-unique* (*rel-resource A B*)
unfolding *left-unique-alt-def left-total-alt-def rel-resource-conversep*[*symmetric*]
apply(*subst rel-resource-eq*[*symmetric*])
apply(*rule order-trans*[*OF rel-resource-pos-distr*])
apply(*erule* (*1*) *rel-resource-mono*)
done

lemma *right-unique-rel-resource*:
⟦ *right-total A*; *right-unique B* ⟧ ⟹ *right-unique* (*rel-resource A B*)
unfolding *right-unique-alt-def right-total-alt-def rel-resource-conversep*[*symmetric*]
apply(*subst rel-resource-eq*[*symmetric*])
apply(*rule order-trans*[*OF rel-resource-pos-distr*])
apply(*erule* (*1*) *rel-resource-mono*)
done

lemma *bi-unique-rel-resource* [*transfer-rule*]:
⟦ *bi-total A*; *bi-unique B* ⟧ ⟹ *bi-unique* (*rel-resource A B*)
unfolding *bi-unique-alt-def bi-total-alt-def* **by**(*blast intro*: *left-unique-rel-resource right-unique-rel-resource*)

definition *rel-witness-resource* :: ('*a* ⇒ '*e* ⇒ *bool*) ⇒ ('*e* ⇒ '*c* ⇒ *bool*) ⇒ ('*b* ⇒ '*d* ⇒ *bool*) ⇒ ('*a*, '*b*) *resource* × ('*c*, '*d*) *resource* ⇒ ('*e*, '*b* × '*d*) *resource* **where**
rel-witness-resource A A' B = *corec-resource* (λ(*res1*, *res2*).
map-spmf (*map-prod id Inr* ∘ *rel-witness-prod*) ∘
rel-witness-spmf (*rel-prod B* (*rel-resource* (*A OO A'*) *B*)) ∘
rel-witness-fun A A' (*run-resource res1*, *run-resource res2*))

lemma *rel-witness-resource-sel* [*simp*]:
run-resource (*rel-witness-resource A A' B* (*res1*, *res2*)) =
map-spmf (*map-prod id* (*rel-witness-resource A A' B*) ∘ *rel-witness-prod*) ∘

rel-witness-spmf (*rel-prod B* (*rel-resource* (*A OO A'*) *B*)) ∘
rel-witness-fun A A' (*run-resource res1*, *run-resource res2*)
by(*auto simp add*: *rel-witness-resource-def o-def fun-eq-iff spmf.map-comp intro*!: *map-spmf-cong*)

lemma assumes *rel-resource* (*A OO A'*) *B res res'*
and *A*: *left-unique A right-total A*
and *A'*: *right-unique A' left-total A'*
shows *rel-witness-resource1*: *rel-resource A* (λ*b* (*b'*, *c*). *b* = *b'* ∧ *B b' c*) *res* (*rel-witness-resource A A' B* (*res*, *res'*)) (**is** *?thesis1*)
and *rel-witness-resource2*: *rel-resource A'* (λ(*b*, *c'*) *c*. *c* = *c'* ∧ *B b c'*) (*rel-witness-resource A A' B* (*res*, *res'*)) *res'* (**is** *?thesis2*)
proof −
show *?thesis1* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *res res'*)
case *rel-resource*
from *this*[*THEN rel-resourceD*] **show** *?case*
by(*simp add*: *rel-fun-comp*)
(*erule rel-fun-mono*[*OF rel-witness-fun1*[*OF - A A'*]]
, *auto simp add*: *spmf-rel-map elim*!: *rel-spmf-mono*[*OF rel-witness-spmf1*])
qed
show *?thesis2* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *res res'*)
case *rel-resource*
from *this*[*THEN rel-resourceD*] **show** *?case*
by(*simp add*: *rel-fun-comp*)
(*erule rel-fun-mono*[*OF rel-witness-fun2*[*OF - A A'*]]
, *auto simp add*: *spmf-rel-map elim*!: *rel-spmf-mono*[*OF rel-witness-spmf2*])
qed
qed

lemma *rel-resource-neg-distr*:
assumes *A*: *left-unique A right-total A*
and *A'*: *right-unique A' left-total A'*
shows *rel-resource* (*A OO A'*) (*B OO B'*) ≤ *rel-resource A B OO rel-resource A' B'*
proof(*rule predicate2I relcomppI*)+
fix *res res''*
assume ∗: *rel-resource* (*A OO A'*) (*B OO B'*) *res res''*
let *?res'* = *map-resource'* (*relcompp-witness B B'*) (*rel-witness-resource A A'* (*B OO B'*) (*res*, *res''*))
show *rel-resource A B res ?res'* **using** *rel-witness-resource1*[*OF* ∗ *A A'*] **unfolding** *resource-rel-map'*
by(*rule rel-resource-mono*[*THEN predicate2D*, *rotated* −*1*]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)
show *rel-resource A' B' ?res' res''* **using** *rel-witness-resource2*[*OF* ∗ *A A'*] **unfolding** *resource-rel-map'*
by(*rule rel-resource-mono*[*THEN predicate2D*, *rotated* −*1*]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)

qed

lemma *left-total-rel-resource*:
⟦ *left-unique A*; *right-total A*; *left-total B* ⟧ ⟹ *left-total* (*rel-resource A B*)
unfolding *left-unique-alt-def left-total-alt-def rel-resource-conversep*[*symmetric*]
apply(*subst rel-resource-eq*[*symmetric*])
apply(*rule order-trans*[*rotated*])
apply(*rule rel-resource-neg-distr*; *simp add*: *left-unique-alt-def*)
apply(*rule rel-resource-mono*; *assumption*)
done

lemma *right-total-rel-resource*:
⟦ *right-unique A*; *left-total A*; *right-total B* ⟧ ⟹ *right-total* (*rel-resource A B*)
unfolding *right-unique-alt-def right-total-alt-def rel-resource-conversep*[*symmetric*]
apply(*subst rel-resource-eq*[*symmetric*])
apply(*rule order-trans*[*rotated*])
apply(*rule rel-resource-neg-distr*; *simp add*: *right-unique-alt-def*)
apply(*rule rel-resource-mono*; *assumption*)
done

lemma *bi-total-rel-resource* [*transfer-rule*]:
⟦ *bi-total A*; *bi-unique A*; *bi-total B* ⟧ ⟹ *bi-total* (*rel-resource A B*)
unfolding *bi-total-alt-def bi-unique-alt-def*
by(*blast intro*: *left-total-rel-resource right-total-rel-resource*)

context includes *lifting-syntax* **begin**

lemma *Resource-parametric* [*transfer-rule*]:
(*A* ===> *rel-spmf* (*rel-prod B* (*rel-resource A B*))) ===> *rel-resource A B*)
Resource Resource
by(*rule rel-funI*)(*simp*)

lemma *run-resource-parametric* [*transfer-rule*]:
(*rel-resource A B* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-resource A B*)))
run-resource run-resource
by(*rule rel-funI*)(*auto dest*: *rel-resourceD*)

lemma *corec-resource-parametric* [*transfer-rule*]:
((*S* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-sum* (*rel-resource A B*) *S*)))
===> *S* ===> *rel-resource A B*)
corec-resource corec-resource
proof((*rule rel-funI*)+, *goal-cases*)
case (*1 f g s1 s2*)
then show *?case* **using** *1*(*2*)
by (*coinduction arbitrary*: *s1 s2*)
(*drule 1*(*1*)[*THEN rel-funD*], *auto 4 4 simp add*: *rel-fun-comp spmf-rel-map prod.rel-map*[*abs-def*] *split*: *sum.split elim*!: *rel-fun-mono rel-spmf-mono elim*: *prod.rel-mono*[*THEN predicate2D*, *rotated* −*1*])
qed

lemma *map-resource-parametric* [*transfer-rule*]:
((A' ===> A) ===> (B ===> B') ===> *rel-resource* A B ===> *rel-resource* A' B') *map-resource map-resource*
unfolding *map-resource-def* **by**(*transfer-prover*)

lemma *map-resource'-parametric* [*transfer-rule*]:
((B ===> B') ===> *rel-resource* (=) B ===> *rel-resource* (=) B') *map-resource' map-resource'*
unfolding *map-resource-id1*[*symmetric*] **by** *transfer-prover*

lemma *case-resource-parametric* [*transfer-rule*]:
(((A ===> *rel-spmf* (*rel-prod* B (*rel-resource* A B))) ===> C) ===> *rel-resource* A B ===> C)
case-resource case-resource
unfolding *case-resource-conv-run-resource* **by** *transfer-prover*

end

lemma *rel-resource-Grp*:
rel-resource (*conversep* (*BNF-Def.Grp UNIV* f)) (*BNF-Def.Grp UNIV* g) = *BNF-Def.Grp UNIV* (*map-resource* f g)
proof((*rule ext iffI*)+, *goal-cases*)
case (*1 res res'*)
have ∗: *rel-resource* $(\lambda a\ b.\ b = f\ a)^{-1-1}$ $(\lambda a\ b.\ b = g\ a)$ *res res'* ⟹ *res'* = *map-resource f g res*
by(*rule sym*, *subst* (*3*) *map-resource-id*[*symmetric*], *subst rel-resource-eq*[*symmetric*])
(*erule map-resource-parametric*[*THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*, *rotated* −*1*]
, *auto simp add*: *rel-fun-def*)

from *1* **show** *?case* **unfolding** *Grp-def* **using** ∗ **by** (*clarsimp simp add*: ∗ *simp del*: *conversep-iff*)
next
case (*2* - -)
then show *?case*
by(*clarsimp simp add*: *Grp-iff*, *subst map-resource-id*[*symmetric*])
(*rule map-resource-parametric*[*THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*, *rotated* −*1*]
, *subst rel-resource-eq*, *auto simp add*: *Grp-iff rel-fun-def*)
qed

1.4 Losslessness

coinductive *lossless-resource* :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*a*, '*b*) *resource* ⇒ *bool*
for $\mathcal{I}$ **where**
lossless-resourceI: *lossless-resource* $\mathcal{I}$ *res* **if**
⋀a. a ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-spmf* (*run-resource res a*)
⋀a b *res'*. ⟦ a ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; (b, *res'*) ∈ *set-spmf* (*run-resource res a*) ⟧ ⟹

lossless-resource $\mathcal{I}$ *res'*

lemma *lossless-resource-coinduct* [*consumes 1*, *case-names lossless-resource*, *case-conclusion lossless-resource lossless step*, *coinduct pred*: *lossless-resource*]:
assumes *X res*
and $\bigwedge$*res a*. ⟦ *X res*; *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *lossless-spmf* (*run-resource res a*) ∧ (∀(*b*, *res'*) ∈ *set-spmf* (*run-resource res a*). *X res'* ∨ *lossless-resource* $\mathcal{I}$ *res'*)
shows *lossless-resource* $\mathcal{I}$ *res*
using *assms*(*1*) **by**(*rule lossless-resource.coinduct*)(*auto dest*: *assms*(*2*))

lemma *lossless-resourceD*:
⟦ *lossless-resource* $\mathcal{I}$ *res*; *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ *lossless-spmf* (*run-resource res a*) ∧ (∀(*x*, *res'*)∈*set-spmf* (*run-resource res a*). *lossless-resource* $\mathcal{I}$ *res'*)
by(*auto elim*: *lossless-resource.cases*)

lemma *lossless-resource-mono*:
assumes *lossless-resource* $\mathcal{I}'$ *res*
and *le*: *outs-*$\mathcal{I}$ $\mathcal{I}$ ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}'$
shows *lossless-resource* $\mathcal{I}$ *res*
using *assms*(*1*)
by(*coinduction arbitrary*: *res*)(*auto dest*: *lossless-resourceD intro*: *subsetD*[*OF le*])

lemma *lossless-resource-mono'*:
⟦ *lossless-resource* $\mathcal{I}'$ *res*; $\mathcal{I}$ ≤ $\mathcal{I}'$ ⟧ ⟹ *lossless-resource* $\mathcal{I}$ *res*
by(*erule lossless-resource-mono*)(*simp add*: *le-*$\mathcal{I}$*-def*)

1.5 Operations

context fixes *oracle* :: *'s* ⇒ *'a* ⇒ (*'b* × *'s*) *spmf* **begin**

primcorec *resource-of-oracle* :: *'s* ⇒ (*'a*, *'b*) *resource* **where**
run-resource (*resource-of-oracle s*) = (λ*a*. *map-spmf* (*map-prod id resource-of-oracle*) (*oracle s a*))

end

lemma *resource-of-oracle-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *A* ===> *rel-spmf* (*rel-prod B S*)) ===> *S* ===> *rel-resource A B*) *resource-of-oracle resource-of-oracle*
unfolding *resource-of-oracle-def* **by** *transfer-prover*

lemma *map-resource-resource-of-oracle*:
map-resource f g (*resource-of-oracle oracle s*) = *resource-of-oracle* (*map-fun id* (*map-fun f* (*map-spmf* (*map-prod g id*))) *oracle*) *s*
for *s* :: *'s*
using *resource-of-oracle-parametric*[*of BNF-Def.Grp UNIV* (*id* :: *'s* ⇒ -) *con-*

versep (*BNF-Def.Grp UNIV f*) *BNF-Def.Grp UNIV g*]
unfolding *prod.rel-Grp option.rel-Grp pmf.rel-Grp rel-fun-Grp rel-resource-Grp*
by *simp*
(*subst* (*asm*) (*1 2*) *eq-alt*[*symmetric*]
, *subst* (*asm*) (*1 2*) *conversep-eq*[*symmetric*]
, *subst* (*asm*) (*1 2*) *eq-alt*
, *unfold rel-fun-Grp*, *simp add*: *rel-fun-Grp rel-fun-def Grp-iff*)

lemma (**in** *callee-invariant-on*) *lossless-resource-of-oracle*:
assumes ∗: ⋀*s x*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *lossless-spmf* (*callee s x*)
and *I s*
shows *lossless-resource* $\mathcal{I}$ (*resource-of-oracle callee s*)
using ‹*I s*› **by**(*coinduction arbitrary*: *s*)(*auto intro*: ∗ *dest*: *callee-invariant*)

context includes *lifting-syntax* **begin**

lemma *resource-of-oracle-rprodl*: **includes** *lifting-syntax* **shows**
resource-of-oracle ((*rprodl* −−−> *id* −−−> *map-spmf* (*map-prod id lprodr*)) *oracle*) ((*s1*, *s2*), *s3*) =
resource-of-oracle oracle (*s1*, *s2*, *s3*)
by(*rule resource-of-oracle-parametric*[*of BNF-Def.Grp UNIV rprodl* (=) (=), *THEN rel-funD*, *THEN rel-funD*, *unfolded rel-resource-eq*])
(*auto simp add*: *Grp-iff rel-fun-def spmf-rel-map intro*!: *rel-spmf-reflI*)

lemma *resource-of-oracle-extend-state-oracle* [*simp*]:
resource-of-oracle (*extend-state-oracle oracle*) (*s'*, *s*) = *resource-of-oracle oracle s*
by(*rule resource-of-oracle-parametric*[*of conversep* (*BNF-Def.Grp UNIV* (λ*s*. (*s'*, *s*))) (=) (=), *THEN rel-funD*, *THEN rel-funD*, *unfolded rel-resource-eq*])
(*auto simp add*: *Grp-iff rel-fun-def spmf-rel-map intro*!: *rel-spmf-reflI*)

end

lemma *exec-gpv-resource-of-oracle*:
exec-gpv run-resource gpv (*resource-of-oracle oracle s*) = *map-spmf* (*map-prod id* (*resource-of-oracle oracle*)) (*exec-gpv oracle gpv s*)
by(*subst spmf.map-id*[*symmetric*], *fold pmf.rel-eq*)
(*rule pmf.map-transfer*[*THEN rel-funD*, *THEN rel-funD*, *rotated*]
, *rule exec-gpv-parametric*[**where** *S*=λ*res s*. *res* = *resource-of-oracle oracle s* **and** *CALL*=(=) **and** *A*=(=), *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*]
, *auto simp add*: *gpv.rel-eq rel-fun-def spmf-rel-map elim*: *option.rel-cases intro*!: *rel-spmf-reflI*)

primcorec *parallel-resource* :: (*'a*, *'b*) *resource* ⇒ (*'c*, *'d*) *resource* ⇒ (*'a* + *'c*, *'b* + *'d*) *resource* **where**
run-resource (*parallel-resource res1 res2*) =
(λ*ac*. *case ac of Inl a* ⇒ *map-spmf* (*map-prod Inl* (λ*res1'*. *parallel-resource res1' res2*)) (*run-resource res1 a*)
| *Inr c* ⇒ *map-spmf* (*map-prod Inr* (λ*res2'*. *parallel-resource res1 res2'*))

(*run-resource res2 c*))

lemma *parallel-resource-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-resource A B ===> rel-resource C D ===> rel-resource* (*rel-sum A C*) (*rel-sum B D*))
parallel-resource parallel-resource
unfolding *parallel-resource-def* **by** *transfer-prover*

We cannot define the analogue of $(\oplus_O)$ because we no longer have access to the state, so state sharing is not possible! So we can only compose resources, but we cannot build one resource with several interfaces this way!

lemma *resource-of-parallel-oracle*:
resource-of-oracle (*parallel-oracle oracle1 oracle2*) (*s1*, *s2*) =
parallel-resource (*resource-of-oracle oracle1 s1*) (*resource-of-oracle oracle2 s2*)
by(*coinduction arbitrary*: *s1 s2*)
(*auto 4 3 simp add*: *rel-fun-def spmf-rel-map split*: *sum.split intro*!: *rel-spmf-reflI*)

lemma *parallel-resource-assoc*: — There's still an ugly map operation in there to rebalance the interface trees, but well...
parallel-resource (*parallel-resource res1 res2*) *res3* =
map-resource rsuml lsumr (*parallel-resource res1* (*parallel-resource res2 res3*))
by(*coinduction arbitrary*: *res1 res2 res3*)
(*auto 4 5 intro*!: *rel-funI rel-spmf-reflI simp add*: *spmf-rel-map split*: *sum.split*)

lemma *lossless-parallel-resource*:
assumes *lossless-resource* $\mathcal{I}$ *res1 lossless-resource* $\mathcal{I}'$ *res2*
shows *lossless-resource* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*parallel-resource res1 res2*)
using *assms*
by(*coinduction arbitrary*: *res1 res2*)(*clarsimp*; *erule PlusE*; *simp*; *frule* (*1*) *lossless-resourceD*; *auto 4 3*)

1.6 Well-typing

coinductive *WT-resource* :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*a*, '*b*) *resource* ⇒ *bool* (- /⊢*res* - √ [*100*, *0*] *99*)
for $\mathcal{I}$ **where**
WT-resourceI: $\mathcal{I}$ ⊢*res res* √
if ⋀*q r res'*. ⟦ *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; (*r*, *res'*) ∈ *set-spmf* (*run-resource res q*) ⟧ ⟹ *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* ∧ $\mathcal{I}$ ⊢*res res'* √

lemma *WT-resource-coinduct* [*consumes 1*, *case-names WT-resource*, *case-conclusion WT-resource response WT-resource*, *coinduct pred*: *WT-resource*]:
assumes *X res*
and ⋀*res q r res'*. ⟦ *X res*; *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; (*r*, *res'*) ∈ *set-spmf* (*run-resource res q*) ⟧
⟹ *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* ∧ (*X res'* ∨ $\mathcal{I}$ ⊢*res res'* √)
shows $\mathcal{I}$ ⊢*res res* √
using *assms*(*1*) **by**(*rule WT-resource.coinduct*)(*blast dest*: *assms*(*2*))

lemma *WT-resourceD*:
assumes $\mathcal{I} \vdash_{res} res \surd$ $q \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $(r, res') \in$ *set-spmf* (*run-resource res q*)
shows $r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* $\wedge$ $\mathcal{I} \vdash_{res} res' \surd$
using *assms* **by**(*auto elim*: *WT-resource.cases*)

lemma *WT-resource-of-oracle* [*simp*]:
assumes $\bigwedge s.$ $\mathcal{I} \vdash_c$ *oracle s* $\surd$
shows $\mathcal{I} \vdash_{res}$ *resource-of-oracle oracle s* $\surd$
by(*coinduction arbitrary*: *s*)(*auto dest*: *WT-calleeD*[*OF assms*])

lemma *WT-resource-bot* [*simp*]: *bot* $\vdash_{res}$ *res* $\surd$
by(*rule WT-resource.intros*)*auto*

lemma *WT-resource-full*: $\mathcal{I}$*-full* $\vdash_{res}$ *res* $\surd$
by(*coinduction arbitrary*: *res*)(*auto*)

lemma (**in** *callee-invariant-on*) *WT-resource-of-oracle*:
I s $\Longrightarrow$ $\mathcal{I} \vdash_{res}$ *resource-of-oracle callee s* $\surd$
by(*coinduction arbitrary*: *s*)(*auto dest*: *callee-invariant'*)

named-theorems *WT-intro Interface typing introduction rules*

lemmas [*WT-intro*] = *WT-gpv-map-gpv' WT-gpv-map-gpv*

lemma *WT-parallel-resource* [*WT-intro*]:
assumes $\mathcal{I}1 \vdash_{res}$ *res1* $\surd$
and $\mathcal{I}2 \vdash_{res}$ *res2* $\surd$
shows $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_{res}$ *parallel-resource res1 res2* $\surd$
using *assms*
by(*coinduction arbitrary*: *res1 res2*)(*auto 4 4 intro*!: *imageI dest*: *WT-resourceD*)

lemma *callee-invariant-run-resource*: *callee-invariant-on run-resource* (λ*res*. $\mathcal{I}$ $\vdash_{res}$ *res* $\surd$) $\mathcal{I}$
by(*unfold-locales*)(*auto dest*: *WT-resourceD intro*: *WT-calleeI*)

lemma *callee-invariant-run-lossless-resource*:
callee-invariant-on run-resource (λ*res*. *lossless-resource* $\mathcal{I}$ *res* $\wedge$ $\mathcal{I} \vdash_{res}$ *res* $\surd$) $\mathcal{I}$
by(*unfold-locales*)(*auto dest*: *WT-resourceD lossless-resourceD intro*: *WT-calleeI*)

interpretation *run-lossless-resource*:
callee-invariant-on run-resource λ*res*. *lossless-resource* $\mathcal{I}$ *res* $\wedge$ $\mathcal{I} \vdash_{res}$ *res* $\surd$ $\mathcal{I}$
for $\mathcal{I}$
by(*rule callee-invariant-run-lossless-resource*)

end
theory *Converter* **imports**
Resource
begin

2 Converters

2.1 Type definition

codatatype (*'a*, *results'-converter*: *'b*, *outs'-converter*: *'out*, *'in*) *converter*
= *Converter* (*run-converter*: *'a* ⇒ (*'b* × (*'a*, *'b*, *'out*, *'in*) *converter*, *'out*, *'in*) *gpv*)
for *map*: *map-converter'*
rel: *rel-converter'*
pred: *pred-converter'*

lemma *case-converter-conv-run-converter*: *case-converter f conv* = *f* (*run-converter conv*)
by(*fact converter.case-eq-if*)

2.2 Functor

context
fixes *a* :: *'a* ⇒ *'a'*
and *b* :: *'b* ⇒ *'b'*
and *out* :: *'out* ⇒ *'out'*
and *inn* :: *'in* ⇒ *'in'*
begin

primcorec *map-converter* :: (*'a'*, *'b*, *'out*, *'in'*) *converter* ⇒ (*'a*, *'b'*, *'out'*, *'in*) *converter* **where**
run-converter (*map-converter conv*) =
map-gpv (*map-prod b map-converter*) *out* ∘ *map-gpv'* *id id inn* ∘ *run-converter conv* ∘ *a*

lemma *map-converter-sel* [*simp*]:
run-converter (*map-converter conv*) *a'* = *map-gpv'* (*map-prod b map-converter*) *out inn* (*run-converter conv* (*a a'*))
by(*simp add*: *map-gpv-conv-map-gpv' map-gpv'-comp*)

declare *map-converter.sel* [*simp del*]

lemma *map-converter-ctr* [*simp*, *code*]:
map-converter (*Converter f*) = *Converter* (*map-fun a* (*map-gpv'* (*map-prod b map-converter*) *out inn*) *f*)
by(*rule converter.expand*; *simp add*: *fun-eq-iff*)

end

lemma *map-converter-id14*: *map-converter id b out id res* = *map-converter' b out res*
by(*coinduction arbitrary*: *res*)
(*auto 4 3 intro*!: *rel-funI simp add*: *converter.map-sel gpv.rel-map map-gpv-conv-map-gpv'*[*symmetric*] *intro*!: *gpv.rel-refl-strong*)

lemma *map-converter-id* [*simp*]: *map-converter id id id id conv* = *conv*
by(*simp add*: *map-converter-id14 converter.map-id*)

lemma *map-converter-compose* [*simp*]:
map-converter a b f g (*map-converter a' b' f' g' conv*) = *map-converter* (*a'* ∘ *a*)
(*b* ∘ *b'*) (*f* ∘ *f'*) (*g'* ∘ *g*) *conv*
by(*coinduction arbitrary*: *conv*)
(*auto 4 3 intro*!: *rel-funI gpv.rel-refl-strong simp add*: *rel-gpv-map-gpv' map-gpv'-comp o-def prod.map-comp*)

functor *converter*: *map-converter* **by**(*simp-all add*: *o-def fun-eq-iff*)

2.3 Set functions with interfaces

context fixes $\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ **and** $\mathcal{I}'$ :: ('*out*, '*in*) $\mathcal{I}$ **begin**

qualified inductive *outsp-converter* :: '*out* ⇒ ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ *bool*
for *out* **where**
Out: *outsp-converter out conv* **if** *out* ∈ *outs-gpv* $\mathcal{I}'$ (*run-converter conv a*) *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$
| *Cont*: *outsp-converter out conv*
if (*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) *outsp-converter out conv' a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$

definition *outs-converter* :: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ '*out set*
where *outs-converter conv* ≡ {*x*. *outsp-converter x conv*}

qualified inductive *resultsp-converter* :: '*b* ⇒ ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ *bool*
for *b* **where**
Result: *resultsp-converter b conv*
if (*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$
| *Cont*: *resultsp-converter b conv*
if (*b'*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) *resultsp-converter b conv' a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$

definition *results-converter* :: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ '*b set*
where *results-converter conv* = {*b*. *resultsp-converter b conv*}

end

lemma *outsp-converter-outs-converter-eq* [*pred-set-conv*]: *Converter.outsp-converter* $\mathcal{I}$ $\mathcal{I}'$ *x* = (λ*conv*. *x* ∈ *outs-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*)
by(*simp add*: *outs-converter-def*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path outs-converter*)›

lemmas *intros* [*intro?*] = *outsp-converter.intros*[*to-set*]

and *Out* = *outsp-converter.Out*[*to-set*]
and *Cont* = *outsp-converter.Cont*[*to-set*]
and *induct* [*consumes 1*, *case-names Out Cont*, *induct set*: *outs-converter*] = *outsp-converter.induct*[*to-set*]
and *cases* [*consumes 1*, *case-names Out Cont*, *cases set*: *outs-converter*] = *outsp-converter.cases*[*to-set*]
and *simps* = *outsp-converter.simps*[*to-set*]
end

inductive-simps *outs-converter-Converter* [*to-set*, *simp*]: *Converter.outsp-converter* $\mathcal{I}$ $\mathcal{I}'$ *x* (*Converter conv*)

lemma *resultsp-converter-results-converter-eq* [*pred-set-conv*]:
Converter.resultsp-converter $\mathcal{I}$ $\mathcal{I}'$ *x* = (λ*conv*. *x* ∈ *results-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*)
by(*simp add*: *results-converter-def*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path results-converter*)›

lemmas *intros* [*intro?*] = *resultsp-converter.intros*[*to-set*]
and *Result* = *resultsp-converter.Result*[*to-set*]
and *Cont* = *resultsp-converter.Cont*[*to-set*]
and *induct* [*consumes 1*, *case-names Result Cont*, *induct set*: *results-converter*] = *resultsp-converter.induct*[*to-set*]
and *cases* [*consumes 1*, *case-names Result Cont*, *cases set*: *results-converter*] = *resultsp-converter.cases*[*to-set*]
and *simps* = *resultsp-converter.simps*[*to-set*]
end

inductive-simps *results-converter-Converter* [*to-set*, *simp*]: *Converter.resultsp-converter* $\mathcal{I}$ $\mathcal{I}'$ *x* (*Converter conv*)

2.4 Relator

coinductive *rel-converter*
:: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*c* ⇒ '*d* ⇒ *bool*) ⇒ ('*out* ⇒ '*out'* ⇒ *bool*) ⇒ ('*in* ⇒ '*in'* ⇒ *bool*)
⇒ ('*a*, '*c*, '*out*, '*in*) *converter* ⇒ ('*b*, '*d*, '*out'*, '*in'*) *converter* ⇒ *bool*
for *A B C R* **where**
rel-converterI:
rel-fun A (*rel-gpv''* (*rel-prod B* (*rel-converter A B C R*)) *C R*) (*run-converter conv1*) (*run-converter conv2*)
⟹ *rel-converter A B C R conv1 conv2*

lemma *rel-converter-coinduct* [*consumes 1*, *case-names rel-converter*, *coinduct pred*: *rel-converter*]:
assumes *X conv1 conv2*
and ⋀*conv1 conv2*. *X conv1 conv2* ⟹

rel-fun A (*rel-gpv''* (*rel-prod B* (λ*conv1 conv2*. *X conv1 conv2* ∨ *rel-converter A B C R conv1 conv2*)) *C R*)
(*run-converter conv1*) (*run-converter conv2*)
shows *rel-converter A B C R conv1 conv2*
using *assms*(*1*) **by**(*rule rel-converter.coinduct*)(*simp add*: *assms*(*2*))

lemma *rel-converter-simps* [*simp*, *code*]:
rel-converter A B C R (*Converter f*) (*Converter g*) ⟷
rel-fun A (*rel-gpv''* (*rel-prod B* (*rel-converter A B C R*)) *C R*) *f g*
by(*subst rel-converter.simps*) *simp*

lemma *rel-converterD*:
rel-converter A B C R conv1 conv2
⟹ *rel-fun A* (*rel-gpv''* (*rel-prod B* (*rel-converter A B C R*)) *C R*) (*run-converter conv1*) (*run-converter conv2*)
by(*blast elim*: *rel-converter.cases*)

lemma *rel-converter-eq14*: *rel-converter* (=) *B C* (=) = *rel-converter'* *B C* (**is** *?lhs* = *?rhs*)
proof(*intro ext iffI*)
show *?rhs conv1 conv2* **if** *?lhs conv1 conv2* **for** *conv1 conv2* **using** *that*
by(*coinduction arbitrary*: *conv1 conv2*)(*auto elim*: *rel-converter.cases simp add*: *rel-gpv-conv-rel-gpv''*)
show *?lhs conv1 conv2* **if** *?rhs conv1 conv2* **for** *conv1 conv2* **using** *that*
by(*coinduction arbitrary*: *conv1 conv2*)
(*auto 4 4 elim*: *converter.rel-cases intro*: *gpv.rel-mono-strong simp add*: *rel-fun-def rel-gpv-conv-rel-gpv''*[*symmetric*])
qed

lemma *rel-converter-eq* [*relator-eq*]: *rel-converter* (=) (=) (=) (=) = (=)
by(*simp add*: *rel-converter-eq14 converter.rel-eq*)

lemma *rel-converter-mono* [*relator-mono*]:
assumes *A'* ≤ *A B* ≤ *B' C* ≤ *C' R'* ≤ *R*
shows *rel-converter A B C R* ≤ *rel-converter A' B' C' R'*
proof
show *rel-converter A' B' C' R' conv1 conv2* **if** *rel-converter A B C R conv1 conv2* **for** *conv1 conv2* **using** *that*
by(*coinduct*)(*auto dest*: *rel-converterD elim*!: *rel-gpv''-mono*[*THEN predicate2D*, *rotated* −*1*] *prod.rel-mono-strong rel-fun-mono intro*: *assms*[*THEN predicate2D*])
qed

lemma *rel-converter-conversep*: *rel-converter* A^{-1-1} B^{-1-1} C^{-1-1} R^{-1-1} = (*rel-converter A B C R*)$^{-1-1}$
proof(*intro ext iffI*; *simp*)
show *rel-converter A B C R conv1 conv2* **if** *rel-converter* A^{-1-1} B^{-1-1} C^{-1-1} R^{-1-1} *conv2 conv1*
for *A* :: *'a1* ⇒ *'a2* ⇒ *bool* **and** *B* :: *'b1* ⇒ *'b2* ⇒ *bool* **and** *C* :: *'c1* ⇒ *'c2* ⇒ *bool* **and** *R* :: *'r1* ⇒ *'r2* ⇒ *bool*

```
    and conv2 conv1
  using that apply(coinduct)
  apply(drule rel-converterD)
  apply(rewrite in ⌑ conversep-iff[symmetric])
  apply(subst rel-fun-conversep[symmetric])
  apply(subst rel-gpv''-conversep[symmetric])
  apply(erule rel-fun-mono, blast)
 by(auto simp add: prod.rel-conversep[symmetric] rel-fun-def conversep-iff[abs-def]
     elim: prod.rel-mono-strong rel-gpv''-mono[THEN predicate2D, rotated -1])

 from this[of A⁻¹⁻¹ B⁻¹⁻¹ C⁻¹⁻¹ R⁻¹⁻¹]
 show rel-converter A⁻¹⁻¹ B⁻¹⁻¹ C⁻¹⁻¹ R⁻¹⁻¹ conv2 conv1 if rel-converter A
B C R conv1 conv2 for conv1 conv2
  using that by simp
qed

lemma rel-converter-map-converter'1:
 rel-converter A B C R (map-converter' f g conv1) conv2 = rel-converter A (λx.
B (f x)) (λx. C (g x)) R conv1 conv2
 (is ?lhs = ?rhs)
proof
 show ?rhs if ?lhs using that
  by(coinduction arbitrary: conv1 conv2)
  (drule rel-converterD, auto intro: prod.rel-mono elim!: rel-fun-mono rel-gpv''-mono[THEN
predicate2D, rotated -1]
    simp add: map-gpv'-id rel-gpv''-map-gpv map-converter.sel map-converter-id1⁴[symmetric]
rel-fun-comp spmf-rel-map prod.rel-map[abs-def])
 show ?lhs if ?rhs using that
  by(coinduction arbitrary: conv1 conv2)
  (drule rel-converterD, auto intro: prod.rel-mono elim!: rel-fun-mono rel-gpv''-mono[THEN
predicate2D, rotated -1]
    simp add: map-gpv'-id rel-gpv''-map-gpv map-converter.sel map-converter-id1⁴[symmetric]
rel-fun-comp spmf-rel-map prod.rel-map[abs-def])
qed

lemma rel-converter-map-converter'2:
 rel-converter A B C R conv1 (map-converter' f g conv2) = rel-converter A (λx
y. B x (f y)) (λx y. C x (g y)) R conv1 conv2
 using rel-converter-map-converter'1[of conversep A conversep B conversep C
conversep R f g conv2 conv1]
 apply(rewrite in ⌑ = - conversep-iff[symmetric])
 apply(rewrite in - = ⌑ conversep-iff[symmetric])
 apply(simp only: rel-converter-conversep[symmetric])
 apply(simp only: conversep-iff[abs-def])
 done

lemmas converter-rel-map' = rel-converter-map-converter'1[abs-def] rel-converter-map-converter'2

lemma rel-converter-pos-distr [relator-distr]:
```

rel-converter A B C R OO rel-converter A′ B′ C′ R′ ≤ *rel-converter* (*A OO A′*) (*B OO B′*) (*C OO C′*) (*R OO R′*)
proof(*rule predicate2I*)
show *rel-converter* (*A OO A′*) (*B OO B′*) (*C OO C′*) (*R OO R′*) *conv1 conv3*
if (*rel-converter A B C R OO rel-converter A′ B′ C′ R′*) *conv1 conv3*
for *conv1 conv3* **using** *that*
apply(*coinduction arbitrary*: *conv1 conv3*)
apply(*erule relcomppE*)
apply(*drule rel-converterD*)+
apply(*rule rel-fun-mono*)
apply(*rule pos-fun-distr*[*THEN predicate2D*])
apply(*erule* (*1*) *relcomppI*)
apply *simp*
apply(*rule rel-gpv′′-mono*[*THEN predicate2D*, *rotated* −*1*])
apply(*erule rel-gpv′′-pos-distr*[*THEN predicate2D*])
by(*auto simp add*: *prod.rel-compp*[*symmetric*] *intro*: *prod.rel-mono*)
qed

lemma *left-unique-rel-converter*:
⟦ *left-total A*; *left-unique B*; *left-unique C*; *left-total R* ⟧ ⟹ *left-unique* (*rel-converter A B C R*)
unfolding *left-unique-alt-def left-total-alt-def rel-converter-conversep*[*symmetric*]
by(*subst rel-converter-eq*[*symmetric*], *rule order-trans*[*OF rel-converter-pos-distr*], *erule* (*3*) *rel-converter-mono*)

lemma *right-unique-rel-converter*:
⟦ *right-total A*; *right-unique B*; *right-unique C*; *right-total R* ⟧ ⟹ *right-unique* (*rel-converter A B C R*)
unfolding *right-unique-alt-def right-total-alt-def rel-converter-conversep*[*symmetric*]
by(*subst rel-converter-eq*[*symmetric*], *rule order-trans*[*OF rel-converter-pos-distr*], *erule* (*3*) *rel-converter-mono*)

lemma *bi-unique-rel-converter* [*transfer-rule*]:
⟦ *bi-total A*; *bi-unique B*; *bi-unique C*; *bi-total R* ⟧ ⟹ *bi-unique* (*rel-converter A B C R*)
unfolding *bi-unique-alt-def bi-total-alt-def* **by**(*blast intro*: *left-unique-rel-converter right-unique-rel-converter*)

definition *rel-witness-converter* :: (′*a* ⇒ ′*e* ⇒ *bool*) ⇒ (′*e* ⇒ ′*c* ⇒ *bool*) ⇒ (′*b* ⇒ ′*d* ⇒ *bool*) ⇒ (′*out* ⇒ ′*out′* ⇒ *bool*) ⇒ (′*in* ⇒ ′*in′′* ⇒ *bool*) ⇒ (′*in′′* ⇒ ′*in′* ⇒ *bool*)
⇒ (′*a*, ′*b*, ′*out*, ′*in*) *converter* × (′*c*, ′*d*, ′*out′*, ′*in′*) *converter* ⇒ (′*e*, ′*b* × ′*d*, ′*out* × ′*out′*, ′*in′′*) *converter* **where**
rel-witness-converter A A′ B C R R′ = *corec-converter* (λ(*conv1*, *conv2*).
map-gpv (*map-prod id Inr* ∘ *rel-witness-prod*) *id* ∘
rel-witness-gpv (*rel-prod B* (*rel-converter* (*A OO A′*) *B C* (*R OO R′*))) *C R R′* ∘
rel-witness-fun A A′ (*run-converter conv1*, *run-converter conv2*))

lemma *rel-witness-converter-sel* [*simp*]:
run-converter (*rel-witness-converter A A′ B C R R′* (*conv1*, *conv2*)) =
map-gpv (*map-prod id* (*rel-witness-converter A A′ B C R R′*) ∘ *rel-witness-prod*) *id* ∘
rel-witness-gpv (*rel-prod B* (*rel-converter* (*A OO A′*) *B C* (*R OO R′*))) *C R R′* ∘
rel-witness-fun A A′ (*run-converter conv1*, *run-converter conv2*)
by(*auto simp add*: *rel-witness-converter-def o-def fun-eq-iff gpv.map-comp intro*!: *gpv.map-cong*)

lemma assumes *rel-converter* (*A OO A′*) *B C* (*R OO R′*) *conv conv′*
and *A*: *left-unique A right-total A*
and *A′*: *right-unique A′ left-total A′*
and *R*: *left-unique R right-total R*
and *R′*: *right-unique R′ left-total R′*
shows *rel-witness-converter1*: *rel-converter A* (λ*b* (*b′*, *c*). *b* = *b′* ∧ *B b′ c*) (λ*c* (*c′*, *d*). *c* = *c′* ∧ *C c′ d*) *R conv* (*rel-witness-converter A A′ B C R R′* (*conv*, *conv′*)) (**is** *?thesis1*)
and *rel-witness-converter2*: *rel-converter A′* (λ(*b*, *c′*) *c*. *c* = *c′* ∧ *B b c′*) (λ(*c*, *d′*) *d*. *d* = *d′* ∧ *C c d′*) *R′* (*rel-witness-converter A A′ B C R R′* (*conv*, *conv′*)) *conv′* (**is** *?thesis2*)
proof −
show *?thesis1* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *conv conv′*)
case *rel-converter*
from *this*[*THEN rel-converterD*] **show** *?case*
apply(*simp add*: *rel-fun-comp*)
apply(*erule rel-fun-mono*[*OF rel-witness-fun1*[*OF - A A′*]]; *clarsimp simp add*: *rel-gpv′′-map-gpv*)
apply(*erule rel-gpv′′-mono*[*THEN predicate2D*, *rotated* −*1*, *OF rel-witness-gpv1*[*OF - R R′*]]; *auto*)
done
qed
show *?thesis2* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *conv conv′*)
case *rel-converter*
from *this*[*THEN rel-converterD*] **show** *?case*
apply(*simp add*: *rel-fun-comp*)
apply(*erule rel-fun-mono*[*OF rel-witness-fun2*[*OF - A A′*]]; *clarsimp simp add*: *rel-gpv′′-map-gpv*)
apply(*erule rel-gpv′′-mono*[*THEN predicate2D*, *rotated* −*1*, *OF rel-witness-gpv2*[*OF - R R′*]]; *auto*)
done
qed
qed

lemma *rel-converter-neg-distr* [*relator-distr*]:
assumes *A*: *left-unique A right-total A*

and A': *right-unique* A' *left-total* A'
and R: *left-unique* R *right-total* R
and R': *right-unique* R' *left-total* R'
shows *rel-converter* $(A\ OO\ A')\ (B\ OO\ B')\ (C\ OO\ C')\ (R\ OO\ R') \leq$ *rel-converter* $A\ B\ C\ R\ OO$ *rel-converter* $A'\ B'\ C'\ R'$
proof(*rule predicate2I relcomppI*)+
fix *conv conv''*
assume $*$: *rel-converter* $(A\ OO\ A')\ (B\ OO\ B')\ (C\ OO\ C')\ (R\ OO\ R')$ *conv conv''*
let *?conv'* = *map-converter'* (*relcompp-witness* $B\ B'$) (*relcompp-witness* $C\ C'$) (*rel-witness-converter* $A\ A'\ (B\ OO\ B')\ (C\ OO\ C')\ R\ R'$ (*conv*, *conv''*))
show *rel-converter* $A\ B\ C\ R$ *conv ?conv'* **using** *rel-witness-converter1*[*OF* $*\ A\ A'\ R\ R'$] **unfolding** *converter-rel-map'*
by(*rule rel-converter-mono*[*THEN predicate2D*, *rotated* -1]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)
show *rel-converter* $A'\ B'\ C'\ R'$ *?conv' conv''* **using** *rel-witness-converter2*[*OF* $*\ A\ A'\ R\ R'$] **unfolding** *converter-rel-map'*
by(*rule rel-converter-mono*[*THEN predicate2D*, *rotated* -1]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)
qed

lemma *left-total-rel-converter*:
⟦ *left-unique* A; *right-total* A; *left-total* B; *left-total* C; *left-unique* R; *right-total* R ⟧
⟹ *left-total* (*rel-converter* $A\ B\ C\ R$)
unfolding *left-unique-alt-def left-total-alt-def rel-converter-conversep*[*symmetric*]
apply(*subst rel-converter-eq*[*symmetric*])
apply(*rule order-trans*[*rotated*])
apply(*rule rel-converter-neg-distr*; *simp add*: *left-unique-alt-def*)
apply(*rule rel-converter-mono*; *assumption*)
done

lemma *right-total-rel-converter*:
⟦ *right-unique* A; *left-total* A; *right-total* B; *right-total* C; *right-unique* R; *left-total* R ⟧
⟹ *right-total* (*rel-converter* $A\ B\ C\ R$)
unfolding *right-unique-alt-def right-total-alt-def rel-converter-conversep*[*symmetric*]
apply(*subst rel-converter-eq*[*symmetric*])
apply(*rule order-trans*[*rotated*])
apply(*rule rel-converter-neg-distr*; *simp add*: *right-unique-alt-def*)
apply(*rule rel-converter-mono*; *assumption*)
done

lemma *bi-total-rel-converter* [*transfer-rule*]:
⟦ *bi-total* A; *bi-unique* A; *bi-total* B; *bi-total* C; *bi-total* R; *bi-unique* R ⟧
⟹ *bi-total* (*rel-converter* $A\ B\ C\ R$)
unfolding *bi-total-alt-def bi-unique-alt-def*
by(*blast intro*: *left-total-rel-converter right-total-rel-converter*)

inductive *pred-converter* :: *'a set* ⇒ (*'b* ⇒ *bool*) ⇒ (*'out* ⇒ *bool*) ⇒ *'in set* ⇒ (*'a*, *'b*, *'out*, *'in*) *converter* ⇒ *bool*
for *A B C R conv* **where**
pred-converter A B C R conv **if**
∀ *x*∈*results-converter* (*I-uniform A UNIV*) (*I-uniform UNIV R*) *conv*. *B x*
∀ *out*∈*outs-converter* (*I-uniform A UNIV*) (*I-uniform UNIV R*) *conv*. *C out*

lemma *pred-gpv'-mono-weak*:
pred-gpv' A C R ≤ *pred-gpv' A' C' R* **if** *A* ≤ *A' C* ≤ *C'*
using *that* **by**(*auto 4 3 simp add*: *pred-gpv'.simps*)

lemma *Domainp-rel-converter-le*:
Domainp (*rel-converter A B C R*) ≤ *pred-converter* (*Collect* (*Domainp A*)) (*Domainp B*) (*Domainp C*) (*Collect* (*Domainp R*))
(**is** *?lhs* ≤ *?rhs*)
proof(*intro predicate1I pred-converter.intros strip*)
fix *conv*
assume ∗: *?lhs conv*
let *?I* = *I-uniform* (*Collect* (*Domainp A*)) *UNIV* **and** *?I'* = *I-uniform UNIV* (*Collect* (*Domainp R*))
show *Domainp B x* **if** *x* ∈ *results-converter ?I ?I' conv* **for** *x* **using** *that* ∗
apply(*induction*)
apply *clarsimp*
apply(*erule rel-converter.cases*; *clarsimp*)
apply(*drule* (*1*) *rel-funD*)
apply(*drule Domainp-rel-gpv''-le*[*THEN predicate1D*, *OF DomainPI*])
apply(*erule pred-gpv'.cases*)
apply *fastforce*
apply *clarsimp*
apply(*erule rel-converter.cases*; *clarsimp*)
apply(*drule* (*1*) *rel-funD*)
apply(*drule Domainp-rel-gpv''-le*[*THEN predicate1D*, *OF DomainPI*])
apply(*erule pred-gpv'.cases*)
apply *fastforce*
done
show *Domainp C x* **if** *x* ∈ *outs-converter ?I ?I' conv* **for** *x* **using** *that* ∗
apply *induction*
apply *clarsimp*
apply(*erule rel-converter.cases*; *clarsimp*)
apply(*drule* (*1*) *rel-funD*)
apply(*drule Domainp-rel-gpv''-le*[*THEN predicate1D*, *OF DomainPI*])
apply(*erule pred-gpv'.cases*)
apply *fastforce*
apply *clarsimp*
apply(*erule rel-converter.cases*; *clarsimp*)
apply(*drule* (*1*) *rel-funD*)
apply(*drule Domainp-rel-gpv''-le*[*THEN predicate1D*, *OF DomainPI*])
apply(*erule pred-gpv'.cases*)
apply *fastforce*

```
  done
qed

lemma rel-converter-Grp:
  rel-converter (BNF-Def.Grp UNIV f)^-1-1 (BNF-Def.Grp B g) (BNF-Def.Grp
C h) (BNF-Def.Grp UNIV k)^-1-1 =
   BNF-Def.Grp {conv. results-converter (ℐ-uniform (range f) UNIV) (ℐ-uniform
UNIV (range k)) conv ⊆ B ∧
    outs-converter (ℐ-uniform (range f) UNIV) (ℐ-uniform UNIV (range k)) conv
⊆ C}
    (map-converter f g h k)
  (is ?lhs = ?rhs)
  including lifting-syntax
proof(intro ext GrpI iffI CollectI conjI subsetI)
  let ?ℐ = ℐ-uniform (range f) UNIV and ?ℐ' = ℐ-uniform UNIV (range k)
  fix conv conv'
  assume ∗: ?lhs conv conv'
  then show map-converter f g h k conv = conv'
    apply(coinduction arbitrary: conv conv')
    apply(drule rel-converterD)
    apply(unfold map-converter.sel)
    apply(subst (2) map-fun-def[symmetric])
    apply(subst map-fun2-id)
    apply(subst rel-fun-comp)
    apply(rule rel-fun-map-fun1)
    apply(erule rel-fun-mono, simp)
    apply(simp add: gpv.rel-map)
   by (auto simp add: rel-gpv-conv-rel-gpv'' prod.rel-map intro!: predicate2I rel-gpv''-map-gpv'1
        elim!: rel-gpv''-mono[THEN predicate2D, rotated -1] prod.rel-mono-strong
GrpE)
  show b ∈ B if b ∈ results-converter ?ℐ ?ℐ' conv for b using ∗ that
    by - (drule Domainp-rel-converter-le[THEN predicate1D, OF DomainPI]
      , auto simp add: Domainp-conversep Rangep-Grp iff: Grp-iff elim: pred-converter.cases)
  show out ∈ C if out ∈ outs-converter ?ℐ ?ℐ' conv for out using ∗ that
    by - (drule Domainp-rel-converter-le[THEN predicate1D, OF DomainPI]
      , auto simp add: Domainp-conversep Rangep-Grp iff: Grp-iff elim: pred-converter.cases)
next
   let ?abr1=λconv. results-converter (ℐ-uniform (range f) UNIV) (ℐ-uniform
UNIV (range k)) conv ⊆ B
  let ?abr2=λconv. outs-converter (ℐ-uniform (range f) UNIV) (ℐ-uniform UNIV
(range k)) conv ⊆ C

  fix conv conv'
  assume ?rhs conv conv'
  hence ∗: conv' = map-converter f g h k conv and f1: ?abr1 conv and f2: ?abr2
conv by(auto simp add: Grp-iff)

  have[intro]: ?abr1 conv ⟹ ?abr2 conv ⟹ z ∈ run-converter conv ‘ range f
⟹
```

```
    out ∈ outs-gpv (ℐ-uniform UNIV (range k)) z ⟹ BNF-Def.Grp C h out
(h out) for conv z out
    by(auto simp add: Grp-iff elim:  outs-converter.Out elim!: subsetD)

  from f1 f2 show ?lhs conv conv' unfolding ∗
    apply(coinduction arbitrary: conv)
    apply(unfold map-converter.sel)
    apply(subst (2) map-fun-def[symmetric])
    apply(subst map-fun2-id)
    apply(subst rel-fun-comp)
    apply(rule rel-fun-map-fun2)
    apply(rule rel-fun-refl-eq-onp)
    apply(unfold map-gpv-conv-map-gpv' gpv.comp comp-id)
    apply(subst map-gpv'-id12)
    apply(rule rel-gpv''-map-gpv'2)
    apply(unfold rel-gpv''-map-gpv)
    apply(rule rel-gpv''-refl-eq-on)
     apply(simp add: prod.rel-map)
     apply(rule prod.rel-refl-strong)
      apply(clarsimp simp add: Grp-iff)
   by (auto intro: results-converter.Result results-converter.Cont outs-converter.Cont
elim!: subsetD)
qed

context
  includes lifting-syntax
  notes [transfer-rule] = map-gpv-parametric'
begin

lemma Converter-parametric [transfer-rule]:
  ((A ===> rel-gpv'' (rel-prod B (rel-converter A B C R)) C R) ===> rel-converter
A B C R) Converter Converter
  by(rule rel-funI)(simp)

lemma run-converter-parametric [transfer-rule]:
  (rel-converter A B C R ===> A ===> rel-gpv'' (rel-prod B (rel-converter A
B C R)) C R)
  run-converter run-converter
  by(rule rel-funI)(auto dest: rel-converterD)

lemma corec-converter-parametric [transfer-rule]:
  ((S ===> A ===> rel-gpv'' (rel-prod B (rel-sum (rel-converter A B C R) S))
C R) ===> S ===> rel-converter A B C R)
   corec-converter corec-converter
proof((rule rel-funI)+, goal-cases)
  case (1 f g s1 s2)
  then show ?case
    by(coinduction arbitrary: s1 s2)
      (drule 1(1)[THEN rel-funD]
```

, *auto 4 4 simp add*: *rel-fun-comp prod.rel-map*[*abs-def*] *rel-gpv''-map-gpv prod.rel-map split*: *sum.split*
intro: *prod.rel-mono elim*!: *rel-fun-mono rel-gpv''-mono*[*THEN predicate2D*, *rotated* −*1*])
qed

lemma *map-converter-parametric* [*transfer-rule*]:
((*A'* ===> *A*) ===> (*B* ===> *B'*) ===> (*C* ===> *C'*) ===> (*R'* ===> *R*) ===> *rel-converter A B C R* ===> *rel-converter A' B' C' R'*)
map-converter map-converter
unfolding *map-converter-def* **by**(*transfer-prover*)

lemma *map-converter'-parametric* [*transfer-rule*]:
((*B* ===> *B'*) ===> (*C* ===> *C'*) ===> *rel-converter* (=) *B C* (=) ===> *rel-converter* (=) *B' C'* (=))
map-converter' map-converter'
unfolding *map-converter-id14*[*symmetric*] **by** *transfer-prover*

lemma *case-converter-parametric* [*transfer-rule*]:
(((*A* ===> *rel-gpv''* (*rel-prod B* (*rel-converter A B C R*)) *C R*) ===> *X*) ===> *rel-converter A B C R* ===> *X*)
case-converter case-converter
unfolding *case-converter-conv-run-converter* **by** *transfer-prover*

end

2.5 Well-typing

coinductive *WT-converter* :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*out*, '*in*) $\mathcal{I}$ ⇒ ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ *bool*
(-,/ - ⊢$_C$/ - √ [*100*, *0*, *0*] *99*)
for $\mathcal{I}$ $\mathcal{I}'$ **where**
WT-converterI: $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ *conv* √ **if**
⋀*q*. *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}'$ ⊢*g run-converter conv q* √
⋀*q r conv'*. ⟦ *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; (*r*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*) ⟧ ⟹ *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* ∧ $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ *conv'* √

lemma *WT-converter-coinduct*[*consumes 1*, *case-names WT-converter*, *case-conclusion WT-converter WT-gpv results-gpv*, *coinduct pred*: *WT-converter*]:
assumes *X conv*
and ⋀*conv q r conv'*. ⟦ *X conv*; *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ $\mathcal{I}'$ ⊢*g run-converter conv q* √ ∧
((*r*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*) ⟶ *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* ∧ (*X conv'* ∨ $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ *conv'* √))
shows $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ *conv* √
using *assms*(*1*) **by**(*rule WT-converter.coinduct*)(*blast dest*: *assms*(*2*))

lemma *WT-converterD*:
assumes $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ *conv* √ *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$

shows *WT-converterD-WT*: $\mathcal{I}' \vdash g$ *run-converter conv q* $\surd$
and *WT-converterD-results*: $(r,$ *conv'*$) \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*)
$\Longrightarrow r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* $\wedge$ $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv'* $\surd$
using *assms* **by**(*auto elim*: *WT-converter.cases*)

lemma *WT-converterD'*:
assumes $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv* $\surd$ *q* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$
shows $\mathcal{I}' \vdash g$ *run-converter conv q* $\surd \wedge (\forall (r,$ *conv'*$) \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*). $r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* $\wedge$ $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv'* $\surd$)
using *assms* **by**(*auto elim*: *WT-converter.cases*)

lemma *WT-converter-bot1* [*simp*]: *bot*, $\mathcal{I} \vdash_C$ *conv* $\surd$
by(*rule WT-converter.intros*) *auto*

lemma *WT-converter-mono*:
$[\![\mathcal{I}1,\mathcal{I}2 \vdash_C$ *conv* $\surd; \mathcal{I}1' \leq \mathcal{I}1; \mathcal{I}2 \leq \mathcal{I}2']\!] \Longrightarrow \mathcal{I}1',\mathcal{I}2' \vdash_C$ *conv* $\surd$
apply(*coinduction arbitrary*: *conv*)
apply(*auto*)
apply(*drule WT-converterD-WT*)
apply(*erule* (*1*) *outs-*$\mathcal{I}$*-mono*[*THEN subsetD*])
apply(*erule WT-gpv-mono*)
apply(*erule outs-*$\mathcal{I}$*-mono*)
apply(*erule* (*1*) *responses-*$\mathcal{I}$*-mono*)
apply(*frule WT-converterD-results*)
apply(*erule* (*1*) *outs-*$\mathcal{I}$*-mono*[*THEN subsetD*])
apply(*erule results-gpv-mono*[*THEN subsetD*])
apply(*erule WT-converterD-WT*)
apply(*erule* (*1*) *outs-*$\mathcal{I}$*-mono*[*THEN subsetD*])
apply *simp*
apply *clarify*
apply(*erule* (*2*) *responses-*$\mathcal{I}$*-mono*[*THEN subsetD*])
apply(*frule WT-converterD-results*)
apply(*erule* (*1*) *outs-*$\mathcal{I}$*-mono*[*THEN subsetD*])
apply(*erule results-gpv-mono*[*THEN subsetD*])
apply(*erule WT-converterD-WT*)
apply(*erule* (*1*) *outs-*$\mathcal{I}$*-mono*[*THEN subsetD*])
apply *simp*
apply *simp*
done

lemma *callee-invariant-on-run-resource* [*simp*]: *callee-invariant-on run-resource* (*WT-resource* $\mathcal{I}$) $\mathcal{I}$
by(*unfold-locales*)(*auto dest*: *WT-resourceD intro*: *WT-calleeI*)

interpretation *run-resource*: *callee-invariant-on run-resource WT-resource* $\mathcal{I}$ $\mathcal{I}$
for $\mathcal{I}$
by *simp*

lemma *raw-converter-invariant-run-converter*: *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ *run-converter*

(*WT-converter* $\mathcal{I}$ $\mathcal{I}'$)
by(*unfold-locales*)(*auto dest*: *WT-converterD*)

interpretation *run-converter*: *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ *run-converter WT-converter* $\mathcal{I}$ $\mathcal{I}'$ **for** $\mathcal{I}$ $\mathcal{I}'$
by(*rule raw-converter-invariant-run-converter*)

lemma *WT-converter-*$\mathcal{I}$*-full*: $\mathcal{I}$*-full*, $\mathcal{I}$*-full* $\vdash_C$ *conv* $\surd$
by(*coinduction arbitrary*: *conv*)(*auto*)

lemma *WT-converter-map-converter* [*WT-intro*]:
$\mathcal{I}$, $\mathcal{I}' \vdash_C$ *map-converter f g f' g' conv* $\surd$ **if**
∗: *map-*$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$, *map-*$\mathcal{I}$ *f' g'* $\mathcal{I}' \vdash_C$ *conv* $\surd$
and *f*: *inj f* **and** *g*: *surj g*
using ∗
proof(*coinduction arbitrary*: *conv*)
case (*WT-converter q r conv' conv*)
have *?WT-gpv* **using** *WT-converter*
by(*auto intro*!: *WT-gpv-map-gpv' elim*: *WT-converterD-WT simp add*: *inv-into-f-f*[*OF f*])
moreover
have *?results-gpv*
proof(*intro strip conjI disjI1*)
assume (*r*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter* (*map-converter f g f' g' conv*) *q*)
then obtain *r' conv''*
where *results*: (*r'*, *conv''*) ∈ *results-gpv* (*map-*$\mathcal{I}$ *f' g'* $\mathcal{I}'$) (*run-converter conv* (*f q*))
and *r*: *r* = *g r'*
and *conv'*: *conv'* = *map-converter f g f' g' conv''*
by *auto*
from *WT-converterD-results*[*OF WT-converter*(*1*), *of f q*] *WT-converter*(*2*) *results*
have *r'*: *r'* ∈ *inv-into UNIV g* ‘ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q*
and *WT'*: *map-*$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$, *map-*$\mathcal{I}$ *f' g'* $\mathcal{I}' \vdash_C$ *conv''* $\surd$
by(*auto simp add*: *inv-into-f-f*[*OF f*])
from *r' r* **show** *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q* **by**(*auto simp add*: *surj-f-inv-f*[*OF g*])

show ∃ *conv*. *conv'* = *map-converter f g f' g' conv* ∧
map-$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$, *map-*$\mathcal{I}$ *f' g'* $\mathcal{I}' \vdash_C$ *conv* $\surd$
using *conv' WT'* **by**(*auto*)
qed
ultimately show *?case* **..**
qed

2.6 Losslessness

coinductive *plossless-converter* :: $('a, 'b)\ \mathcal{I} \Rightarrow ('out, 'in)\ \mathcal{I} \Rightarrow ('a, 'b, 'out, 'in)$ *converter* $\Rightarrow$ *bool*
for $\mathcal{I}\ \mathcal{I}'$ **where**
plossless-converterI: *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv* **if**
$\bigwedge a.\ a \in$ *outs-*$\mathcal{I}\ \mathcal{I} \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*)
$\bigwedge a\ b\ conv'.\ [\![\ a \in$ *outs-*$\mathcal{I}\ \mathcal{I};\ (b,\ conv') \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) $]\!] \Longrightarrow$ *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv'*

lemma *plossless-converter-coinduct*[*consumes 1*, *case-names plossless-converter*, *case-conclusion plossless-converter plossless step*, *coinduct pred*: *plossless-converter*]:
assumes *X conv*
and *step*: $\bigwedge$*conv a*. $[\![\ X$ *conv*; $a \in$ *outs-*$\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*) $\wedge$
$(\forall (b,$ *conv'*$) \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*). *X conv'* $\vee$ *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv'*)
shows *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv*
using *assms*(*1*) **by**(*rule plossless-converter.coinduct*)(*auto dest*: *step*)

lemma *plossless-converterD*:
$[\![$ *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv*; $a \in$ *outs-*$\mathcal{I}\ \mathcal{I}\ ]\!]$
$\Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*) $\wedge$
$(\forall (b,$ *conv'*$) \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*). *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv'*)
by(*auto elim*: *plossless-converter.cases*)

lemma *plossless-converter-bot1* [*simp*]: *plossless-converter bot* $\mathcal{I}$ *conv*
by(*rule plossless-converterI*) *auto*

lemma *plossless-converter-mono*:
assumes $*$: *plossless-converter* $\mathcal{I}1\ \mathcal{I}2$ *conv*
and *le*: *outs-*$\mathcal{I}\ \mathcal{I}1' \subseteq$ *outs-*$\mathcal{I}\ \mathcal{I}1\ \mathcal{I}2 \leq \mathcal{I}2'$
and *WT*: $\mathcal{I}1, \mathcal{I}2 \vdash_C$ *conv* $\surd$
shows *plossless-converter* $\mathcal{I}1'\ \mathcal{I}2'$ *conv*
using $*$ *WT*
apply(*coinduction arbitrary*: *conv*)
apply(*drule plossless-converterD*)
apply(*erule le*(*1*)[*THEN subsetD*])
apply(*drule WT-converterD'*)
apply(*erule le*(*1*)[*THEN subsetD*])
using *le*(*2*)[*THEN responses-*$\mathcal{I}$*-mono*]
by(*auto intro*: *plossless-gpv-mono*[*OF - le*(*2*)] *results-gpv-mono*[*OF le*(*2*), *THEN subsetD*] *dest*: *bspec*)

lemma *raw-converter-invariant-run-plossless-converter*: *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ *run-converter* (λ*conv*. *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv* $\wedge\ \mathcal{I}$,$\mathcal{I}' \vdash_C$ *conv* $\surd$)
by(*unfold-locales*)(*auto dest*: *WT-converterD plossless-converterD*)

interpretation *run-plossless-converter*: *raw-converter-invariant*

$\mathcal{I}$ $\mathcal{I}'$ *run-converter* λ*conv*. *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* $\wedge$ $\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *conv* $\surd$ **for** $\mathcal{I}$ $\mathcal{I}'$
by(*rule raw-converter-invariant-run-plossless-converter*)

named-theorems *plossless-intro Introduction rules for probabilistic losslessness*

2.7 Operations

context
fixes *callee* :: *'s* $\Rightarrow$ *'a* $\Rightarrow$ (*'b* $\times$ *'s*, *'out*, *'in*) *gpv*
begin

primcorec *converter-of-callee* :: *'s* $\Rightarrow$ (*'a*, *'b*, *'out*, *'in*) *converter* **where**
run-converter (*converter-of-callee s*) = (λ*a*. *map-gpv* (*map-prod id converter-of-callee*) *id* (*callee s a*))

end

lemma *converter-of-callee-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *A* ===> *rel-gpv''* (*rel-prod B S*) *C R*) ===> *S* ===> *rel-converter A B C R*)
converter-of-callee converter-of-callee
unfolding *converter-of-callee-def* **supply** *map-gpv-parametric'*[*transfer-rule*] **by** *transfer-prover*

lemma *map-converter-of-callee*:
map-converter f g h k (*converter-of-callee callee s*) =
converter-of-callee (*map-fun id* (*map-fun f* (*map-gpv'* (*map-prod g id*) *h k*)) *callee*) *s*
proof(*coinduction arbitrary*: *s*)
case *Eq-converter*
have $*$: *map-gpv'* (*map-prod g id*) *h k gpv* = *map-gpv* (*map-prod g id*) *id* (*map-gpv' id h k gpv*) **for** *gpv*
by(*simp add*: *map-gpv-conv-map-gpv' gpv.compositionality*)
show *?case*
by(*auto simp add*: *rel-fun-def map-gpv'-map-gpv-swap gpv.rel-map* $*$ *intro*!: *gpv.rel-refl-strong*)
qed

lemma *WT-converter-of-callee*:
assumes *WT*: $\bigwedge$*s q*. *q* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\Longrightarrow$ $\mathcal{I}'$ $\vdash$*g callee s q* $\surd$
and *res*: $\bigwedge$*s q r s'*. $[\![$ *q* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; (*r*, *s'*) $\in$ *results-gpv* $\mathcal{I}'$ (*callee s q*) $]\!]$ $\Longrightarrow$ *r* $\in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q*
shows $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *converter-of-callee callee s* $\surd$
by(*coinduction arbitrary*: *s*)(*auto simp add*: *WT res*)

We can define two versions of parallel composition. One that attaches to the same interface and one that attach to different interfaces. We choose the one variant where both attach to the same interface because (1) this is

more general and (2) we do not have to assume that the resource respects the parallel composition.

primcorec *parallel-converter*
:: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ ('*c*, '*d*, '*out*, '*in*) *converter* ⇒ ('*a* + '*c*, '*b* + '*d*, '*out*, '*in*) *converter*
where
run-converter (*parallel-converter conv1 conv2*) = (λ*ac*. *case ac of*
Inl a ⇒ *map-gpv* (*map-prod Inl* (λ*conv1'*. *parallel-converter conv1' conv2*)) *id* (*run-converter conv1 a*)
| *Inr b* ⇒ *map-gpv* (*map-prod Inr* (λ*conv2'*. *parallel-converter conv1 conv2'*)) *id* (*run-converter conv2 b*))

lemma *parallel-callee-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-converter A' B' C R* ===> *rel-converter* (*rel-sum A A'*) (*rel-sum B B'*) *C R*)
parallel-converter parallel-converter
unfolding *parallel-converter-def* **supply** *map-gpv-parametric'*[*transfer-rule*] **by** *transfer-prover*

lemma *parallel-converter-assoc*:
parallel-converter (*parallel-converter conv1 conv2*) *conv3* =
map-converter rsuml lsumr id id (*parallel-converter conv1* (*parallel-converter conv2 conv3*))
by(*coinduction arbitrary*: *conv1 conv2 conv3*)
(*auto 4 5 intro*!: *rel-funI gpv.rel-refl-strong split*: *sum.split simp add*: *gpv.rel-map map-gpv'-id map-gpv-conv-map-gpv'*[*symmetric*])

lemma *plossless-parallel-converter* [*plossless-intro*]:
⟦ *plossless-converter* $\mathcal{I}1$ $\mathcal{I}$ *conv1*; *plossless-converter* $\mathcal{I}2$ $\mathcal{I}$ *conv2*; $\mathcal{I}1, \mathcal{I} \vdash_C$ *conv1* √; $\mathcal{I}2, \mathcal{I} \vdash_C$ *conv2* √ ⟧
⟹ *plossless-converter* ($\mathcal{I}1$ $\oplus_{\mathcal{I}}$ $\mathcal{I}2$) $\mathcal{I}$ (*parallel-converter conv1 conv2*)
by(*coinduction arbitrary*: *conv1 conv2*)
(*clarsimp*; *erule PlusE*; *drule* (*1*) *plossless-converterD*; *drule* (*1*) *WT-converterD'*; *fastforce*)

primcorec *id-converter* :: ('*a*, '*b*, '*a*, '*b*) *converter* **where**
run-converter id-converter = (λ*a*.
map-gpv (*map-prod id* (λ-. *id-converter*)) *id* (*Pause a* (λ*b*. *Done* (*b*, ()))))

lemma *id-converter-parametric* [*transfer-rule*]: *rel-converter A B A B id-converter id-converter*
unfolding *id-converter-def*
supply *map-gpv-parametric'*[*transfer-rule*] *Done-parametric'*[*transfer-rule*] *Pause-parametric'*[*transfer-rule*]
by *transfer-prover*

lemma *converter-of-callee-id-oracle* [*simp*]:
converter-of-callee id-oracle s = *id-converter*
by(*coinduction*) (*auto simp add*: *id-oracle-def*)

lemma *conv-callee-plus-id-left*: *converter-of-callee* (*plus-intercept id-oracle callee*) *s* =
parallel-converter id-converter (*converter-of-callee callee s*)
by (*coinduction arbitrary*: *callee s*)
(*clarsimp split*!: *sum.split intro*!: *rel-funI*
, *force simp add*: *gpv.rel-map id-oracle-def*, *force simp add*: *gpv.rel-map intro*!: *gpv.rel-refl*)

lemma *conv-callee-plus-id-right*: *converter-of-callee* (*plus-intercept callee id-oracle*) *s* =
parallel-converter (*converter-of-callee callee s*) *id-converter*
by (*coinduction arbitrary*: *callee s*)
(*clarsimp split*!: *sum.split intro*!: *rel-funI*
, (*force intro*: *gpv.rel-refl* | *simp add*: *gpv.rel-map id-oracle-def*)+)

lemma *plossless-id-converter* [*simp*, *plossless-intro*]: *plossless-converter* $\mathcal{I}$ $\mathcal{I}$ *id-converter*
by(*coinduction*) *auto*

lemma *WT-converter-id* [*simp*, *intro*, *WT-intro*]: $\mathcal{I}$, $\mathcal{I}$ $\vdash_C$ *id-converter* $\surd$
by(*coinduction*) *auto*

lemma *WT-map-converter-idD*:
$\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *map-converter id id f g id-converter* $\surd$ $\Longrightarrow$ $\mathcal{I} \leq$ *map-*$\mathcal{I}$ *f g* $\mathcal{I}'$
unfolding *le-*$\mathcal{I}$*-def* **by**(*auto 4 3 dest*: *WT-converterD*)

definition *fail-converter* :: ('*a*, '*b*, '*out*, '*in*) *converter* **where**
fail-converter = *Converter* (λ-. *Fail*)

lemma *fail-converter-sel* [*simp*]: *run-converter fail-converter a* = *Fail*
by(*simp add*: *fail-converter-def*)

lemma *fail-converter-parametric* [*transfer-rule*]: *rel-converter A B C R fail-converter fail-converter*
unfolding *fail-converter-def* **supply** *Fail-parametric'*[*transfer-rule*] **by** *transfer-prover*

lemma *plossless-fail-converter* [*simp*]: *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *fail-converter* $\longleftrightarrow$ $\mathcal{I}$ = *bot* (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?rhs* **if** *?lhs* **using** *that* **by**(*cases*)(*auto intro*!: $\mathcal{I}$*-eqI*)
qed *simp*

lemma *plossless-fail-converterI* [*plossless-intro*]: *plossless-converter bot* $\mathcal{I}'$ *fail-converter*
by *simp*

lemma *WT-fail-converter* [*simp*, *WT-intro*]: $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *fail-converter* $\surd$
by(*rule WT-converter.intros*) *simp-all*

lemma *map-converter-id-move-left*:

map-converter f g f′ g′ id-converter = map-converter (f′ ∘ f) (g ∘ g′) id id id-converter
by *coinduction*(*simp add*: *rel-funI*)

lemma *map-converter-id-move-right*:
map-converter f g f′ g′ id-converter = map-converter id id (f′ ∘ f) (g ∘ g′) id-converter
by *coinduction*(*simp add*: *rel-funI*)

And here is the version for parallel composition that assumes disjoint interfaces.

primcorec *parallel-converter2*
:: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ ('*c*, '*d*, '*out*′, '*in*′) *converter* ⇒ ('*a* + '*c*, '*b* + '*d*, '*out* + '*out*′, '*in* + '*in*′) *converter*
where
run-converter (*parallel-converter2 conv1 conv2*) = (λ*ac*. *case ac of*
Inl a ⇒ *map-gpv* (*map-prod Inl* (λ*conv1*′. *parallel-converter2 conv1*′ *conv2*)) *id* (*left-gpv* (*run-converter conv1 a*))
| *Inr b* ⇒ *map-gpv* (*map-prod Inr* (λ*conv2*′. *parallel-converter2 conv1 conv2*′)) *id* (*right-gpv* (*run-converter conv2 b*)))

lemma *parallel-converter2-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-converter A*′ *B*′ *C*′ *R*′
===> *rel-converter* (*rel-sum A A*′) (*rel-sum B B*′) (*rel-sum C C*′) (*rel-sum R R*′))
parallel-converter2 parallel-converter2
unfolding *parallel-converter2-def*
supply *left-gpv-parametric*′[*transfer-rule*] *right-gpv-parametric*′[*transfer-rule*] *map-gpv-parametric*′[*transfer-r*
by *transfer-prover*

lemma *map-converter-parallel-converter2*:
map-converter (*map-sum f f*′) (*map-sum g g*′) (*map-sum h h*′) (*map-sum k k*′) (*parallel-converter2 conv1 conv2*) =
parallel-converter2 (*map-converter f g h k conv1*) (*map-converter f*′ *g*′ *h*′ *k*′ *conv2*)
using *parallel-converter2-parametric*[*of*
conversep (*BNF-Def.Grp UNIV f*) *BNF-Def.Grp UNIV g* *BNF-Def.Grp UNIV h* *conversep* (*BNF-Def.Grp UNIV k*)
conversep (*BNF-Def.Grp UNIV f*′) *BNF-Def.Grp UNIV g*′ *BNF-Def.Grp UNIV h*′ *conversep* (*BNF-Def.Grp UNIV k*′)]
unfolding *sum.rel-conversep sum.rel-Grp*
by(*simp add*: *rel-converter-Grp rel-fun-def Grp-iff*)

lemma *WT-converter-parallel-converter2* [*WT-intro*]:
assumes $\mathcal{I}1$,$\mathcal{I}2$ ⊢$_C$ *conv1* √
and $\mathcal{I}1'$,$\mathcal{I}2'$ ⊢$_C$ *conv2* √
shows $\mathcal{I}1$ ⊕$_\mathcal{I}$ $\mathcal{I}1'$,$\mathcal{I}2$ ⊕$_\mathcal{I}$ $\mathcal{I}2'$ ⊢$_C$ *parallel-converter2 conv1 conv2* √
using *assms*
apply(*coinduction arbitrary*: *conv1 conv2*)

apply(*clarsimp split*!: *sum.split*)
subgoal by(*auto intro*: *WT-gpv-left-gpv dest*: *WT-converterD-WT*)
subgoal by(*auto dest*: *WT-converterD-results*)
subgoal by(*auto dest*: *WT-converterD-results*)
subgoal by(*auto intro*: *WT-gpv-right-gpv dest*: *WT-converterD-WT*)
subgoal by(*auto dest*: *WT-converterD-results*)
subgoal by(*auto 4 3 dest*: *WT-converterD-results*)
done

lemma *plossless-parallel-converter2* [*plossless-intro*]:
assumes *plossless-converter* $\mathcal{I}1$ $\mathcal{I}1'$ *conv1*
and *plossless-converter* $\mathcal{I}2$ $\mathcal{I}2'$ *conv2*
shows *plossless-converter* ($\mathcal{I}1$ $\oplus_{\mathcal{I}}$ $\mathcal{I}2$) ($\mathcal{I}1'$ $\oplus_{\mathcal{I}}$ $\mathcal{I}2'$) (*parallel-converter2 conv1 conv2*)
using *assms*
by(*coinduction arbitrary*: *conv1 conv2*)
((*rule exI conjI refl*)+ | *auto dest*: *plossless-converterD*)+

lemma *parallel-converter2-map1-out*:
parallel-converter2 (*map-converter f g h k conv1*) *conv2* =
map-converter (*map-sum f id*) (*map-sum g id*) (*map-sum h id*) (*map-sum k id*) (*parallel-converter2 conv1 conv2*)
by(*simp add*: *map-converter-parallel-converter2*)

lemma *parallel-converter2-map2-out*:
parallel-converter2 conv1 (*map-converter f g h k conv2*) =
map-converter (*map-sum id f*) (*map-sum id g*) (*map-sum id h*) (*map-sum id k*) (*parallel-converter2 conv1 conv2*)
by(*simp add*: *map-converter-parallel-converter2*)

primcorec *left-interface* :: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ ('*a*, '*b*, '*out* + '*out'*, '*in* + '*in'*) *converter* **where**
run-converter (*left-interface conv*) = (λ*a*. *map-gpv* (*map-prod id left-interface*) *id* (*left-gpv* (*run-converter conv a*)))

lemma *left-interface-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-converter A B* (*rel-sum C C'*) (*rel-sum R R'*)) *left-interface left-interface*
unfolding *left-interface-def*
supply *left-gpv-parametric'*[*transfer-rule*] *map-gpv-parametric'*[*transfer-rule*] **by** *transfer-prover*

primcorec *right-interface* :: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ ('*a*, '*b*, '*out'* + '*out*, '*in'* + '*in*) *converter* **where**
run-converter (*right-interface conv*) = (λ*a*. *map-gpv* (*map-prod id right-interface*) *id* (*right-gpv* (*run-converter conv a*)))

lemma *right-interface-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**

(*rel-converter A B C′ R′ ===> rel-converter A B* (*rel-sum C C′*) (*rel-sum R R′*)) *right-interface right-interface*
unfolding *right-interface-def*
supply *right-gpv-parametric′*[*transfer-rule*] *map-gpv-parametric′*[*transfer-rule*] **by** *transfer-prover*

lemma *parallel-converter2-alt-def*:
parallel-converter2 conv1 conv2 = *parallel-converter* (*left-interface conv1*) (*right-interface conv2*)
by(*coinduction arbitrary*: *conv1 conv2 rule*: *converter.coinduct-strong*)
(*auto 4 5 intro*!: *rel-funI gpv.rel-refl-strong split*: *sum.split simp add*: *gpv.rel-map*)

lemma *conv-callee-parallel-id-left*: *converter-of-callee* (*parallel-intercept id-oracle callee*) (*s*, *s′*) =
parallel-converter2 (*id-converter*) (*converter-of-callee callee s′*)
apply (*coinduction arbitrary*: *callee s′*)
apply (*rule rel-funI*)
apply (*clarsimp simp add*: *gpv.rel-map left-gpv-map*[*of - - - id*]
right-gpv-map[*of - - - id*] *split*!: *sum.split*)
apply (*force simp add*: *id-oracle-def split*!: *sum.split*)
apply (*rule gpv.rel-refl*)
by *force*+

lemma *conv-callee-parallel-id-right*: *converter-of-callee* (*parallel-intercept callee id-oracle*) (*s*, *s′*) =
parallel-converter2 (*converter-of-callee callee s*) (*id-converter*)
apply (*coinduction arbitrary*: *callee s*)
apply (*rule rel-funI*)
apply (*clarsimp simp add*: *gpv.rel-map left-gpv-map*[*of - - - id*]
right-gpv-map[*of - - - id*] *split*!: *sum.split*)
apply (*rule gpv.rel-refl*)
by (*force simp add*: *id-oracle-def split*!: *sum.split*)+

lemma *conv-callee-parallel*: *converter-of-callee* (*parallel-intercept callee1 callee2*) (*s*,*s′*)
= *parallel-converter2* (*converter-of-callee callee1 s*) (*converter-of-callee callee2 s′*)
apply (*coinduction arbitrary*: *callee1 callee2 s s′*)
apply (*clarsimp simp add*: *gpv.rel-map left-gpv-map*[*of - - - id*] *right-gpv-map*[*of - - - id*] *intro*!: *rel-funI split*!: *sum.split*)
apply (*rule gpv.rel-refl*)
apply *force*+
apply (*rule gpv.rel-refl*)
by *force*+

lemma *WT-converter-parallel-converter* [*WT-intro*]:
assumes $\mathcal{I}1, \mathcal{I} \vdash_C$ *conv1* $\surd$
and $\mathcal{I}2, \mathcal{I} \vdash_C$ *conv2* $\surd$
shows $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I} \vdash_C$ *parallel-converter conv1 conv2* $\surd$
using *assms* **by**(*coinduction arbitrary*: *conv1 conv2*)(*auto 4 4 dest*: *WT-converterD*

intro!: *imageI*)

primcorec *converter-of-resource* :: ('*a*, '*b*) *resource* ⇒ ('*a*, '*b*, '*c*, '*d*) *converter*
where
run-converter (*converter-of-resource res*) = (λ*x*. *map-gpv* (*map-prod id converter-of-resource*) *id* (*lift-spmf* (*run-resource res x*)))

lemma *WT-converter-of-resource* [*WT-intro*]:
assumes $\mathcal{I} \vdash_{res}$ *res* √
shows $\mathcal{I}, \mathcal{I}' \vdash_C$ *converter-of-resource res* √
using *assms* **by**(*coinduction arbitrary*: *res*)(*auto dest*: *WT-resourceD*)

lemma *plossless-converter-of-resource* [*plossless-intro*]:
assumes *lossless-resource* $\mathcal{I}$ *res*
shows *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ (*converter-of-resource res*)
using *assms* **by**(*coinduction arbitrary*: *res*)(*auto 4 3 dest*: *lossless-resourceD*)

lemma *plossless-converter-of-callee*:
assumes ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}1$ ⟹ *plossless-gpv* $\mathcal{I}2$ (*callee s x*) ∧ (∀(*y*, *s'*)∈*results-gpv* $\mathcal{I}2$ (*callee s x*). *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}1$ *x*)
shows *plossless-converter* $\mathcal{I}1$ $\mathcal{I}2$ (*converter-of-callee callee s*)
apply(*coinduction arbitrary*: *s*)
subgoal for *x s* **by**(*drule assms*[**where** *s*=*s*]) *auto*
done

context
fixes *A* :: '*a set*
and $\mathcal{I}$ :: ('*c*, '*d*) $\mathcal{I}$
begin

primcorec *restrict-converter* :: ('*a*, '*b*, '*c*, '*d*) *converter* ⇒ ('*a*, '*b*, '*c*, '*d*) *converter*
where
run-converter (*restrict-converter cnv*) = (λ*a*. *if a* ∈ *A then*
map-gpv (*map-prod id* (λ*cnv'*. *restrict-converter cnv'*)) *id* (*restrict-gpv* $\mathcal{I}$ (*run-converter cnv a*))
else Fail)

end

lemma *WT-restrict-converter* [*WT-intro*]:
assumes $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *cnv* √
shows $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *restrict-converter A* $\mathcal{I}'$ *cnv* √
using *assms* **by**(*coinduction arbitrary*: *cnv*)(*auto dest*: *WT-converterD dest*!: *in-results-gpv-restrict-gpvD*)

lemma *pgen-lossless-restrict-gpv* [*simp*]:
$\mathcal{I} \vdash_g$ *gpv* √ ⟹ *pgen-lossless-gpv b* $\mathcal{I}$ (*restrict-gpv* $\mathcal{I}$ *gpv*) = *pgen-lossless-gpv b* $\mathcal{I}$ *gpv*
unfolding *pgen-lossless-gpv-def* **by**(*simp add*: *expectation-gpv-restrict-gpv*)

lemma *plossless-restrict-converter* [*simp*]:
assumes *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*
and $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *conv* $\surd$
and *outs-*$\mathcal{I}$ $\mathcal{I} \subseteq A$
shows *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ (*restrict-converter A* $\mathcal{I}'$ *conv*)
using *assms*
by(*coinduction arbitrary*: *conv*)
(*auto dest*!: *in-results-gpv-restrict-gpvD WT-converterD' plossless-converterD*)

lemma *plossless-map-converter*:
plossless-converter $\mathcal{I}$ $\mathcal{I}'$ (*map-converter f g h k conv*)
if *plossless-converter* (*map-*$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$) (*map-*$\mathcal{I}$ *h k* $\mathcal{I}'$) *conv inj f*
using *that*
by(*coinduction arbitrary*: *conv*)(*auto dest*!: *plossless-converterD*[**where** *a*=*f* -])

2.8 Attaching converters to resources

primcorec *attach* :: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ ('*out*, '*in*) *resource* ⇒ ('*a*, '*b*) *resource* **where**
run-resource (*attach conv res*) = (λ*a*.
map-spmf (λ((*b*, *conv'*), *res'*). (*b*, *attach conv' res'*)) (*exec-gpv run-resource* (*run-converter conv a*) *res*))

lemma *attach-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-resource C R* ===> *rel-resource A B*) *attach attach*
unfolding *attach-def*
supply *exec-gpv-parametric'*[*transfer-rule*] **by** *transfer-prover*

lemma *attach-map-converter*:
attach (*map-converter f g h k conv*) *res* = *map-resource f g* (*attach conv* (*map-resource h k res*))
using *attach-parametric*[*of conversep* (*BNF-Def.Grp UNIV f*) *BNF-Def.Grp UNIV g BNF-Def.Grp UNIV h conversep* (*BNF-Def.Grp UNIV k*)]
unfolding *rel-converter-Grp rel-resource-Grp*
by (*simp*, *rewrite at rel-fun* - (*rel-fun* ⌑ -) **in** *asm conversep-iff*[*symmetric*, *abs-def*])
(*simp add*: *rel-resource-conversep*[*symmetric*] *rel-fun-def Grp-iff conversep-conversep rel-resource-Grp*)

lemma *WT-resource-attach* [*WT-intro*]: ⟦ $\mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv* $\surd$; $\mathcal{I}' \vdash$*res res* $\surd$ ⟧ ⟹ $\mathcal{I} \vdash$*res attach conv res* $\surd$
by(*coinduction arbitrary*: *conv res*)
(*auto 4 3 intro*!: *exI dest*: *run-resource.in-set-spmf-exec-gpv-into-results-gpv WT-converterD intro*: *run-resource.exec-gpv-invariant*)

lemma *lossless-attach* [*plossless-intro*]:

assumes *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*
and *lossless-resource* $\mathcal{I}'$ *res*
and $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv* $\surd$ $\mathcal{I}' \vdash$*res res* $\surd$
shows *lossless-resource* $\mathcal{I}$ (*attach conv res*)
using *assms*
proof(*coinduction arbitrary*: *res conv*)
case (*lossless-resource a res conv*)
from *plossless-converterD*[*OF lossless-resource*(*1*,*5*)] **have** *lossless*: *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*)
$\bigwedge$*b conv'*. (*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) ⟹ *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv'* **by** *auto*
from *WT-converterD'*[*OF lossless-resource*(*3*,*5*)] **have** *WT*: $\mathcal{I}' \vdash g$ *run-converter conv a* $\surd$
$\bigwedge$*b conv'*. (*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) ⟹ *b* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *a* ∧ $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv'* $\surd$ **by** *auto*
have *?lossless* **using** *lossless*(*1*) *WT*(*1*) *lossless-resource*(*2*,*4*)
by(*auto intro*: *run-lossless-resource.plossless-exec-gpv dest*: *lossless-resourceD*)
moreover have *?step* (**is** ∀(*b*, *res'*)∈*?set*. *?P b res'* ∨ -)
proof(*safe*)
fix *b res''*
assume (*b*, *res''*) ∈ *?set*
then obtain *conv' res'* **where** ∗: ((*b*, *conv'*), *res'*) ∈ *set-spmf* (*exec-gpv run-resource* (*run-converter conv a*) *res*)
and [*simp*]: *res''* = *attach conv' res'* **by** *auto*
from *run-lossless-resource.in-set-spmf-exec-gpv-into-results-gpv*[*OF* ∗, *of* $\mathcal{I}'$] *lossless-resource*(*2*,*4*) *WT*
have *conv'*: (*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) **by** *auto*
from *run-lossless-resource.exec-gpv-invariant*[*OF* ∗, *of* $\mathcal{I}'$] *WT*(*2*)[*OF this*] *WT*(*1*) *lossless*(*2*)[*OF this*] *lossless-resource*
show *?P b res''* **by** *auto*
qed
ultimately show *?case* **..**
qed

definition *attach-callee*
:: ('*s* ⇒ '*a* ⇒ ('*b* × '*s*, '*out*, '*in*) *gpv*)
⇒ ('*s'* ⇒ '*out* ⇒ ('*in* × '*s'*) *spmf*)
⇒ ('*s* × '*s'* ⇒ '*a* ⇒ ('*b* × '*s* × '*s'*) *spmf*) **where**
attach-callee callee oracle = (λ(*s*, *s'*) *q*. *map-spmf rprodl* (*exec-gpv oracle* (*callee s q*) *s'*))

lemma *attach-callee-simps* [*simp*]:
attach-callee callee oracle (*s*, *s'*) *q* = *map-spmf rprodl* (*exec-gpv oracle* (*callee s q*) *s'*)
by(*simp add*: *attach-callee-def*)

lemma *attach-CNV-RES*:
attach (*converter-of-callee callee s*) (*resource-of-oracle res s'*) =

resource-of-oracle (*attach-callee callee res*) (*s*, *s'*)
by(*coinduction arbitrary*: *s s'*)
(*clarsimp simp add*: *spmf-rel-map rel-fun-def exec-gpv-map-gpv-id*
, *rule exec-gpv-parametric*[**where** *S*=λ*l r*. *l* = *resource-of-oracle res r* **and** *A*=(=) **and** *CALL*=(=), *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-spmf-mono*]
, *auto 4 3 simp add*: *rel-fun-def spmf-rel-map gpv.rel-eq intro*!: *rel-spmf-reflI*)

lemma *attach-stateless-callee*:
attach-callee (*stateless-callee callee*) *oracle* = *extend-state-oracle* (λ*s q*. *exec-gpv oracle* (*callee q*) *s*)
by(*simp add*: *attach-callee-def stateless-callee-def fun-eq-iff exec-gpv-map-gpv-id spmf.map-comp o-def split-def apfst-def map-prod-def*)

lemma *attach-id-converter* [*simp*]: *attach id-converter res* = *res*
by(*coinduction arbitrary*: *res*)(*auto simp add*: *rel-fun-def spmf-rel-map split-def map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *rel-spmf-reflI*)

lemma *attach-callee-parallel-intercept*: **includes** *lifting-syntax* **shows**
attach-callee (*parallel-intercept callee1 callee2*) (*plus-oracle oracle1 oracle2*) =
(*rprodl* −−−> *id* −−−> *map-spmf* (*map-prod id lprodr*)) (*plus-oracle* (*lift-state-oracle extend-state-oracle* (*attach-callee callee1 oracle1*)) (*extend-state-oracle* (*attach-callee callee2 oracle2*)))
proof ((*rule ext*)+, *clarify*, *goal-cases*)
case (*1 s1 s2 s q*)
then show *?case* **by**(*cases q*) (*auto simp add*: *exec-gpv-plus-oracle-left exec-gpv-plus-oracle-right spmf.map-comp apfst-def o-def prod.map-comp split-def exec-gpv-map-gpv-id intro*!: *map-spmf-cong*)
qed

lemma *attach-callee-id-oracle* [*simp*]:
attach-callee id-oracle oracle = *extend-state-oracle oracle*
by(*clarsimp simp add*: *fun-eq-iff id-oracle-def map-spmf-conv-bind-spmf split-def*)

lemma *attach-parallel2*: *attach* (*parallel-converter2 conv1 conv2*) (*parallel-resource res1 res2*)
= *parallel-resource* (*attach conv1 res1*) (*attach conv2 res2*)
apply(*coinduction arbitrary*: *conv1 conv2 res1 res2*)
apply *simp*
apply(*rule rel-funI*)
apply *clarsimp*
apply(*simp split*!: *sum.split*)
subgoal for *conv1 conv2 res1 res2 a*
apply(*simp add*: *exec-gpv-map-gpv-id spmf-rel-map*)
apply(*rule rel-spmf-mono*)
apply(*rule*
exec-gpv-parametric'[**where** *?S* = λ*res1res2 res1*. *res1res2* = *parallel-resource res1 res2* **and**
A=(=) **and** *CALL*=λ*l r*. *l* = *Inl r* **and** *R*=λ*l r*. *l* = *Inl r*,

THEN rel-funD, *THEN rel-funD*, *THEN rel-funD*
])
subgoal by(*auto simp add*: *rel-fun-def spmf-rel-map intro*!: *rel-spmf-reflI*)
subgoal by (*simp add*: *left-gpv-Inl-transfer*)
subgoal by *blast*
apply *clarsimp*
apply(*rule exI conjI refl*)+
done
subgoal for *conv1 conv2 res1 res2 a*
apply(*simp add*: *exec-gpv-map-gpv-id spmf-rel-map*)
apply(*rule rel-spmf-mono*)
apply(*rule*
exec-gpv-parametric'[**where** *?S* = *λres1res2 res2*. *res1res2* = *parallel-resource res1 res2* **and**
A=(=) **and** *CALL*=*λl r*. *l* = *Inr r* **and** *R*=*λl r*. *l* = *Inr r*,
THEN rel-funD, *THEN rel-funD*, *THEN rel-funD*
])
subgoal by(*auto simp add*: *rel-fun-def spmf-rel-map intro*: *rel-spmf-reflI*)
subgoal by (*simp add*: *right-gpv-Inr-transfer*)
subgoal by *blast*
apply *clarsimp*
apply(*rule exI conjI refl*)+
done
done

2.9 Composing converters

primcorec *comp-converter* :: ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ ('*out*, '*in*, '*out'*, '*in'*) *converter* ⇒ ('*a*, '*b*, '*out'*, '*in'*) *converter* **where**
run-converter (*comp-converter conv1 conv2*) = (*λa*.
map-gpv (*λ*((*b*, *conv1'*), *conv2'*). (*b*, *comp-converter conv1' conv2'*)) *id* (*inline run-converter* (*run-converter conv1 a*) *conv2*))

lemma *comp-converter-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-converter C R C' R'* ===> *rel-converter A B C' R'*)
comp-converter comp-converter
unfolding *comp-converter-def*
supply *inline-parametric'*[*transfer-rule*] *map-gpv-parametric'*[*transfer-rule*] **by** *transfer-prover*

lemma *comp-converter-map-converter1*:
fixes *conv'* :: ('*a*, '*b*, '*out*, '*in*) *converter* **shows**
comp-converter (*map-converter f g h k conv*) *conv'* = *map-converter f g id id* (*comp-converter conv* (*map-converter h k id id conv'*))
using *comp-converter-parametric*[*of*
conversep (*BNF-Def.Grp UNIV f*) *BNF-Def.Grp UNIV g BNF-Def.Grp UNIV h conversep* (*BNF-Def.Grp UNIV k*)
BNF-Def.Grp UNIV (*id* :: '*out* ⇒ -) *conversep* (*BNF-Def.Grp UNIV* (*id* ::

'in ⇒ -))
]
apply(*unfold rel-converter-Grp*)
apply(*simp add*: *rel-fun-def Grp-iff*)
apply(*rewrite at* ∀ - - -. ⌑ ⟶ - **in** *asm conversep-iff*[*symmetric*])
apply(*unfold rel-converter-conversep*[*symmetric*] *conversep-conversep eq-alt*[*symmetric*])
apply(*rewrite* **in** *rel-converter* - - ⌑ - **in** *asm conversep-eq*)
apply(*rewrite* **in** *rel-converter* - - - ⌑ **in** *asm conversep-eq*[*symmetric*])
apply(*rewrite* **in** *rel-converter* - - ⌑ - **in** *asm eq-alt*)
apply(*rewrite* **in** *rel-converter* - - - ⌑ **in** *asm eq-alt*)
apply(*unfold rel-converter-Grp*)
apply(*simp add*: *Grp-iff*)
done

lemma *comp-converter-map-converter2*:
fixes *conv* :: ('*a*, '*b*, '*out*, '*in*) *converter* **shows**
comp-converter conv (*map-converter f g h k conv'*) = *map-converter id id h k*
(*comp-converter* (*map-converter id id f g conv*) *conv'*)
using *comp-converter-parametric*[*of*
BNF-Def.Grp UNIV (*id* :: '*a* ⇒ -) *conversep* (*BNF-Def.Grp UNIV* (*id* :: '*b*
⇒ -))
conversep (*BNF-Def.Grp UNIV f*) *BNF-Def.Grp UNIV g BNF-Def.Grp*
UNIV h conversep (*BNF-Def.Grp UNIV k*)
]
apply(*unfold rel-converter-Grp*)
apply(*simp add*: *rel-fun-def Grp-iff*)
apply(*rewrite at* ∀ - -. ⌑ ⟶ - **in** *asm conversep-iff*[*symmetric*])
apply(*unfold rel-converter-conversep*[*symmetric*] *conversep-conversep rel-converter-Grp*)
apply *simp*
apply(*unfold eq-alt*[*symmetric*])
apply(*rewrite* **in** *rel-converter* - ⌑ **in** *asm conversep-eq*)
apply(*rewrite* **in** *rel-converter* ⌑ - **in** *asm conversep-eq*[*symmetric*])
apply(*rewrite* **in** *rel-converter* ⌑ - **in** *asm eq-alt*)
apply(*rewrite* **in** *rel-converter* - ⌑ **in** *asm eq-alt*)
apply(*unfold rel-converter-Grp*)
apply(*simp add*: *Grp-iff*)
done

lemma *attach-compose*:
attach (*comp-converter conv1 conv2*) *res* = *attach conv1* (*attach conv2 res*)
apply(*coinduction arbitrary*: *conv1 conv2 res*)
apply(*auto intro*!: *rel-funI simp add*: *spmf-rel-map exec-gpv-map-gpv-id exec-gpv-inline*
o-def split-beta)
including *lifting-syntax*
apply(*rule rel-spmf-mono*)
apply(*rule exec-gpv-parametric*[**where** *A*=(=) **and** *CALL*=(=) **and** *S*=λ(*l*, *r*)
s2. *s2* = *attach l r*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
prefer *4*
apply *clarsimp*

by(*auto simp add*: *case-prod-def spmf-rel-map gpv.rel-eq split-def intro*!: *rel-funI rel-spmf-reflI*)

lemma *comp-converter-assoc*:
comp-converter (*comp-converter conv1 conv2*) *conv3* = *comp-converter conv1* (*comp-converter conv2 conv3*)
apply(*coinduction arbitrary*: *conv1 conv2 conv3*)
apply(*rule rel-funI*)
apply(*clarsimp simp add*: *gpv.rel-map inline-map-gpv*)
apply(*subst inline-assoc*)
apply(*simp add*: *gpv.rel-map*)
including *lifting-syntax*
apply(*rule gpv.rel-mono-strong*)
apply(*rule inline-parametric*[**where** *C*=(=) **and** *C'*=(=) **and** *A*=(=) **and** *S*=λ(*l*, *r*) *s2*. *s2* = *comp-converter l r*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
prefer *4*
apply *clarsimp*
by(*auto simp add*: *gpv.rel-eq gpv.rel-map split-beta intro*!: *rel-funI gpv.rel-refl-strong*)

lemma *comp-converter-assoc-left*:
assumes *comp-converter conv1 conv2* = *conv3*
shows *comp-converter conv1* (*comp-converter conv2 conv*) = *comp-converter conv3 conv*
by(*fold comp-converter-assoc*)(*simp add*: *assms*)

lemma *comp-converter-attach-left*:
assumes *comp-converter conv1 conv2* = *conv3*
shows *attach conv1* (*attach conv2 res*) = *attach conv3 res*
by(*fold attach-compose*)(*simp add*: *assms*)

lemmas *comp-converter-eqs* =
asm-rl[**where** *psi*=*x* = *y* **for** *x y* :: (-, -, -, -) *converter*]
comp-converter-assoc-left
comp-converter-attach-left

lemma *WT-converter-comp* [*WT-intro*]:
⟦ $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv* √; $\mathcal{I}', \mathcal{I}'' \vdash_C$ *conv'* √ ⟧ ⟹ $\mathcal{I}, \mathcal{I}'' \vdash_C$ *comp-converter conv conv'* √
by(*coinduction arbitrary*: *conv conv'*)
(*auto*; *auto 4 4 dest*: *WT-converterD run-converter.results-gpv-inline intro*: *run-converter.WT-gpv-inline-invar*[**where** $\mathcal{I}$=$\mathcal{I}'$ **and** $\mathcal{I}'$=$\mathcal{I}''$])

lemma *plossless-comp-converter* [*plossless-intro*]:
assumes *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*
and *plossless-converter* $\mathcal{I}'$ $\mathcal{I}''$ *conv'*
and $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv* √ $\mathcal{I}', \mathcal{I}'' \vdash_C$ *conv'* √

shows *plossless-converter* $\mathcal{I}$ $\mathcal{I}''$ (*comp-converter conv conv'*)
using *assms*
proof(*coinduction arbitrary*: *conv conv'*)
case (*plossless-converter a conv conv'*)
have *conv1*: *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*)
using *plossless-converter*(*1*, *5*) **by**(*simp add*: *plossless-converterD*)
have *conv2*: $\mathcal{I}' \vdash g$ *run-converter conv a* $\surd$
using *plossless-converter*(*3*, *5*) **by**(*simp add*: *WT-converterD*)
have *?plossless* **using** *plossless-converter*(*2*,*4*,*5*)
by(*auto intro*: *run-plossless-converter.plossless-inline*[*OF conv1*] *dest*: *plossless-converterD intro*: *conv2*)
moreover have *?step* (**is** $\forall$ (*b*, *conv'*)$\in$*?res*. *?P b conv'* $\vee$ -)
proof(*clarify*)
fix *b conv''*
assume (*b*, *conv''*) $\in$ *?res*
then obtain *conv1 conv2* **where** [*simp*]: *conv''* = *comp-converter conv1 conv2*

and *inline*: ((*b*, *conv1*), *conv2*) $\in$ *results-gpv* $\mathcal{I}''$ (*inline run-converter* (*run-converter conv a*) *conv'*)
by *auto*
from *run-plossless-converter.results-gpv-inline*[*OF inline conv2*] *plossless-converter*(*2*,*4*)
have *run*: (*b*, *conv1*) $\in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*)
and $*$: *plossless-converter* $\mathcal{I}'$ $\mathcal{I}''$ *conv2* $\mathcal{I}'$, $\mathcal{I}'' \vdash_C$ *conv2* $\surd$ **by** *auto*
with *WT-converterD*(*2*)[*OF plossless-converter*(*3*,*5*) *run*] *plossless-converterD*[*THEN conjunct2*, *rule-format*, *OF plossless-converter*(*1*,*5*) *run*]
show *?P b conv''* **by** *auto*
qed
ultimately show *?case* **..**
qed

lemma *comp-converter-id-left*: *comp-converter id-converter conv* = *conv*
by (*coinduction arbitrary*:*conv*)
(*auto simp add*: *gpv.rel-map split-def map-gpv-conv-bind*[*symmetric*] *intro*!:*rel-funI gpv.rel-refl-strong*)

lemma *comp-converter-id-right*: *comp-converter conv id-converter* = *conv*
proof −
have *lem4*: *inline run-converter gpv id-converter* = *inline id-oracle gpv id-converter* **for** *gpv*
by (*simp only*: *gpv.rel-eq*[*symmetric*])
(*rule gpv.rel-mono-strong*
, *rule inline-parametric*[**where** *A*=(=) **and** *C*=(=) **and** *C'*=(=) **and** *S*=λl *r*. *l* = *r* $\wedge$ *r* = *id-converter*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*]
, *auto simp add*: *id-oracle-def intro*!: *rel-funI gpv.rel-refl-strong*)
show *?thesis*
by (*coinduction arbitrary*:*conv*)
(*auto simp add*: *lem4 gpv.rel-map intro*!:*rel-funI gpv.rel-refl-strong*)
qed

lemma *comp-coverter-of-callee*: *comp-converter* (*converter-of-callee callee1 s1*) (*converter-of-callee callee2 s2*)
= *converter-of-callee* (λ(*s1*, *s2*) *q*. *map-gpv rprodl id* (*inline callee2* (*callee1 s1 q*) *s2*)) (*s1*, *s2*)
apply (*coinduction arbitrary*: *callee1 s1 callee2 s2*)
apply (*rule rel-funI*)
apply (*clarsimp simp add*: *gpv.rel-map inline-map-gpv*)
subgoal for *cal1 s1 cal2 s2 y*
apply (*rule gpv.rel-mono-strong*)
apply (*rule inline-parametric*[**where** *A*=(=) **and** *C*=(=) **and** *C'*=(=) **and** *S*=λ*c s*. *c* = *converter-of-callee cal2 s*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
apply(*auto simp add*: *gpv.rel-eq rel-fun-def gpv.rel-map intro*!: *gpv.rel-refl-strong*)
by (*auto simp add*: *rprodl-def intro*!:*exI*)
done

lemmas *comp-converter-of-callee'* = *comp-converter-eqs*[*OF comp-coverter-of-callee*]

lemma *comp-converter-parallel2*: *comp-converter* (*parallel-converter2 conv1l conv1r*) (*parallel-converter2 conv2l conv2r*) =
parallel-converter2 (*comp-converter conv1l conv2l*) (*comp-converter conv1r conv2r*)
apply (*coinduction arbitrary*: *conv1l conv1r conv2l conv2r*)
apply (*rule rel-funI*)
apply (*clarsimp simp add*: *gpv.rel-map inline-map-gpv split*!: *sum.split*)
subgoal for *conv1l conv1r conv2l conv2r input*
apply(*subst left-gpv-map*[**where** *h*=*id*])
apply(*simp add*: *gpv.rel-map left-gpv-inline*)
apply(*unfold rel-gpv-conv-rel-gpv''*)
apply(*rule rel-gpv''-mono*[*THEN predicate2D*, *rotated* −*1*])
apply(*rule inline-parametric'*[**where** *S*=λ*c1 c2*. *c1* = *parallel-converter2 c2 conv2r* **and** *C*=λ*l r*. *l* = *Inl r* **and** *R*=λ*l r*. *l* = *Inl r* **and** *C'* = (=) **and** *R'*=(=), *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
subgoal by(*auto split*: *sum.split simp add*: *gpv.rel-map rel-gpv-conv-rel-gpv''*[*symmetric*] *intro*!: *gpv.rel-refl-strong rel-funI*)
apply(*rule left-gpv-Inl-transfer*)
apply(*auto 4 6 simp add*: *sum.map-id*)
done
subgoal for *conv1l conv1r conv2l conv2r input*
apply(*subst right-gpv-map*[**where** *h*=*id*])
apply(*simp add*: *gpv.rel-map right-gpv-inline*)
apply(*unfold rel-gpv-conv-rel-gpv''*)
apply(*rule rel-gpv''-mono*[*THEN predicate2D*, *rotated* −*1*])
apply(*rule inline-parametric'*[**where** *S*=λ*c1 c2*. *c1* = *parallel-converter2 conv2l c2* **and** *C*=λ*l r*. *l* = *Inr r* **and** *R*=λ*l r*. *l* = *Inr r* **and** *C'* = (=) **and** *R'*=(=), *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
subgoal by(*auto split*: *sum.split simp add*: *gpv.rel-map rel-gpv-conv-rel-gpv''*[*symmetric*] *intro*!: *gpv.rel-refl-strong rel-funI*)
apply(*rule right-gpv-Inr-transfer*)

apply(*auto 4 6 simp add*: *sum.map-id*)
done
done

lemmas *comp-converter-parallel2'* = *comp-converter-eqs*[*OF comp-converter-parallel2*]

lemma *comp-converter-map1-out*:
comp-converter (*map-converter f g id id conv*) *conv'* = *map-converter f g id id* (*comp-converter conv conv'*)
by(*simp add*: *comp-converter-map-converter1*)

lemma *parallel-converter2-comp1-out*:
parallel-converter2 (*comp-converter conv conv'*) *conv''* = *comp-converter* (*parallel-converter2 conv id-converter*) (*parallel-converter2 conv' conv''*)
by(*simp add*: *comp-converter-parallel2 comp-converter-id-left*)

lemma *parallel-converter2-comp2-out*:
parallel-converter2 conv'' (*comp-converter conv conv'*) = *comp-converter* (*parallel-converter2 id-converter conv*) (*parallel-converter2 conv'' conv'*)
by(*simp add*: *comp-converter-parallel2 comp-converter-id-left*)

2.10 Interaction bound

coinductive *interaction-any-bounded-converter* :: ('*a*, '*b*, '*c*, '*d*) *converter* ⇒ *enat* ⇒ *bool* **where**
interaction-any-bounded-converter conv n **if**
⋀*a*. *interaction-any-bounded-by* (*run-converter conv a*) *n*
⋀*a b conv'*. (*b*, *conv'*) ∈ *results'-gpv* (*run-converter conv a*) ⟹ *interaction-any-bounded-converter conv' n*

lemma *interaction-any-bounded-converterD*:
assumes *interaction-any-bounded-converter conv n*
shows *interaction-any-bounded-by* (*run-converter conv a*) *n* ∧ (∀ (*b*, *conv'*)∈*results'-gpv* (*run-converter conv a*). *interaction-any-bounded-converter conv' n*)
using *assms*
by(*auto elim*: *interaction-any-bounded-converter.cases*)

lemma *interaction-any-bounded-converter-mono*:
assumes *interaction-any-bounded-converter conv n*
and *n* ≤ *m*
shows *interaction-any-bounded-converter conv m*
using *assms*
by(*coinduction arbitrary*: *conv*)(*auto elim*: *interaction-any-bounded-converter.cases intro*: *interaction-bounded-by-mono*)

lemma *interaction-any-bounded-converter-trivial* [*simp*]: *interaction-any-bounded-converter conv* ∞
by(*coinduction arbitrary*: *conv*)
(*auto simp add*: *interaction-bounded-by.simps*)

lemmas *interaction-any-bounded-converter-start* =
interaction-any-bounded-converter-mono
interaction-bounded-by-mono

method *interaction-bound-converter-start* = (*rule interaction-any-bounded-converter-start*)
method *interaction-bound-converter-step* **uses** *add simp* =
((*match* **conclusion in** *interaction-bounded-by* - - - ⇒ *fail* | *interaction-any-bounded-converter* - - ⇒ *fail* | - ⇒ ‹*solves* ‹*clarsimp simp add*: *simp*››) | *rule add interaction-bound*)
method *interaction-bound-converter-rec* **uses** *add simp* =
(*interaction-bound-converter-step add*: *add simp*: *simp*; (*interaction-bound-converter-rec add*: *add simp*: *simp*)*?*)
method *interaction-bound-converter* **uses** *add simp* =
(*interaction-bound-converter-start*, *interaction-bound-converter-rec add*: *add simp*: *simp*)

lemma *interaction-any-bounded-converter-id* [*interaction-bound*]:
interaction-any-bounded-converter id-converter 1
by(*coinduction*) *simp*

lemma *raw-converter-invariant-interaction-any-bounded-converter*:
raw-converter-invariant $\mathcal{I}$*-full* $\mathcal{I}$*-full run-converter* (λ*conv*. *interaction-any-bounded-converter conv n*)
by(*unfold-locales*)(*auto simp add*: *results-gpv-*$\mathcal{I}$*-full dest*: *interaction-any-bounded-converterD*)

lemma *interaction-bounded-by-left-gpv* [*interaction-bound*]:
assumes *interaction-bounded-by consider gpv n*
and ⋀*x*. *consider*′ (*Inl x*) ⟹ *consider x*
shows *interaction-bounded-by consider*′ (*left-gpv gpv*) *n*
proof −
define *ib* :: (′*b*, ′*a*, ′*c*) *gpv* ⇒ - **where** *ib* ≡ *interaction-bound consider*
have *interaction-bound consider*′ (*left-gpv gpv*) ≤ *ib gpv*
proof(*induction arbitrary*: *gpv rule*: *interaction-bound-fixp-induct*)
case (*step interaction-bound*′)
show *?case* **unfolding** *ib-def*
apply(*subst interaction-bound.simps*)
apply(*rule SUP-least*)
apply(*clarsimp split*!: *generat.split if-split*)
apply(*rule SUP-upper2*, *assumption*)
apply(*clarsimp split*!: *if-split simp add*: *assms*(*2*))
apply(*rule SUP-mono*)
subgoal for ... *input*
by(*cases input*)(*auto 4 3 intro*: *step.IH*[*unfolded ib-def*] *order-trans*[*OF step.hyps*(*1*)])
apply(*rule SUP-upper2*, *assumption*)
apply(*clarsimp split*!: *if-split*)
apply(*rule order-trans*, *rule ile-eSuc*)
apply(*simp*)
apply(*rule SUP-mono*)

```
    subgoal for ... input
      by(cases input)(auto 4 3 intro: step.IH[unfolded ib-def] order-trans[OF
step.hyps(1)])
    apply(rule SUP-mono)
    subgoal for ... input
      by(cases input)(auto 4 3 intro: step.IH[unfolded ib-def] order-trans[OF
step.hyps(1)])
    done
  qed simp-all
  then show ?thesis using assms(1)
    by(auto simp add: ib-def interaction-bounded-by.simps intro: order-trans)
qed

lemma interaction-bounded-by-right-gpv [interaction-bound]:
  assumes interaction-bounded-by consider gpv n
    and ⋀x. consider' (Inr x) ⟹ consider x
  shows interaction-bounded-by consider' (right-gpv gpv) n
proof −
  define ib :: ('b, 'a, 'c) gpv ⇒ - where ib ≡ interaction-bound consider
  have interaction-bound consider' (right-gpv gpv) ≤ ib gpv
  proof(induction arbitrary: gpv rule: interaction-bound-fixp-induct)
    case (step interaction-bound')
    show ?case unfolding ib-def
      apply(subst interaction-bound.simps)
      apply(rule SUP-least)
      apply(clarsimp split!: generat.split if-split)
       apply(rule SUP-upper2, assumption)
       apply(clarsimp split!: if-split simp add: assms(2))
       apply(rule SUP-mono)
      subgoal for ... input
        by(cases input)(auto 4 3 intro: step.IH[unfolded ib-def] order-trans[OF
step.hyps(1)])
      apply(rule SUP-upper2, assumption)
      apply(clarsimp split!: if-split)
       apply(rule order-trans, rule ile-eSuc)
       apply(simp)
       apply(rule SUP-mono)
      subgoal for ... input
        by(cases input)(auto 4 3 intro: step.IH[unfolded ib-def] order-trans[OF
step.hyps(1)])
      apply(rule SUP-mono)
      subgoal for ... input
        by(cases input)(auto 4 3 intro: step.IH[unfolded ib-def] order-trans[OF
step.hyps(1)])
      done
  qed simp-all
  then show ?thesis using assms(1)
    by(auto simp add: ib-def interaction-bounded-by.simps intro: order-trans)
qed
```

lemma *interaction-any-bounded-converter-parallel-converter2*:
assumes *interaction-any-bounded-converter conv1 n*
and *interaction-any-bounded-converter conv2 n*
shows *interaction-any-bounded-converter* (*parallel-converter2 conv1 conv2*) *n*
using *assms*
by(*coinduction arbitrary*: *conv1 conv2*)
(*auto 4 4 split*: *sum.split intro*!: *interaction-bounded-by-map-gpv-id intro*: *interaction-bounded-by-left-gpv interaction-bounded-by-right-gpv elim*: *interaction-any-bounded-converter.cases*)

lemma *interaction-any-bounded-converter-parallel-converter2'* [*interaction-bound*]:
assumes *interaction-any-bounded-converter conv1 n*
and *interaction-any-bounded-converter conv2 m*
shows *interaction-any-bounded-converter* (*parallel-converter2 conv1 conv2*) (*max n m*)
by(*rule interaction-any-bounded-converter-parallel-converter2*; *rule assms*[*THEN interaction-any-bounded-converter-mono*]; *simp*)

lemma *interaction-any-bounded-converter-compose* [*interaction-bound*]:
assumes *interaction-any-bounded-converter conv1 n*
and *interaction-any-bounded-converter conv2 m*
shows *interaction-any-bounded-converter* (*comp-converter conv1 conv2*) (*n* ∗ *m*)
proof −
have [*simp*]: ⟦*interaction-any-bounded-converter conv1 n*; *interaction-any-bounded-converter conv2 m*⟧ ⟹
interaction-any-bounded-by (*inline run-converter* (*run-converter conv1 x*) *conv2*) (*n* ∗ *m*) **for** *conv1 conv2 x*
by (*rule interaction-bounded-by-inline-invariant*[**where** *I*=λ*conv2*. *interaction-any-bounded-converter conv2 m* **and** *consider'*=λ-. *True*])
(*auto dest*: *interaction-any-bounded-converterD*)

show *?thesis* **using** *assms*
by(*coinduction arbitrary*: *conv1 conv2*)
((*clarsimp simp add*: *results-gpv-ℐ-full*[*symmetric*] | *intro conjI strip interaction-bounded-by-map-gpv-id*)+
, *drule raw-converter-invariant.results-gpv-inline*[*OF raw-converter-invariant-interaction-any-bounded-con*
, (*rule exI conjI refl WT-gpv-full* | *auto simp add*: *results-gpv-ℐ-full dest*: *interaction-any-bounded-converterD*
raw-converter-invariant.results-gpv-inline[*OF raw-converter-invariant-interaction-any-bounded-converte*
)
qed

lemma *interaction-any-bounded-converter-of-callee* [*interaction-bound*]:
assumes ⋀*s x*. *interaction-any-bounded-by* (*conv s x*) *n*
shows *interaction-any-bounded-converter* (*converter-of-callee conv s*) *n*
by(*coinduction arbitrary*: *s*)(*auto intro*!: *interaction-bounded-by-map-gpv-id assms*)

lemma *interaction-any-bounded-converter-map-converter* [*interaction-bound*]:
assumes *interaction-any-bounded-converter conv n*
and *surj k*

shows *interaction-any-bounded-converter* (*map-converter f g h k conv*) *n*
using *assms*
by(*coinduction arbitrary*: *conv*)
(*auto 4 3 simp add*: *assms results'-gpv-map-gpv'*[*OF* ‹*surj k*›] *intro*: *assms interaction-any-bounded-by-map-gpv' dest*: *interaction-any-bounded-converterD*)

lemma *interaction-any-bounded-converter-parallel-converter*:
assumes *interaction-any-bounded-converter conv1 n*
and *interaction-any-bounded-converter conv2 n*
shows *interaction-any-bounded-converter* (*parallel-converter conv1 conv2*) *n*
using *assms*
by(*coinduction arbitrary*: *conv1 conv2*)
(*auto 4 4 split*: *sum.split intro*!: *interaction-bounded-by-map-gpv-id elim*: *interaction-any-bounded-converter.c*

lemma *interaction-any-bounded-converter-parallel-converter'* [*interaction-bound*]:
assumes *interaction-any-bounded-converter conv1 n*
and *interaction-any-bounded-converter conv2 m*
shows *interaction-any-bounded-converter* (*parallel-converter conv1 conv2*) (*max n m*)
by(*rule interaction-any-bounded-converter-parallel-converter*; *rule assms*[*THEN interaction-any-bounded-converter-mono*]; *simp*)

lemma *interaction-any-bounded-converter-converter-of-resource*:
interaction-any-bounded-converter (*converter-of-resource res*) *n*
by(*coinduction arbitrary*: *res*)(*auto intro*: *interaction-bounded-by-map-gpv-id*)

lemma *interaction-any-bounded-converter-converter-of-resource'* [*interaction-bound*]:
interaction-any-bounded-converter (*converter-of-resource res*) *0*
by(*rule interaction-any-bounded-converter-converter-of-resource*)

lemma *interaction-any-bounded-converter-restrict-converter* [*interaction-bound*]:
interaction-any-bounded-converter (*restrict-converter A* $\mathcal{I}$ *cnv*) *bound*
if *interaction-any-bounded-converter cnv bound*
using *that*
by(*coinduction arbitrary*: *cnv*)
(*auto 4 3 dest*: *interaction-any-bounded-converterD dest*!: *in-results'-gpv-restrict-gpvD intro*!: *interaction-bound*)

end
theory *Converter-Rewrite* **imports**
Converter
begin

3 Equivalence of converters restricted by interfaces

coinductive *eq-resource-on* :: *'a set* ⇒ (*'a*, *'b*) *resource* ⇒ (*'a*, *'b*) *resource* ⇒ *bool* (- $\vdash_R$ / - ∼ / - [*100*, *99*, *99*] *99*)

for A **where**
$\mathit{eq\text{-}resource\text{-}onI}$: $A \vdash_R \mathit{res} \sim \mathit{res}'$ **if**
$\bigwedge a.\ a \in A \Longrightarrow \mathit{rel\text{-}spmf}\ (\mathit{rel\text{-}prod}\ (=)\ (\mathit{eq\text{-}resource\text{-}on}\ A))\ (\mathit{run\text{-}resource}\ \mathit{res}\ a)\ (\mathit{run\text{-}resource}\ \mathit{res}'\ a)$

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}coinduct}$ [$\mathit{consumes}\ 1$, $\mathit{case\text{-}names}\ \mathit{eq\text{-}resource\text{-}on}$, $\mathit{coinduct}\ \mathit{pred}$: $\mathit{eq\text{-}resource\text{-}on}$]:
assumes $X\ \mathit{res}\ \mathit{res}'$
and $\bigwedge \mathit{res}\ \mathit{res}'\ a.\ [\![\ X\ \mathit{res}\ \mathit{res}';\ a \in A\ ]\!]$
$\Longrightarrow \mathit{rel\text{-}spmf}\ (\mathit{rel\text{-}prod}\ (=)\ (\lambda \mathit{res}\ \mathit{res}'.\ X\ \mathit{res}\ \mathit{res}' \vee A \vdash_R \mathit{res} \sim \mathit{res}'))\ (\mathit{run\text{-}resource}\ \mathit{res}\ a)\ (\mathit{run\text{-}resource}\ \mathit{res}'\ a)$
shows $A \vdash_R \mathit{res} \sim \mathit{res}'$
using $\mathit{assms}(1)$ **by**($\mathit{rule}\ \mathit{eq\text{-}resource\text{-}on.coinduct}$)($\mathit{auto}\ \mathit{dest}$: $\mathit{assms}(2)$)

lemma $\mathit{eq\text{-}resource\text{-}onD}$:
assumes $A \vdash_R \mathit{res} \sim \mathit{res}'\ a \in A$
shows $\mathit{rel\text{-}spmf}\ (\mathit{rel\text{-}prod}\ (=)\ (\mathit{eq\text{-}resource\text{-}on}\ A))\ (\mathit{run\text{-}resource}\ \mathit{res}\ a)\ (\mathit{run\text{-}resource}\ \mathit{res}'\ a)$
using assms **by**($\mathit{auto}\ \mathit{elim}$: $\mathit{eq\text{-}resource\text{-}on.cases}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}refl}$ [simp]: $A \vdash_R \mathit{res} \sim \mathit{res}$
by($\mathit{coinduction}\ \mathit{arbitrary}$: res)($\mathit{auto}\ \mathit{intro}$: $\mathit{rel\text{-}spmf\text{-}reflI}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}reflI}$: $\mathit{res} = \mathit{res}' \Longrightarrow A \vdash_R \mathit{res} \sim \mathit{res}'$
by($\mathit{simp}\ \mathit{add}$: $\mathit{eq\text{-}resource\text{-}on\text{-}refl}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}sym}$: $A \vdash_R \mathit{res} \sim \mathit{res}'$ **if** $A \vdash_R \mathit{res}' \sim \mathit{res}$
using that
by($\mathit{coinduction}\ \mathit{arbitrary}$: $\mathit{res}\ \mathit{res}'$)
($\mathit{drule}\ (1)\ \mathit{eq\text{-}resource\text{-}onD}$, $\mathit{rewrite}$ **in** ⌑ $\mathit{conversep\text{-}iff}$ [$\mathit{symmetric}$]
, $\mathit{auto}\ \mathit{simp}\ \mathit{add}$: $\mathit{spmf\text{-}rel\text{-}conversep}$[$\mathit{symmetric}$] elim!: $\mathit{rel\text{-}spmf\text{-}mono}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}trans}$ [trans]: $A \vdash_R \mathit{res} \sim \mathit{res}''$ **if** $A \vdash_R \mathit{res} \sim \mathit{res}'\ A \vdash_R \mathit{res}' \sim \mathit{res}''$
using that **by**($\mathit{coinduction}\ \mathit{arbitrary}$: $\mathit{res}\ \mathit{res}'\ \mathit{res}''$)
(($\mathit{drule}\ (1)\ \mathit{eq\text{-}resource\text{-}onD}$)+, $\mathit{drule}\ (1)\ \mathit{rel\text{-}spmf\text{-}OO\text{-}trans}$, $\mathit{auto}\ \mathit{elim}$!:$\mathit{rel\text{-}spmf\text{-}mono}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}UNIV\text{-}D}$ [simp]: $\mathit{res} = \mathit{res}'$ **if** $\mathit{UNIV} \vdash_R \mathit{res} \sim \mathit{res}'$
using that **by**($\mathit{coinduction}\ \mathit{arbitrary}$: $\mathit{res}\ \mathit{res}'$)($\mathit{auto}\ \mathit{dest}$: $\mathit{eq\text{-}resource\text{-}onD}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}UNIV\text{-}iff}$: $\mathit{UNIV} \vdash_R \mathit{res} \sim \mathit{res}' \longleftrightarrow \mathit{res} = \mathit{res}'$
by($\mathit{auto}\ \mathit{dest}$: $\mathit{eq\text{-}resource\text{-}on\text{-}UNIV\text{-}D}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}mono}$: $[\![\ A' \vdash_R \mathit{res} \sim \mathit{res}';\ A \subseteq A'\ ]\!] \Longrightarrow A \vdash_R \mathit{res} \sim \mathit{res}'$
by($\mathit{coinduction}\ \mathit{arbitrary}$: $\mathit{res}\ \mathit{res}'$)($\mathit{auto}\ \mathit{dest}$: $\mathit{eq\text{-}resource\text{-}onD}\ \mathit{elim}$!: $\mathit{rel\text{-}spmf\text{-}mono}$)

lemma $\mathit{eq\text{-}resource\text{-}on\text{-}empty}$ [simp]: $\{\} \vdash_R \mathit{res} \sim \mathit{res}'$
by($\mathit{rule}\ \mathit{eq\text{-}resource\text{-}onI}$; simp)

lemma *eq-resource-on-resource-of-oracleI*:
includes *lifting-syntax*
fixes S
assumes *sim*: (*S* ===> *eq-on A* ===> *rel-spmf* (*rel-prod* (=) *S*)) *r1 r2*
and *S*: *S s1 s2*
shows $A \vdash_R$ *resource-of-oracle r1 s1* $\sim$ *resource-of-oracle r2 s2*
using *S* **by**(*coinduction arbitrary*: *s1 s2*)
(*drule sim*[*THEN rel-funD*, *THEN rel-funD*], *simp add*: *eq-on-def*
, *fastforce simp add*: *eq-on-def spmf-rel-map elim*: *rel-spmf-mono*)

lemma *exec-gpv-eq-resource-on*:
assumes *outs-*$\mathcal{I}$ $\mathcal{I} \vdash_R$ *res* $\sim$ *res'*
and $\mathcal{I} \vdash g$ *gpv* $\surd$
and $\mathcal{I} \vdash$*res res* $\surd$
shows *rel-spmf* (*rel-prod* (=) (*eq-resource-on* (*outs-*$\mathcal{I}$ $\mathcal{I}$))) (*exec-gpv run-resource gpv res*) (*exec-gpv run-resource gpv res'*)
using *assms*
proof(*induction arbitrary*: *res res' gpv rule*: *exec-gpv-fixp-induct*)
case (*step exec-gpv'*)
have[*simp*]: ⟦(*s*, *r1*) ∈ *set-spmf* (*run-resource res g1*); (*s*, *r2*) ∈ *set-spmf* (*run-resource res' g1*);
IO g1 g2 ∈ *set-spmf* (*the-gpv gpv*); *outs-*$\mathcal{I}$ $\mathcal{I} \vdash_R$ *r1* $\sim$ *r2*⟧ ⟹ *rel-spmf* (*rel-prod* (=) (*eq-resource-on* (*outs-*$\mathcal{I}$ $\mathcal{I}$)))
(*exec-gpv'* (*g2 s*) *r1*) (*exec-gpv'* (*g2 s*) *r2*) **for** *g1 g2 r1 s r2*
by(*rule step.IH*, *simp*, *rule WT-gpv-ContD*[*OF step.prems*(*2*)], *assumption*)
(*auto elim*: *outs-gpv.IO WT-calleeD*[*OF run-resource.WT-callee*, *OF step.prems*(*3*)]
dest!: *WT-resourceD*[*OF step.prems*(*3*), *rotated 1*] *intro*: *WT-gpv-outs-gpv*[*THEN subsetD*, *OF step.prems*(*2*)])

show *?case*
by(*clarsimp intro*!: *rel-spmf-bind-reflI step.prems split*!: *generat.split*)
(*rule rel-spmf-bindI'*, *rule eq-resource-onD*[*OF step.prems*(*1*)]
, *auto elim*: *outs-gpv.IO intro*: *eq-resource-onD*[*OF step.prems*(*1*)] *WT-gpv-outs-gpv*[*THEN subsetD*, *OF step.prems*(*2*)])
qed *simp-all*

inductive *eq-*$\mathcal{I}$*-generat* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*out*, '*in*) $\mathcal{I}$ ⇒ ('*c* ⇒ '*d* ⇒ *bool*) ⇒ ('*a*, '*out*, '*in* ⇒ '*c*) *generat* ⇒ ('*b*, '*out*, '*in* ⇒ '*d*) *generat* ⇒ *bool*
for *A* $\mathcal{I}$ *D* **where**
Pure: *eq-*$\mathcal{I}$*-generat A* $\mathcal{I}$ *D* (*Pure x*) (*Pure y*) **if** *A x y*
| *IO*: *eq-*$\mathcal{I}$*-generat A* $\mathcal{I}$ *D* (*IO out c*) (*IO out c'*) **if** *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⋀*input*. *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* ⟹ *D* (*c input*) (*c' input*)

hide-fact (**open**) *Pure IO*

inductive-simps *eq-*$\mathcal{I}$*-generat-simps* [*simp*, *code*]:
eq-$\mathcal{I}$*-generat A* $\mathcal{I}$ *D* (*Pure x*) (*Pure y*)
eq-$\mathcal{I}$*-generat A* $\mathcal{I}$ *D* (*IO out c*) (*Pure y*)

eq-$\mathcal{I}$-generat A $\mathcal{I}$ D (Pure x) (IO out' c')
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D (IO out c) (IO out' c')

inductive-simps *eq-$\mathcal{I}$-generat-iff1*:
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D (Pure x) g'
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D (IO out c) g'

inductive-simps *eq-$\mathcal{I}$-generat-iff2*:
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D g (Pure x)
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D g (IO out c)

lemma *eq-$\mathcal{I}$-generat-mono'*:
⟦ *eq-$\mathcal{I}$-generat A $\mathcal{I}$ D x y*; $\bigwedge$*x y. A x y* $\Longrightarrow$ *A' x y*; $\bigwedge$*x y. D x y* $\Longrightarrow$ *D' x y*; $\mathcal{I} \leq \mathcal{I}'$ ⟧
$\Longrightarrow$ *eq-$\mathcal{I}$-generat A' $\mathcal{I}'$ D' x y*
by(*auto 4 4 elim*!: *eq-$\mathcal{I}$-generat.cases simp add*: *le-$\mathcal{I}$-def*)

lemma *eq-$\mathcal{I}$-generat-mono*: *eq-$\mathcal{I}$-generat A $\mathcal{I}$ D* $\leq$ *eq-$\mathcal{I}$-generat A' $\mathcal{I}'$ D'* **if** $A \leq A'$ $D \leq D'$ $\mathcal{I} \leq \mathcal{I}'$
using *that* **by**(*auto elim*!: *eq-$\mathcal{I}$-generat-mono' dest*: *predicate2D*)

lemma *eq-$\mathcal{I}$-generat-mono''* [*mono*]:
⟦ $\bigwedge$*x y. A x y* $\longrightarrow$ *A' x y*; $\bigwedge$*x y. D x y* $\longrightarrow$ *D' x y* ⟧
$\Longrightarrow$ *eq-$\mathcal{I}$-generat A $\mathcal{I}$ D x y* $\longrightarrow$ *eq-$\mathcal{I}$-generat A' $\mathcal{I}$ D' x y*
by(*auto elim*: *eq-$\mathcal{I}$-generat-mono'*)

lemma *eq-$\mathcal{I}$-generat-conversep*: *eq-$\mathcal{I}$-generat* A^{-1-1} $\mathcal{I}$ D^{-1-1} = (*eq-$\mathcal{I}$-generat A $\mathcal{I}$ D*)$^{-1-1}$
by(*fastforce elim*: *eq-$\mathcal{I}$-generat.cases*)

lemma *eq-$\mathcal{I}$-generat-reflI*:
assumes $\bigwedge$*x. x* $\in$ *generat-pures generat* $\Longrightarrow$ *A x x*
and $\bigwedge$*out c. generat = IO out c* $\Longrightarrow$ *out* $\in$ *outs-$\mathcal{I}$ $\mathcal{I}$* $\wedge$ ($\forall$ *input*$\in$*responses-$\mathcal{I}$ $\mathcal{I}$ out. D (c input) (c input)*)
shows *eq-$\mathcal{I}$-generat A $\mathcal{I}$ D generat generat*
using *assms* **by**(*cases generat*) *auto*

lemma *eq-$\mathcal{I}$-generat-relcompp*:
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D OO eq-$\mathcal{I}$-generat A' $\mathcal{I}$ D' = eq-$\mathcal{I}$-generat (A OO A') $\mathcal{I}$ (D OO D')
by(*auto 4 3 intro*!: *ext elim*!: *eq-$\mathcal{I}$-generat.cases simp add*: *eq-$\mathcal{I}$-generat-iff1 eq-$\mathcal{I}$-generat-iff2 relcompp.simps*) *metis*

lemma *eq-$\mathcal{I}$-generat-map1*:
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D (map-generat f id (∘) g) generat) generat' $\longleftrightarrow$
eq-$\mathcal{I}$-generat (λ*x. A (f x)*) $\mathcal{I}$ (λ*x. D (g x)*) *generat generat'*
by(*cases generat*; *cases generat'*) *auto*

lemma *eq-$\mathcal{I}$-generat-map2*:

eq-ℐ-generat A ℐ D generat (map-generat f id ((∘) g) generat′) ⟷
eq-ℐ-generat (λx y. A x (f y)) ℐ (λx y. D x (g y)) generat generat′
by(*cases generat*; *cases generat′*) *auto*

lemmas *eq-ℐ-generat-map* [*simp*] =
eq-ℐ-generat-map1[*abs-def*] *eq-ℐ-generat-map2*
eq-ℐ-generat-map1[**where** *g*=*id*, *unfolded fun.map-id0*, *abs-def*] *eq-ℐ-generat-map2*[**where** *g*=*id*, *unfolded fun.map-id0*]

lemma *eq-ℐ-generat-into-rel-generat*:
eq-ℐ-generat A ℐ-full D generat generat′ ⟹ rel-generat A (=) (rel-fun (=) D) generat generat′
by(*erule eq-ℐ-generat.cases*) *auto*

coinductive *eq-ℐ-gpv* :: (′*a* ⇒ ′*b* ⇒ *bool*) ⇒ (′*out*, ′*in*) ℐ ⇒ (′*a*, ′*out*, ′*in*) *gpv* ⇒ (′*b*, ′*out*, ′*in*) *gpv* ⇒ *bool*
for *A ℐ* **where**
eq-ℐ-gpvI: *eq-ℐ-gpv A ℐ gpv gpv′*
if *rel-spmf* (*eq-ℐ-generat A ℐ* (*eq-ℐ-gpv A ℐ*)) (*the-gpv gpv*) (*the-gpv gpv′*)

lemma *eq-ℐ-gpv-coinduct* [*consumes 1*, *case-names eq-ℐ-gpv*, *coinduct pred*: *eq-ℐ-gpv*]:
assumes *X gpv gpv′*
and ⋀*gpv gpv′*. *X gpv gpv′*
⟹ *rel-spmf* (*eq-ℐ-generat A ℐ* (λ*gpv gpv′*. *X gpv gpv′* ∨ *eq-ℐ-gpv A ℐ gpv gpv′*)) (*the-gpv gpv*) (*the-gpv gpv′*)
shows *eq-ℐ-gpv A ℐ gpv gpv′*
using *assms*(*1*) **by**(*rule eq-ℐ-gpv.coinduct*)(*blast dest*: *assms*(*2*))

lemma *eq-ℐ-gpvD*:
eq-ℐ-gpv A ℐ gpv gpv′ ⟹ rel-spmf (*eq-ℐ-generat A ℐ* (*eq-ℐ-gpv A ℐ*)) (*the-gpv gpv*) (*the-gpv gpv′*)
by(*blast elim*!: *eq-ℐ-gpv.cases*)

lemma *eq-ℐ-gpv-Done* [*intro*!]: *A x y ⟹ eq-ℐ-gpv A ℐ* (*Done x*) (*Done y*)
by(*rule eq-ℐ-gpvI*) *simp*

lemma *eq-ℐ-gpv-Done-iff* [*simp*]: *eq-ℐ-gpv A ℐ* (*Done x*) (*Done y*) ⟷ *A x y*
by(*auto dest*: *eq-ℐ-gpvD*)

lemma *eq-ℐ-gpv-Pause*:
⟦ *out* ∈ *outs-ℐ ℐ*; ⋀*input*. *input* ∈ *responses-ℐ ℐ out ⟹ eq-ℐ-gpv A ℐ* (*rpv input*) (*rpv′ input*) ⟧
⟹ *eq-ℐ-gpv A ℐ* (*Pause out rpv*) (*Pause out rpv′*)
by(*rule eq-ℐ-gpvI*) *simp*

lemma *eq-ℐ-gpv-mono*: *eq-ℐ-gpv A ℐ* ≤ *eq-ℐ-gpv A′ ℐ′* **if** *A*: *A* ≤ *A′ ℐ* ≤ *ℐ′*
proof
show *eq-ℐ-gpv A′ ℐ′ gpv gpv′* **if** *eq-ℐ-gpv A ℐ gpv gpv′* **for** *gpv gpv′* **using** *that*
by(*coinduction arbitrary*: *gpv gpv′*)

(*drule eq-ℐ-gpvD*, *auto dest*: *eq-ℐ-gpvD elim*: *rel-spmf-mono eq-ℐ-generat-mono*[*OF A*(*1*) - *A*(*2*), *THEN predicate2D*, *rotated* −*1*])
qed

lemma *eq-ℐ-gpv-mono′*:
⟦ *eq-ℐ-gpv A ℐ gpv gpv′*; ⋀*x y*. *A x y* ⟹ *A′ x y*; *ℐ* ≤ *ℐ′* ⟧ ⟹ *eq-ℐ-gpv A′ ℐ′ gpv gpv′*
by(*blast intro*: *eq-ℐ-gpv-mono*[*THEN predicate2D*])

lemma *eq-ℐ-gpv-mono″* [*mono*]:
eq-ℐ-gpv A ℐ gpv gpv′ ⟶ *eq-ℐ-gpv A′ ℐ gpv gpv′* **if** ⋀*x y*. *A x y* ⟶ *A′ x y*
using *that* **by**(*blast intro*: *eq-ℐ-gpv-mono′*)

lemma *eq-ℐ-gpv-conversep*: *eq-ℐ-gpv A*$^{-1-1}$ *ℐ* = (*eq-ℐ-gpv A ℐ*)$^{-1-1}$
proof(*intro ext iffI*; *simp*)
show *eq-ℐ-gpv A ℐ gpv gpv′* **if** *eq-ℐ-gpv A*$^{-1-1}$ *ℐ gpv′ gpv* **for** *A* **and** *gpv gpv′*
using *that*
by(*coinduction arbitrary*: *gpv gpv′*)
(*drule eq-ℐ-gpvD*, *rewrite* **in** ¤ *conversep-iff*[*symmetric*]
, *auto simp add*: *pmf.rel-conversep*[*symmetric*] *option.rel-conversep*[*symmetric*] *eq-ℐ-generat-conversep*[*symmetric*] *elim*: *eq-ℐ-generat-mono′ rel-spmf-mono*)

from *this*[*of conversep A*] **show** *eq-ℐ-gpv A*$^{-1-1}$ *ℐ gpv′ gpv* **if** *eq-ℐ-gpv A ℐ gpv gpv′* **for** *gpv gpv′*
using *that* **by** *simp*
qed

lemma *eq-ℐ-gpv-reflI*:
⟦ ⋀*x*. *x* ∈ *results-gpv ℐ gpv* ⟹ *A x x*; *ℐ* ⊢*g gpv* √ ⟧ ⟹ *eq-ℐ-gpv A ℐ gpv gpv*
by(*coinduction arbitrary*: *gpv*)(*auto intro*!: *rel-spmf-reflI eq-ℐ-generat-reflI elim*!: *generat.set-cases intro*: *results-gpv.intros dest*: *WT-gpvD*)

lemma *eq-ℐ-gpv-into-rel-gpv*: *eq-ℐ-gpv A ℐ-full gpv gpv′* ⟹ *rel-gpv A* (=) *gpv gpv′*
by(*coinduction arbitrary*: *gpv gpv′*)
(*drule eq-ℐ-gpvD*, *auto elim*: *spmf-rel-mono-strong generat.rel-mono-strong dest*: *eq-ℐ-generat-into-rel-generat*)

lemma *eq-ℐ-gpv-relcompp*: *eq-ℐ-gpv* (*A OO A′*) *ℐ* = *eq-ℐ-gpv A ℐ OO eq-ℐ-gpv A′ ℐ* (**is** *?lhs* = *?rhs*)
proof(*intro ext iffI relcomppI*; (*elim relcomppE*)*?*)
fix *gpv gpv″*
assume ∗: *?lhs gpv gpv″*
define *middle* **where** *middle* = *corec-gpv* (λ(*gpv*, *gpv″*).
map-spmf (*map-generat* (*relcompp-witness A A′*) (*relcompp-witness* (=) (=)) ((∘) *Inr* ∘ *rel-witness-fun* (=) (=)) ∘
rel-witness-generat)
(*rel-witness-spmf* (*eq-ℐ-generat* (*A OO A′*) *ℐ* (*eq-ℐ-gpv* (*A OO A′*) *ℐ*)) (*the-gpv gpv*, *the-gpv gpv″*)))

have *middle-sel* [*simp*]: *the-gpv* (*middle* (*gpv*, *gpv''*)) =
map-spmf (*map-generat* (*relcompp-witness A A'*) (*relcompp-witness* (=) (=))
((∘) *middle* ∘ *rel-witness-fun* (=) (=)) ∘
rel-witness-generat)
(*rel-witness-spmf* (*eq-ℐ-generat* (*A OO A'*) *ℐ* (*eq-ℐ-gpv* (*A OO A'*) *ℐ*)) (*the-gpv gpv*, *the-gpv gpv''*))
for *gpv gpv''* **by**(*auto simp add*: *middle-def spmf.map-comp o-def generat.map-comp*)
show *eq-ℐ-gpv A ℐ gpv* (*middle* (*gpv*, *gpv''*)) **using** ∗
by(*coinduction arbitrary*: *gpv gpv''*)
(*drule eq-ℐ-gpvD*, *simp add*: *spmf-rel-map*, *erule rel-witness-spmf1*[*THEN rel-spmf-mono*]
, *auto 4 3 del*: *relcomppE elim*!: *relcompp-witness eq-ℐ-generat.cases*)

show *eq-ℐ-gpv A' ℐ* (*middle* (*gpv*, *gpv''*)) *gpv''* **using** ∗
by(*coinduction arbitrary*: *gpv gpv''*)
(*drule eq-ℐ-gpvD*, *simp add*: *spmf-rel-map*, *erule rel-witness-spmf2*[*THEN rel-spmf-mono*]
, *auto 4 3 del*: *relcomppE elim*: *rel-witness-spmf2*[*THEN rel-spmf-mono*] *elim*!: *relcompp-witness eq-ℐ-generat.cases*)
next
show *?lhs gpv gpv''* **if** *eq-ℐ-gpv A ℐ gpv gpv'* **and** *eq-ℐ-gpv A' ℐ gpv' gpv''* **for** *gpv gpv' gpv''* **using** *that*
by(*coinduction arbitrary*: *gpv gpv' gpv''*)
((*drule eq-ℐ-gpvD*)+, *simp*, *drule* (*1*) *rel-spmf-OO-trans*, *erule rel-spmf-mono*
, *auto simp add*: *eq-ℐ-generat-relcompp elim*: *eq-ℐ-generat-mono'*)
qed

lemma *eq-ℐ-gpv-map-gpv1*: *eq-ℐ-gpv A ℐ* (*map-gpv f id gpv*) *gpv'* ⟷ *eq-ℐ-gpv* (λ*x*. *A* (*f x*)) *ℐ gpv gpv'* (**is** *?lhs* ⟷ *?rhs*)
proof
show *?rhs* **if** *?lhs* **using** *that*
by(*coinduction arbitrary*: *gpv gpv'*)
(*drule eq-ℐ-gpvD*, *auto simp add*: *gpv.map-sel spmf-rel-map elim*!: *rel-spmf-mono eq-ℐ-generat-mono'*)
show *?lhs* **if** *?rhs* **using** *that*
by(*coinduction arbitrary*: *gpv gpv'*)
(*drule eq-ℐ-gpvD*, *auto simp add*: *gpv.map-sel spmf-rel-map elim*!: *rel-spmf-mono eq-ℐ-generat-mono'*)
qed

lemma *eq-ℐ-gpv-map-gpv2*: *eq-ℐ-gpv A ℐ gpv* (*map-gpv f id gpv'*) = *eq-ℐ-gpv* (λ*x y*. *A x* (*f y*)) *ℐ gpv gpv'*
using *eq-ℐ-gpv-map-gpv1*[*of conversep A ℐ f gpv' gpv*]
by(*rewrite* **in** - = ¤ *conversep-iff*[*symmetric*] , *simp add*: *eq-ℐ-gpv-conversep*[*symmetric*])
(*subst* (*asm*) *eq-ℐ-gpv-conversep* , *simp add*: *conversep-iff*[*abs-def*])

lemmas *eq-ℐ-gpv-map-gpv* [*simp*] = *eq-ℐ-gpv-map-gpv1*[*abs-def*] *eq-ℐ-gpv-map-gpv2*

lemma (**in** *callee-invariant-on*) *eq-ℐ-exec-gpv*:

⟦ *eq-ℐ-gpv A ℐ gpv gpv'*; *I s* ⟧ ⟹ *rel-spmf* (*rel-prod A* (*eq-onp I*)) (*exec-gpv callee gpv s*) (*exec-gpv callee gpv' s*)
proof(*induction arbitrary*: *s gpv gpv' rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf exec-gpv.mono exec-gpv.mono exec-gpv-def exec-gpv-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv' exec-gpv''*)
show *?case* **using** *step.prems*
by − (*drule eq-ℐ-gpvD*, *erule rel-spmf-bindI*
, *auto split*!: *generat.split simp add*: *eq-onp-same-args*
intro: *WT-callee*[*THEN WT-calleeD*] *callee-invariant step.IH intro*!: *rel-spmf-bind-reflI*)
qed

lemma *eq-ℐ-gpv-coinduct-bind* [*consumes 1*, *case-names eq-ℐ-gpv*]:
fixes *gpv* :: ('*a*, '*out*, '*in*) *gpv* **and** *gpv'* :: ('*a'*, '*out*, '*in*) *gpv*
assumes *X*: *X gpv gpv'*
and *step*: ⋀*gpv gpv'*. *X gpv gpv'*
⟹ *rel-spmf* (*eq-ℐ-generat A ℐ* (λ*gpv gpv'*. *X gpv gpv'* ∨ *eq-ℐ-gpv A ℐ gpv gpv'* ∨
(∃ *gpv'' gpv'''* (*B* :: '*b* ⇒ '*b'* ⇒ *bool*) *f g*. *gpv* = *bind-gpv gpv'' f* ∧ *gpv'* = *bind-gpv gpv''' g* ∧ *eq-ℐ-gpv B ℐ gpv'' gpv'''* ∧ (*rel-fun B X*) *f g*))) (*the-gpv gpv*) (*the-gpv gpv'*)
shows *eq-ℐ-gpv A ℐ gpv gpv'*
proof −
fix *x y*
define *gpv''* :: ('*b*, '*out*, '*in*) *gpv* **where** *gpv''* ≡ *Done x*
define *f* :: '*b* ⇒ ('*a*, '*out*, '*in*) *gpv* **where** *f* ≡ λ-. *gpv*
define *gpv'''* :: ('*b'*, '*out*, '*in*) *gpv* **where** *gpv'''* ≡ *Done y*
define *g* :: '*b'* ⇒ ('*a'*, '*out*, '*in*) *gpv* **where** *g* ≡ λ-. *gpv'*
have *eq-ℐ-gpv* (λ*x y*. *X* (*f x*) (*g y*)) *ℐ gpv'' gpv'''* **using** *X*
by(*simp add*: *f-def g-def gpv''-def gpv'''-def*)
then have *eq-ℐ-gpv A ℐ* (*bind-gpv gpv'' f*) (*bind-gpv gpv''' g*)
by(*coinduction arbitrary*: *gpv'' f gpv''' g*)
(*drule eq-ℐ-gpvD*, (*clarsimp simp add*: *bind-gpv.sel spmf-rel-map simp del*: *bind-gpv-sel'* *elim*!: *rel-spmf-bindI split*!: *generat.split dest*!: *step*)
, *erule rel-spmf-mono*, (*erule eq-ℐ-generat.cases*; *clarsimp*), (*erule meta-allE*, *erule* (*1*) *meta-impE*)
, (*fastforce* | *force intro*: *exI*[**where** *x*=*Done* -] *elim*!: *eq-ℐ-gpv-mono'* *dest*: *rel-funD*)+)

then show *?thesis* **unfolding** *gpv''-def gpv'''-def f-def g-def* **by** *simp*
qed

context
fixes *S* :: '*s1* ⇒ '*s2* ⇒ *bool*
and *callee1* :: '*s1* ⇒ '*out* ⇒ ('*in* × '*s1*, '*out'*, '*in'*) *gpv*
and *callee2* :: '*s2* ⇒ '*out* ⇒ ('*in* × '*s2*, '*out'*, '*in'*) *gpv*
and *ℐ* :: ('*out*, '*in*) *ℐ*

and $\mathcal{I}'$ :: ('*out*', '*in*') $\mathcal{I}$
assumes *callee*: $\bigwedge$*s1 s2 q*. ⟦ *S s1 s2*; *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *eq-*$\mathcal{I}$*-gpv* (*rel-prod* (*eq-onp* (λ*r*. *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q*)) *S*) $\mathcal{I}'$ (*callee1 s1 q*) (*callee2 s2 q*)
begin

lemma *eq-*$\mathcal{I}$*-gpv-inline1*:
includes *lifting-syntax*
assumes *S s1 s2 eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv1 gpv2*
shows *rel-spmf* (*rel-sum* (*rel-prod A S*)
(λ(*q*, *rpv1*, *rpv2*) (*q'*, *rpv1'*, *rpv2'*). *q* = *q'* ∧ *q'* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}'$ ∧ (∃ *q''* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$.
(∀ *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *q'*. *eq-*$\mathcal{I}$*-gpv* (*rel-prod* (*eq-onp* (λ*r'*. *r'* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q''*)) *S*) $\mathcal{I}'$ (*rpv1 r*) (*rpv1' r*)) ∧
(∀ *r'* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q''*. *eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ (*rpv2 r'*) (*rpv2' r'*)))))
(*inline1 callee1 gpv1 s1*) (*inline1 callee2 gpv2 s2*)
using *assms*
proof(*induction arbitrary*: *gpv1 gpv2 s1 s2 rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf inline1.mono inline1.mono inline1-def inline1-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1' inline1''*)
from *step.prems* **show** *?case*
by − (*erule eq-*$\mathcal{I}$*-gpvD*[*THEN rel-spmf-bindI*]
, *clarsimp split*!: *generat.split*
, *erule eq-*$\mathcal{I}$*-gpvD*[*OF callee*(*1*), *THEN rel-spmf-bindI*]
, *auto simp add*: *eq-onp-def intro*: *step.IH*[*THEN rel-spmf-mono*] *elim*: *eq-*$\mathcal{I}$*-gpvD*[*OF callee*(*1*), *THEN rel-spmf-bindI*] *split*!: *generat.split*)
qed

lemma *eq-*$\mathcal{I}$*-gpv-inline*:
assumes *S*: *S s1 s2*
and *gpv*: *eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv1 gpv2*
shows *eq-*$\mathcal{I}$*-gpv* (*rel-prod A S*) $\mathcal{I}'$ (*inline callee1 gpv1 s1*) (*inline callee2 gpv2 s2*)
using *S gpv*
by (*coinduction arbitrary*: *gpv1 gpv2 s1 s2 rule*: *eq-*$\mathcal{I}$*-gpv-coinduct-bind*)
(*clarsimp simp add*: *inline-sel spmf-rel-map*, *drule* (*1*) *eq-*$\mathcal{I}$*-gpv-inline1*
, *fastforce split*!: *generat.split sum.split del*: *disjCI intro*!: *disjI2 rel-funI elim*: *rel-spmf-mono simp add*: *eq-onp-def*)

end

lemma *eq-*$\mathcal{I}$*-gpv-left-gpv-cong*:
eq-$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv gpv'* ⟹ *eq-*$\mathcal{I}$*-gpv A* ($\mathcal{I}$ ⊕$_\mathcal{I}$ $\mathcal{I}'$) (*left-gpv gpv*) (*left-gpv gpv'*)
by(*coinduction arbitrary*: *gpv gpv'*)
(*drule eq-*$\mathcal{I}$*-gpvD*, *auto 4 4 simp add*: *spmf-rel-map elim*!: *rel-spmf-mono eq-*$\mathcal{I}$*-generat.cases*)

lemma *eq-*$\mathcal{I}$*-gpv-right-gpv-cong*:
eq-$\mathcal{I}$*-gpv A* $\mathcal{I}'$ *gpv gpv'* ⟹ *eq-*$\mathcal{I}$*-gpv A* ($\mathcal{I}$ ⊕$_\mathcal{I}$ $\mathcal{I}'$) (*right-gpv gpv*) (*right-gpv gpv'*)

by(*coinduction arbitrary*: *gpv gpv'*)
(*drule eq-ℐ-gpvD*, *auto 4 4 simp add*: *spmf-rel-map elim*!: *rel-spmf-mono eq-ℐ-generat.cases*)

lemma *eq-ℐ-gpvD-WT1*: ⟦ *eq-ℐ-gpv A ℐ gpv gpv'*; *ℐ* ⊢*g gpv* √ ⟧ ⟹ *ℐ* ⊢*g gpv'* √
by(*coinduction arbitrary*: *gpv gpv'*)(*fastforce simp add*: *eq-ℐ-generat-iff2 dest*: *WT-gpv-ContD eq-ℐ-gpvD dest*!: *rel-setD2 set-spmf-parametric*[*THEN rel-funD*])

lemma *eq-ℐ-gpvD-results-gpv2*:
assumes *eq-ℐ-gpv A ℐ gpv gpv' y* ∈ *results-gpv ℐ gpv'*
shows ∃ *x* ∈ *results-gpv ℐ gpv*. *A x y*
using *assms*(*2*,*1*)
by(*induction arbitrary*: *gpv*)
(*fastforce dest*!: *set-spmf-parametric*[*THEN rel-funD*] *rel-setD2 dest*: *eq-ℐ-gpvD simp add*: *eq-ℐ-generat-iff2 intro*: *results-gpv.intros*)+

coinductive *eq-ℐ-converter* :: ('*a*, '*b*) *ℐ* ⇒ ('*out*, '*in*) *ℐ* ⇒ ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ *bool*
(-,- ⊢$_C$/ - ∼/ - [*100*, *0*, *99*, *99*] *99*)
for *ℐ ℐ'* **where**
eq-ℐ-converterI: *ℐ*, *ℐ'* ⊢$_C$ *conv* ∼ *conv'* **if**
⋀*q*. *q* ∈ *outs-ℐ ℐ* ⟹ *eq-ℐ-gpv* (*rel-prod* (*eq-onp* (λ*r*. *r* ∈ *responses-ℐ ℐ q*)) (*eq-ℐ-converter ℐ ℐ'*)) *ℐ'* (*run-converter conv q*) (*run-converter conv' q*)

lemma *eq-ℐ-converter-coinduct* [*consumes 1*, *case-names eq-ℐ-converter*, *coinduct pred*: *eq-ℐ-converter*]:
assumes *X conv conv'*
and ⋀*conv conv' q*. ⟦ *X conv conv'*; *q* ∈ *outs-ℐ ℐ* ⟧
⟹ *eq-ℐ-gpv* (*rel-prod* (*eq-onp* (λ*r*. *r* ∈ *responses-ℐ ℐ q*)) (λ*conv conv'*. *X conv conv'* ∨ *ℐ*, *ℐ'* ⊢$_C$ *conv* ∼ *conv'*)) *ℐ'*
(*run-converter conv q*) (*run-converter conv' q*)
shows *ℐ*, *ℐ'* ⊢$_C$ *conv* ∼ *conv'*
using *assms*(*1*) **by**(*rule eq-ℐ-converter.coinduct*)(*blast dest*: *assms*(*2*))

lemma *eq-ℐ-converterD*:
eq-ℐ-gpv (*rel-prod* (*eq-onp* (λ*r*. *r* ∈ *responses-ℐ ℐ q*)) (*eq-ℐ-converter ℐ ℐ'*)) *ℐ'* (*run-converter conv q*) (*run-converter conv' q*)
if *ℐ*, *ℐ'* ⊢$_C$ *conv* ∼ *conv' q* ∈ *outs-ℐ ℐ*
using *that* **by**(*blast elim*: *eq-ℐ-converter.cases*)

lemma *eq-ℐ-converter-reflI*: *ℐ*, *ℐ'* ⊢$_C$ *conv* ∼ *conv* **if** *ℐ*, *ℐ'* ⊢$_C$ *conv* √
using *that* **by**(*coinduction arbitrary*: *conv*)(*auto intro*!: *eq-ℐ-gpv-reflI dest*: *WT-converterD simp add*: *eq-onp-same-args*)

lemma *eq-ℐ-converter-sym* [*sym*]: *ℐ*, *ℐ'* ⊢$_C$ *conv* ∼ *conv'* **if** *ℐ*, *ℐ'* ⊢$_C$ *conv'* ∼ *conv*
using *that*
by(*coinduction arbitrary*: *conv conv'*)
(*drule* (*1*) *eq-ℐ-converterD*, *rewrite* **in** ¤ *conversep-iff*[*symmetric*]
, *auto simp add*: *eq-ℐ-gpv-conversep*[*symmetric*] *eq-onp-def elim*: *eq-ℐ-gpv-mono'*)

lemma *eq-$\mathcal{I}$-converter-trans* [*trans*]:
⟦ $\mathcal{I}, \mathcal{I}' \vdash_C conv \sim conv'$; $\mathcal{I}, \mathcal{I}' \vdash_C conv' \sim conv''$ ⟧ ⟹ $\mathcal{I}, \mathcal{I}' \vdash_C conv \sim conv''$
by(*coinduction arbitrary*: *conv conv' conv''*)
((*drule* (*1*) *eq-$\mathcal{I}$-converterD*)+, *drule* (*1*) *eq-$\mathcal{I}$-gpv-relcompp*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*, *OF relcomppI*]
, *auto simp add*: *eq-OO prod.rel-compp*[*symmetric*] *eq-onp-def elim*!: *eq-$\mathcal{I}$-gpv-mono'*)

lemma *eq-$\mathcal{I}$-converter-mono*:
assumes ∗: $\mathcal{I}1, \mathcal{I}2 \vdash_C conv \sim conv'$
and *le*: $\mathcal{I}1' \leq \mathcal{I}1$ $\mathcal{I}2 \leq \mathcal{I}2'$
shows $\mathcal{I}1', \mathcal{I}2' \vdash_C conv \sim conv'$
using ∗
by(*coinduction arbitrary*: *conv conv'*)
(*auto simp add*: *eq-onp-def dest*!:*eq-$\mathcal{I}$-converterD intro*: *responses-$\mathcal{I}$-mono*[*THEN subsetD*, *OF le*(*1*)]
elim!: *eq-$\mathcal{I}$-gpv-mono'*[*OF - - le*(*2*)] *outs-$\mathcal{I}$-mono*[*THEN subsetD*, *OF le*(*1*)])

lemma *eq-$\mathcal{I}$-converter-eq*: *conv1* = *conv2* **if** *$\mathcal{I}$-full*, *$\mathcal{I}$-full* $\vdash_C$ *conv1* ∼ *conv2*
using *that*
by(*coinduction arbitrary*: *conv1 conv2*)
(*auto simp add*: *eq-$\mathcal{I}$-gpv-into-rel-gpv eq-onp-def intro*!: *rel-funI elim*!: *gpv.rel-mono-strong eq-$\mathcal{I}$-gpv-into-rel-gpv dest*:*eq-$\mathcal{I}$-converterD*)

lemma *eq-$\mathcal{I}$-attach-on*:
assumes $\mathcal{I}' \vdash res$ *res* $\surd$ *$\mathcal{I}$-uniform A UNIV*, $\mathcal{I}' \vdash_C conv \sim conv'$
shows $A \vdash_R$ *attach conv res* ∼ *attach conv' res*
using *assms*
by(*coinduction arbitrary*: *conv conv' res*)
(*auto 4 4 simp add*: *eq-onp-def spmf-rel-map dest*: *eq-$\mathcal{I}$-converterD intro*!: *rel-funI run-resource.eq-$\mathcal{I}$-exec-gpv*[*THEN rel-spmf-mono*])

lemma *eq-$\mathcal{I}$-attach-on'*:
assumes $\mathcal{I}' \vdash res$ *res* $\surd$ $\mathcal{I}$, $\mathcal{I}' \vdash_C conv \sim conv'$ $A \subseteq$ *outs-$\mathcal{I}$* $\mathcal{I}$
shows $A \vdash_R$ *attach conv res* ∼ *attach conv' res*
using *assms*(*1*) *assms*(*2*)[*THEN eq-$\mathcal{I}$-converter-mono*]
by(*rule eq-$\mathcal{I}$-attach-on*)(*use assms*(*3*) **in** ‹*auto simp add*: *le-$\mathcal{I}$-def*›)

lemma *eq-$\mathcal{I}$-attach*:
⟦ $\mathcal{I}' \vdash res$ *res* $\surd$; *$\mathcal{I}$-full*, $\mathcal{I}' \vdash_C conv \sim conv'$ ⟧ ⟹ *attach conv res* = *attach conv' res*
by(*rule eq-resource-on-UNIV-D*)(*simp add*: *eq-$\mathcal{I}$-attach-on*)

lemma *eq-$\mathcal{I}$-comp-cong*:
⟦ $\mathcal{I}1, \mathcal{I}2 \vdash_C conv1 \sim conv1'$; $\mathcal{I}2, \mathcal{I}3 \vdash_C conv2 \sim conv2'$ ⟧
⟹ $\mathcal{I}1, \mathcal{I}3 \vdash_C$ *comp-converter conv1 conv2* ∼ *comp-converter conv1' conv2'*
by(*coinduction arbitrary*: *conv1 conv2 conv1' conv2'*)
(*clarsimp*, *rule eq-$\mathcal{I}$-gpv-mono'*[*OF eq-$\mathcal{I}$-gpv-inline*[**where** *S*=*eq-$\mathcal{I}$-converter* $\mathcal{I}2$ $\mathcal{I}3$]]

, (*fastforce elim*!: *eq-I-converterD*)+)

lemma *comp-converter-cong*: *comp-converter conv1 conv2* = *comp-converter conv1' conv2'*
if *I-full*, $\mathcal{I} \vdash_C$ *conv1* $\sim$ *conv1'* $\mathcal{I}$, *I-full* $\vdash_C$ *conv2* $\sim$ *conv2'*
by(*rule eq-I-converter-eq*)(*rule eq-I-comp-cong*[*OF that*])

lemma *parallel-converter2-id-id*:
$\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_C$ *parallel-converter2 id-converter id-converter* $\sim$ *id-converter*
by(*coinduction*)(*auto split*: *sum.split intro*!: *eq-I-gpv-Pause simp add*: *eq-onp-same-args*)

lemma *parallel-converter2-eq-I-cong*:
$[\![\mathcal{I}1, \mathcal{I}1' \vdash_C$ *conv1* $\sim$ *conv1'*; $\mathcal{I}2, \mathcal{I}2' \vdash_C$ *conv2* $\sim$ *conv2'* $]\!]$
$\Longrightarrow \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C$ *parallel-converter2 conv1 conv2* $\sim$ *parallel-converter2 conv1' conv2'*
by(*coinduction arbitrary*: *conv1 conv2 conv1' conv2'*)
(*fastforce intro*!: *eq-I-gpv-left-gpv-cong eq-I-gpv-right-gpv-cong dest*: *eq-I-converterD elim*!: *eq-I-gpv-mono' simp add*: *eq-onp-def*)

lemma *id-converter-eq-self*: $\mathcal{I},\mathcal{I}' \vdash_C$ *id-converter* $\sim$ *id-converter* **if** $\mathcal{I} \leq \mathcal{I}'$
by(*rule eq-I-converter-mono*[*OF - order-refl that*])(*rule eq-I-converter-reflI*[*OF WT-converter-id*])

lemma *eq-I-converterD-WT1*:
assumes $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\sim$ *conv2* **and** $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\surd$
shows $\mathcal{I},\mathcal{I}' \vdash_C$ *conv2* $\surd$
using *assms*
by(*coinduction arbitrary*: *conv1 conv2*)
(*drule* (1) *eq-I-converterD*, *auto* 4 3 *dest*: *eq-I-converterD eq-I-gpvD-WT1 WT-converterD-WT WT-converterD-results eq-I-gpvD-results-gpv2 simp add*: *eq-onp-def*)

lemma *eq-I-converterD-WT*:
assumes $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\sim$ *conv2*
shows $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\surd \longleftrightarrow \mathcal{I},\mathcal{I}' \vdash_C$ *conv2* $\surd$
proof(*rule iffI*, *goal-cases*)
case 1 **then show** *?case* **using** *assms* **by** (*auto intro*: *eq-I-converterD-WT1*)
next
case 2 **then show** *?case* **using** *assms*[*symmetric*] **by** (*auto intro*: *eq-I-converterD-WT1*)
qed

lemma *eq-I-gpv-Fail* [*simp*]: *eq-I-gpv A I Fail Fail*
by(*rule eq-I-gpv.intros*) *simp*

lemma *eq-I-restrict-gpv*:
assumes *eq-I-gpv A I gpv gpv'*
shows *eq-I-gpv A I* (*restrict-gpv I gpv*) *gpv'*
using *assms*
by(*coinduction arbitrary*: *gpv gpv'*)
(*fastforce dest*: *eq-I-gpvD simp add*: *spmf-rel-map pmf.rel-map option-rel-Some1*

eq-$\mathcal{I}$-generat-iff1 elim!: *pmf.rel-mono-strong eq-$\mathcal{I}$-generat-mono′ split*: *option.split generat.split*)

lemma *eq-$\mathcal{I}$-restrict-converter*:
assumes $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *cnv* $\surd$
and *outs-$\mathcal{I}$* $\mathcal{I} \subseteq A$
shows $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *restrict-converter A* $\mathcal{I}'$ *cnv* $\sim$ *cnv*
using *assms*(*1*)
by(*coinduction arbitrary*: *cnv*)
(*use assms*(*2*) **in** ‹*auto intro*!: *eq-$\mathcal{I}$-gpv-reflI eq-$\mathcal{I}$-restrict-gpv simp add*: *eq-onp-def dest*: *WT-converterD*›)

lemma *eq-$\mathcal{I}$-restrict-gpv-full*:
eq-$\mathcal{I}$-gpv A $\mathcal{I}$-full (*restrict-gpv* $\mathcal{I}$ *gpv*) (*restrict-gpv* $\mathcal{I}$ *gpv′*)
if *eq-$\mathcal{I}$-gpv A* $\mathcal{I}$ *gpv gpv′*
using *that*
by(*coinduction arbitrary*: *gpv gpv′*)
(*fastforce dest*: *eq-$\mathcal{I}$-gpvD simp add*: *pmf.rel-map in-set-spmf*[*symmetric*] *elim*!: *pmf.rel-mono-strong split*!: *option.split generat.split*)

lemma *eq-$\mathcal{I}$-restrict-converter-cong*:
assumes $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *cnv* $\sim$ *cnv′*
and $A \subseteq$ *outs-$\mathcal{I}$* $\mathcal{I}$
shows *restrict-converter A* $\mathcal{I}'$ *cnv* = *restrict-converter A* $\mathcal{I}'$ *cnv′*
using *assms*
by(*coinduction arbitrary*: *cnv cnv′*)
(*fastforce intro*: *eq-$\mathcal{I}$-gpv-into-rel-gpv eq-$\mathcal{I}$-restrict-gpv-full elim*!: *eq-$\mathcal{I}$-gpv-mono′ simp add*: *eq-onp-def rel-fun-def gpv.rel-map dest*: *eq-$\mathcal{I}$-converterD*)

end

4 Trace equivalence for resources

theory *Random-System* **imports** *Converter-Rewrite* **begin**

fun *trace-callee* :: (*'a*, *'b*, *'s*) *callee* ⇒ *'s spmf* ⇒ (*'a* × *'b*) *list* ⇒ *'a* ⇒ *'b spmf* **where**
trace-callee callee p [] *x* = *bind-spmf p* (λ*s*. *map-spmf fst* (*callee s x*))
| *trace-callee callee p* ((*a*, *b*) # *xs*) *x* =
trace-callee callee (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee s a*)) *b*) *xs x*

definition *trace-callee-eq* :: (*'a*, *'b*, *'s1*) *callee* ⇒ (*'a*, *'b*, *'s2*) *callee* ⇒ *'a set* ⇒ *'s1 spmf* ⇒ *'s2 spmf* ⇒ *bool* **where**
trace-callee-eq callee1 callee2 A p q ⟷
(∀ *xs*. *set xs* ⊆ *A* × *UNIV* ⟶ (∀ *x* ∈ *A*. *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*))

abbreviation *trace-callee-eq′* :: *'a set* ⇒ (*'a*, *'b*, *'s1*) *callee* ⇒ *'s1* ⇒ (*'a*, *'b*, *'s2*) *callee* ⇒ *'s2* ⇒ *bool*

(- $\vdash_C$/ (-'((-)')) $\approx$/ (-'((-)')) [90, 0, 0, 0, 0] 91)
where *trace-callee-eq' A callee1 s1 callee2 s2* $\equiv$ *trace-callee-eq callee1 callee2 A* (*return-spmf s1*) (*return-spmf s2*)

lemma *trace-callee-eqI*:
assumes $\bigwedge$*xs x*. ⟦ *set xs* $\subseteq$ *A* $\times$ *UNIV*; *x* $\in$ *A* ⟧
$\Longrightarrow$ *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*
shows *trace-callee-eq callee1 callee2 A p q*
using *assms* **by**(*simp add*: *trace-callee-eq-def*)

lemma *trace-callee-eqD*:
assumes *trace-callee-eq callee1 callee2 A p q*
and *set xs* $\subseteq$ *A* $\times$ *UNIV x* $\in$ *A*
shows *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*
using *assms* **by**(*simp add*: *trace-callee-eq-def*)

lemma *cond-spmf-fst-None* [*simp*]: *cond-spmf-fst* (*return-pmf None*) *x* = *return-pmf None*
by(*simp*)

lemma *trace-callee-None* [*simp*]:
trace-callee callee (*return-pmf None*) *xs x* = *return-pmf None*
by(*induction xs*)(*auto*)

proposition *trace'-eqI-sim*:
fixes *callee1* :: ('*a*, '*b*, '*s1*) *callee* **and** *callee2* :: ('*a*, '*b*, '*s2*) *callee*
assumes *start*: *S p q*
and *step*: $\bigwedge$*p q a*. ⟦ *S p q*; *a* $\in$ *A* ⟧ $\Longrightarrow$
bind-spmf p (λ*s*. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ*s*. *map-spmf fst* (*callee2 s a*))
and *sim*: $\bigwedge$*p q a res b s'*. ⟦ *S p q*; *a* $\in$ *A*; *res* $\in$ *set-spmf q*; (*b*, *s'*) $\in$ *set-spmf* (*callee2 res a*) ⟧
$\Longrightarrow$ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s a*)) *b*)
shows *trace-callee-eq callee1 callee2 A p q*
proof(*rule trace-callee-eqI*)
fix *xs* :: ('*a* $\times$ '*b*) *list* **and** *z*
assume *xs*: *set xs* $\subseteq$ *A* $\times$ *UNIV* **and** *z*: *z* $\in$ *A*

from *start* **show** *trace-callee callee1 p xs z* = *trace-callee callee2 q xs z* **using** *xs*
proof(*induction xs arbitrary*: *p q*)
case *Nil*
then show *?case* **using** *z* **by**(*simp add*: *step*)
next
case (*Cons xy xs*)
obtain *x y* **where** *xy* [*simp*]: *xy* = (*x*, *y*) **by**(*cases xy*)
have *trace-callee callee1 p* (*xy* # *xs*) *z* =
trace-callee callee1 (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee1 s x*)) *y*) *xs z*
by(*simp add*: *bind-map-spmf split-def o-def*)

also have ... = *trace-callee callee2* (*cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s x*)) *y*) *xs z*
proof(*cases* ∃ *s* ∈ *set-spmf q*. ∃ *s'*. (*y*, *s'*) ∈ *set-spmf* (*callee2 s x*))
case *True*
then obtain *s s'* **where** *s* ∈ *set-spmf q* (*y*, *s'*) ∈ *set-spmf* (*callee2 s x*) **by** *blast*
from *sim*[*OF* ‹*S p q*› - *this*] **show** *?thesis* **using** *Cons.prems* **by** (*auto intro*: *Cons.IH*)
next
case *False*
then have *cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s x*)) *y* = *return-pmf None*
by(*auto simp add*: *bind-eq-return-pmf-None*)
moreover from *step*[*OF* ‹*S p q*›, *of x*, *THEN arg-cong*[**where** *f*=*set-spmf*], *THEN eq-refl*] *Cons.prems False*
have *cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee1 s x*)) *y* = *return-pmf None*
by(*clarsimp simp add*: *bind-eq-return-pmf-None*)(*rule ccontr*; *fastforce*)
ultimately show *?thesis* **by**(*simp del*: *cond-spmf-fst-eq-return-None*)
qed
also have ... = *trace-callee callee2 q* (*xy* # *xs*) *z*
by(*simp add*: *split-def o-def*)
finally show *?case* **.**
qed
qed

fun *trace-callee-aux* :: ('*a*, '*b*, '*s*) *callee* ⇒ '*s spmf* ⇒ ('*a* × '*b*) *list* ⇒ '*s spmf* **where**
trace-callee-aux callee p [] = *p*
| *trace-callee-aux callee p* ((*x*, *y*) # *xs*) = *trace-callee-aux callee* (*cond-spmf-fst* (*bind-spmf p* (λ*res*. *callee res x*)) *y*) *xs*

lemma *trace-callee-conv-trace-callee-aux*:
trace-callee callee p xs a = *bind-spmf* (*trace-callee-aux callee p xs*) (λ*s*. *map-spmf fst* (*callee s a*))
by(*induction xs arbitrary*: *p*)(*auto simp add*: *split-def*)

lemma *trace-callee-aux-append*:
trace-callee-aux callee p (*xs* @ *ys*) = *trace-callee-aux callee* (*trace-callee-aux callee p xs*) *ys*
by(*induction xs arbitrary*: *p*)(*auto simp add*: *split-def*)

inductive *trace-callee-closure* :: ('*a*, '*b*, '*s1*) *callee* ⇒ ('*a*, '*b*, '*s2*) *callee* ⇒ '*a set* ⇒ '*s1 spmf* ⇒ '*s2 spmf* ⇒ '*s1 spmf* ⇒ '*s2 spmf* ⇒ *bool*
for *callee1 callee2 A p q* **where**
trace-callee-closure callee1 callee2 A p q (*trace-callee-aux callee1 p xs*) (*trace-callee-aux callee2 q xs*) **if** *set xs* ⊆ *A* × *UNIV*

lemma *trace-callee-closure-start*: *trace-callee-closure callee1 callee2 A p q p q*
by(*simp add*: *trace-callee-closure.simps exI*[**where** *x*=[]])

lemma *trace-callee-closure-step*:
assumes *trace-callee-eq callee1 callee2 A p q*
and *trace-callee-closure callee1 callee2 A p q p′ q′*
and $a \in A$
shows *bind-spmf p′* (λ*s*. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q′* (λ*s*. *map-spmf fst* (*callee2 s a*))
proof −
from *assms*(*2*) **obtain** *xs* **where** *xs*: *set xs* ⊆ $A \times$ *UNIV*
and *p*: $p' =$ *trace-callee-aux callee1 p xs* **and** *q*: $q' =$ *trace-callee-aux callee2 q xs* **by**(*cases*)
from *trace-callee-eqD*[*OF assms*(*1*) *xs assms*(*3*)] *p q* **show** *?thesis*
by(*simp add*: *trace-callee-conv-trace-callee-aux*)
qed

lemma *trace-callee-closure-sim*:
assumes *trace-callee-closure callee1 callee2 A p q p′ q′*
and $a \in A$
shows *trace-callee-closure callee1 callee2 A p q*
(*cond-spmf-fst* (*bind-spmf p′* (λ*s*. *callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q′* (λ*s*. *callee2 s a*)) *b*)
using *assms* **proof** (*cases*)
case (*1 xs*)
then show *?thesis* **by**
(*auto simp add*:*trace-callee-closure.simps assms trace-callee-aux-append split-def map-spmf-conv-bind-spmf intro*!: *exI*[**where** *x*=*xs* @ [(*a*, *b*)]])
qed

proposition *trace-callee-eq-complete*:
assumes *trace-callee-eq callee1 callee2 A p q*
obtains *S*
where *S p q*
and $\bigwedge p\ q\ a.\ [\![\ S\ p\ q;\ a \in A\ ]\!] \Longrightarrow$
bind-spmf p (λ*s*. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ*s*. *map-spmf fst* (*callee2 s a*))
and $\bigwedge p\ q\ a\ s\ b\ s'.\ [\![\ S\ p\ q;\ a \in A;\ s \in$ *set-spmf q*; $(b, s') \in$ *set-spmf* (*callee2 s a*)]\!]
$\Longrightarrow$ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s a*)) *b*)
by(*rule that*[**where** *S*=*trace-callee-closure callee1 callee2 A p q*])
(*auto intro*: *trace-callee-closure-start trace-callee-closure-step*[*OF assms*] *trace-callee-closure-sim*)

lemma *set-spmf-cond-spmf-fst*: *set-spmf* (*cond-spmf-fst p a*) = *snd* ‘ (*set-spmf p* ∩ {*a*} × *UNIV*)
by(*simp add*: *cond-spmf-fst-def*)

lemma *trace-callee-eq-run-gpv*:
fixes *callee1* :: (′*a*, ′*b*, ′*s1*) *callee* **and** *callee2* :: (′*a*, ′*b*, ′*s2*) *callee*
assumes *trace-eq*: *trace-callee-eq callee1 callee2 A p q*
and *inv1*: *callee-invariant-on callee1 I1* $\mathcal{I}$

and *inv1*: *callee-invariant-on callee2 I2 $\mathcal{I}$*
and *WT*: *$\mathcal{I}$ ⊢g gpv √*
and *out*: *outs-gpv $\mathcal{I}$ gpv ⊆ A*
and *pq*: *lossless-spmf p lossless-spmf q*
and *I1*: *∀x∈set-spmf p. I1 x*
and *I2*: *∀y∈set-spmf q. I2 y*
shows *bind-spmf p (run-gpv callee1 gpv) = bind-spmf q (run-gpv callee2 gpv)*
proof −
from *trace-eq* **obtain** *S* **where** *start*: *S p q*
and *sim*: *⋀p q a. ⟦ S p q; a ∈ A ⟧ ⟹ bind-spmf p (λs. map-spmf fst (callee1 s a)) = bind-spmf q (λs. map-spmf fst (callee2 s a))*
and *S*: *⋀p q a s b s′. ⟦ S p q; a ∈ A; s ∈ set-spmf q; (b, s′) ∈ set-spmf (callee2 s a) ⟧ ⟹ S (cond-spmf-fst (bind-spmf p (λs. callee1 s a)) b) (cond-spmf-fst (bind-spmf q (λs. callee2 s a)) b)*
by(*rule trace-callee-eq-complete*) *blast*

interpret *I1*: *callee-invariant-on callee1 I1 $\mathcal{I}$* **by** *fact*
interpret *I2*: *callee-invariant-on callee2 I2 $\mathcal{I}$* **by** *fact*

from ‹*S p q*› *out pq WT I1 I2* **show** *?thesis*
proof(*induction arbitrary*: *p q gpv rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf exec-gpv.mono exec-gpv.mono exec-gpv-def exec-gpv-def*, *case-names adm bottom step*])
case (*step exec-gpv′ g*)
have[*simp*]: *generat ∈ set-spmf (the-gpv gpv) ⟹ p ⤜ (λxa. map-spmf fst (case generat of Pure x ⇒ return-spmf (x, xa) | IO out c ⇒ callee1 xa out ⤜ (λ(x, y). exec-gpv′ (c x) y))) = q ⤜ (λxa. map-spmf fst (case generat of Pure x ⇒ return-spmf (x, xa) | IO out c ⇒ callee2 xa out ⤜ (λ(x, y). g (c x) y))) **for** generat*
proof (*cases generat*, *goal-cases*)
case (*2 out rpv*)
have [*simp*]: *IO out rpv ∈ set-spmf (the-gpv gpv) ⟹ generat = IO out rpv ⟹ map-spmf fst (p ⤜ (λxa. callee1 xa out)) = map-spmf fst (q ⤜ (λxa. callee2 xa out))*
unfolding *map-bind-spmf o-def*
by (*rule sim*) (*use step.prems* **in** ‹*auto intro*: *outs-gpv.IO*›)

have[*simp*]: *⟦IO out rpv ∈ set-spmf (the-gpv gpv); generat = IO out rpv; x ∈ set-spmf q; (a, b) ∈ set-spmf (callee2 x out)⟧ ⟹ cond-spmf-fst (p ⤜ (λxa. callee1 xa out)) a ⤜ (λx. map-spmf fst (exec-gpv′ (rpv a) x)) = cond-spmf-fst (q ⤜ (λxa. callee2 xa out)) a ⤜ (λx. map-spmf fst (g (rpv a) x)) **for** a b x*
proof (*rule step.IH*, *goal-cases*)

```
        case 1 then show ?case using step.prems by(auto intro!: S intro: outs-gpv.IO)
    next
      case 2
       then show ?case using step.prems by(force intro: outs-gpv.Cont dest: WT-calleeD[OF I2.WT-callee] WT-gpvD[OF step.prems(5)])
    next
      case 3
      then show ?case using sim[OF ‹S p q›, of out] step.prems(2)
        by(force simp add: bind-UNION image-Union intro: rev-image-eqI intro: outs-gpv.IO dest: arg-cong[where f=set-spmf])
    next
      case 4
       then show ?case by(auto 4 3 simp add: bind-UNION image-Union intro: rev-image-eqI)
    next
      case 5
        then show ?case using step.prems by(auto 4 3 dest: WT-calleeD[OF I2.WT-callee] intro: WT-gpvD)
    next
      case 6
         then show ?case using step.prems by(auto simp add: bind-UNION image-Union set-spmf-cond-spmf-fst intro: I1.callee-invariant WT-gpvD)
    next
      case 7
         then show ?case using step.prems by(auto simp add: bind-UNION image-Union set-spmf-cond-spmf-fst intro: I2.callee-invariant WT-gpvD)
    qed

    from 2 show ?case
      by(simp add: map-bind-spmf o-def)
        (subst (1 2) bind-spmf-assoc[symmetric]
          , rewrite in bind-spmf ⌑ - = - cond-spmf-fst-inverse[symmetric]
          , rewrite in - = bind-spmf ⌑ - cond-spmf-fst-inverse[symmetric]
          , subst (1 2) bind-spmf-assoc
          , auto simp add: bind-map-spmf o-def intro!: bind-spmf-cong)
  qed (simp add: bind-spmf-const lossless-weight-spmfD step.prems)

  show ?case unfolding map-bind-spmf o-def by(subst (1 2) bind-commute-spmf)(auto intro: bind-spmf-cong[OF refl])
 qed simp-all
qed

lemma trace-callee-eq'-run-gpv:
 fixes callee1 :: ('a, 'b, 's1) callee and callee2 :: ('a, 'b, 's2) callee
 assumes trace-eq: A ⊢C callee1(s1) ≈ callee2(s2)
   and inv1: callee-invariant-on callee1 I1 ℐ
   and inv1: callee-invariant-on callee2 I2 ℐ
   and WT: ℐ ⊢g gpv √
```

and *out*: *outs-gpv* $\mathcal{I}$ *gpv* $\subseteq$ *A*
and *I1*: *I1 s1*
and *I2*: *I2 s2*
shows *run-gpv callee1 gpv s1* = *run-gpv callee2 gpv s2*
using *trace-callee-eq-run-gpv*[*OF assms*(1−5)] *assms*(6−) **by** *simp*

abbreviation *trace-eq* :: *'a set* ⇒ (*'a*, *'b*) *resource spmf* ⇒ (*'a*, *'b*) *resource spmf* ⇒ *bool* **where**
trace-eq ≡ *trace-callee-eq run-resource run-resource*

abbreviation *trace-eq'* :: *'a set* ⇒ (*'a*, *'b*) *resource* ⇒ (*'a*, *'b*) *resource* ⇒ *bool* ((-) ⊢$_R$/ (-)/ ≈ (-) [*90*, *90*, *90*] *91*) **where**
A ⊢$_R$ *res* ≈ *res'* ≡ *trace-eq A* (*return-spmf res*) (*return-spmf res'*)

lemma *trace-callee-resource-of-oracle2*:
trace-callee run-resource (*map-spmf* (*resource-of-oracle callee*) *p*) *xs x* =
trace-callee callee p xs x
proof(*induction xs arbitrary*: *p*)
case (*Cons xy xs*)
then show *?case* **by** (*cases xy*) (*simp add*: *bind-map-spmf o-def Cons.IH*[*symmetric*] *cond-spmf-fst-def map-bind-spmf*[*symmetric*, *unfolded o-def*] *spmf.map-comp map-prod-vimage*)
qed (*simp add*: *map-bind-spmf bind-map-spmf o-def spmf.map-comp*)

lemma *trace-callee-resource-of-oracle* [*simp*]:
trace-callee run-resource (*return-spmf* (*resource-of-oracle callee s*)) *xs x* =
trace-callee callee (*return-spmf s*) *xs x*
using *trace-callee-resource-of-oracle2*[*of callee return-spmf s xs x*] **by** *simp*

lemma *trace-eq'-resource-of-oracle* [*simp*]:
A ⊢$_R$ *resource-of-oracle callee1 s1* ≈ *resource-of-oracle callee2 s2* =
A ⊢$_C$ *callee1*(*s1*) ≈ *callee2*(*s2*)
by(*simp add*: *trace-callee-eq-def*)

end

5 Distinguisher

theory *Distinguisher* **imports** *Random-System* **begin**

type-synonym (*'a*, *'b*) *distinguisher* = (*bool*, *'a*, *'b*) *gpv*

translations
(*type*) (*'a*, *'b*) *distinguisher* <= (*type*) (*bool*, *'a*, *'b*) *gpv*

definition *connect* :: (*'a*, *'b*) *distinguisher* ⇒ (*'a*, *'b*) *resource* ⇒ *bool spmf* **where**
connect d res = *run-gpv run-resource d res*

definition *absorb* :: (*'a*, *'b*) *distinguisher* ⇒ (*'a*, *'b*, *'out*, *'in*) *converter* ⇒ (*'out*, *'in*) *distinguisher* **where**

absorb d conv = map-gpv fst id (inline run-converter d conv)

lemma *distinguish-attach*: *connect d (attach conv res) = connect (absorb d conv) res*
proof −
let *?R = λres (conv′, res′). res = attach conv′ res′*
have∗: *rel-spmf (rel-prod (=) ?R) (exec-gpv run-resource d (attach conv res))*
(exec-gpv (λp y. map-spmf (λp. (fst (fst p), snd (fst p), snd p))
(exec-gpv run-resource (run-converter (fst p) y) (snd p))) d (conv, res))
by(*rule exec-gpv-parametric*[**where** *S=λres (conv′, res′). res = attach conv′ res′* **and** *CALL=(=), THEN rel-funD, THEN rel-funD, THEN rel-funD*])
(*auto simp add*: *gpv.rel-eq spmf-rel-map split-def intro*: *rel-spmf-reflI intro*!: *rel-funI*)

show *?thesis*
by(*simp add*: *connect-def absorb-def exec-gpv-map-gpv-id spmf.map-comp exec-gpv-inline o-def split-def spmf-rel-eq*[*symmetric*])
(*rule pmf.map-transfer*[*THEN rel-funD, THEN rel-funD*]
, *rule option.map-transfer*[**where** *Rb=rel-prod (=) ?R, THEN rel-funD*]
, *auto simp add*: ∗ *intro*: *fst-transfer*)
qed

lemma *absorb-comp-converter*: *absorb d (comp-converter conv conv′) = absorb (absorb d conv) conv′*
proof −
let *?R = λconv (conv′, conv″). conv = comp-converter conv′ conv″*
have∗: *rel-gpv (rel-prod (=) ?R) (=) (inline run-converter d (comp-converter conv conv′))*
(inline (λp c2. map-gpv (λp. (fst (fst p), snd (fst p), snd p)) id (inline run-converter (run-converter (fst p) c2) (snd p))) d (conv, conv′))
by(*rule inline-parametric*[**where** *C=(=), THEN rel-funD, THEN rel-funD, THEN rel-funD*])
(*auto simp add*: *gpv.rel-eq gpv.rel-map split-def intro*: *gpv.rel-refl-strong intro*!: *rel-funI*)

show *?thesis*
by(*simp add*: *gpv.rel-eq*[*symmetric*] *absorb-def inline-map-gpv gpv.map-comp inline-assoc o-def split-def id-def*[*symmetric*])
(*rule gpv.map-transfer*[**where** *R1b=rel-prod (=) ?R, THEN rel-funD, THEN rel-funD, THEN rel-funD*]
, *auto simp add*: ∗ *intro*: *fst-transfer id-transfer*)
qed

lemma *connect-cong-trace*:
fixes *res1 res2* :: (′*a*, ′*b*) *resource*
assumes *trace-eq*: *A* ⊢$_R$ *res1* ≈ *res2*
and *WT*: *I* ⊢*g d* √
and *out*: *outs-gpv I d* ⊆ *A*
and *WT1*: *I* ⊢*res res1* √

and *WT2*: $\mathcal{I} \vdash res\ res2\ \surd$
shows *connect d res1 = connect d res2*
unfolding *connect-def* **using** *trace-eq callee-invariant-run-resource callee-invariant-run-resource WT out WT1 WT2*
by(*rule trace-callee-eq'-run-gpv*)

lemma *distinguish-trace-eq*:
assumes *distinguish*: $\bigwedge$*distinguisher*. $\mathcal{I} \vdash g$ *distinguisher* $\surd \Longrightarrow$ *connect distinguisher res = connect distinguisher res'*
and *WT1*: $\mathcal{I} \vdash res\ res1\ \surd$
and *WT2*: $\mathcal{I} \vdash res\ res2\ \surd$
shows *outs-*$\mathcal{I}$ $\mathcal{I} \vdash_R$ *res* $\approx$ *res'*
proof(*rule ccontr*)
let *?P* = λ*xs*. $\exists$ *x*. *set xs* $\subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\times$ *UNIV* $\wedge$ *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\wedge$ *trace-callee run-resource* (*return-spmf res*) *xs x* $\neq$ *trace-callee run-resource* (*return-spmf res'*) *xs x*
assume $\neg$ *?thesis*
then have *Ex ?P* **unfolding** *trace-callee-eq-def* **by** *auto*
with *wf-strict-prefix*[*unfolded wfP-eq-minimal*, *THEN spec*, *of Collect ?P*]
obtain *xs x* **where** *xs*: *set xs* $\subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\times$ *UNIV*
and *x*: *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$
and *neq*: *trace-callee run-resource* (*return-spmf res*) *xs x* $\neq$ *trace-callee run-resource* (*return-spmf res'*) *xs x*
and *shortest*: $\bigwedge$*xs' x*. $[\![$ *strict-prefix xs' xs*; *set xs'* $\subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\times$ *UNIV*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!]$
$\Longrightarrow$ *trace-callee run-resource* (*return-spmf res*) *xs' x* = *trace-callee run-resource* (*return-spmf res'*) *xs' x*
by(*auto*)
have *shortest*: $\bigwedge$*xs' x*. $[\![$ *strict-prefix xs' xs*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!]$
$\Longrightarrow$ *trace-callee run-resource* (*return-spmf res*) *xs' x* = *trace-callee run-resource* (*return-spmf res'*) *xs' x*
by(*rule shortest*)(*use xs* **in** ‹*auto 4 3 dest*: *strict-prefix-setD*›)

define *p* **where** *p* $\equiv$ *return-spmf res*
define *q* **where** *q* $\equiv$ *return-spmf res'*
have *p*: *lossless-spmf p* **and** *q*: *lossless-spmf q* **by**(*simp-all add*: *p-def q-def*)
from *neq* **obtain** *y* **where** *y*: *spmf* (*trace-callee run-resource p xs x*) *y* $\neq$ *spmf* (*trace-callee run-resource q xs x*) *y*
by(*fastforce intro*: *spmf-eqI simp add*: *p-def q-def*)
have *eq*: *spmf* (*trace-callee run-resource p ys x*) *y* = *spmf* (*trace-callee run-resource q ys x*) *y*
if *strict-prefix ys xs x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ **for** *ys x y* **using** *shortest*[*OF that*]
by(*auto intro*: *spmf-eqI simp add*: *p-def q-def*)
define *d* :: (*'a* $\times$ *'b*) *list* $\Rightarrow$ -
where *d* = *rec-list* (*Pause x* (λ*y'*. *Done* (*y* = *y'*))) (λ(*x*, *y*) *xs rec*. *Pause x* (λ*input*. *if input* = *y then rec else Done False*))
have *d-simps* [*simp*]:
d [] = *Pause x* (λ*y'*. *Done* (*y* = *y'*))
d ((*a*, *b*) # *xs*) = *Pause a* (λ*input*. *if input* = *b then d xs else Done False*) **for**

a b xs
by(*simp-all add*: *d-def fun-eq-iff*)
have *WT-d*: $\mathcal{I} \vdash_g d\ xs\ \surd$ **using** *xs* **by**(*induction xs*)(*use x* **in** *auto*)
from *distinguish*[*OF WT-d*]
have *spmf* (*bind-spmf p* (*connect* (*d xs*))) *True* = *spmf* (*bind-spmf q* (*connect* (*d xs*))) *True*
by(*simp add*: *p-def q-def*)
thus *False* **using** *y eq xs*
proof(*induction xs arbitrary*: *p q*)
case *Nil*
then show *?case*
by(*simp add*: *connect-def*[*abs-def*] *map-bind-spmf o-def split-def*)
(*simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *map-bind-spmf*[*unfolded o-def*, *symmetric*] *spmf-map vimage-def eq-commute*)
next
case (*Cons xy xs p q*)
obtain *x′ y′* **where** *xy* [*simp*]: *xy* = (*x′*, *y′*) **by**(*cases xy*)
let *?p* = *cond-spmf-fst* (*p* ⤜ (λ*s*. *run-resource s x′*)) *y′*
and *?q* = *cond-spmf-fst* (*q* ⤜ (λ*s*. *run-resource s x′*)) *y′*
from *Cons.prems*(*1*)
have *spmf* ((*map-spmf fst* (*p* ⤜ (λ*y*. *run-resource y x′*)) ⤜ (λ*x*. *map-spmf* (*Pair x*) (*cond-spmf-fst* (*p* ⤜ (λ*y*. *run-resource y x′*)) *x*))) ⤜ (λ(*a*, *b*). *if a* = *y′ then connect* (*d xs*) *b else return-spmf False*)) *True* =
spmf ((*map-spmf fst* (*q* ⤜ (λ*y*. *run-resource y x′*)) ⤜ (λ*x*. *map-spmf* (*Pair x*) (*cond-spmf-fst* (*q* ⤜ (λ*y*. *run-resource y x′*)) *x*))) ⤜ (λ(*a*, *b*). *if a* = *y′ then connect* (*d xs*) *b else return-spmf False*)) *True*
unfolding *cond-spmf-fst-inverse*
by(*clarsimp simp add*: *connect-def*[*abs-def*] *map-bind-spmf o-def split-def if-distrib*[**where** *f*=λ*x*. *run-gpv - x -*] *cong del*: *if-weak-cong*)
hence *spmf* ((*p* ⤜ (λ*s*. *map-spmf fst* (*run-resource s x′*))) ⤜
(λ*a*. *if a* = *y′ then cond-spmf-fst* (*p* ⤜ (λ*y*. *run-resource y x′*)) *a* ⤜ *connect* (*d xs*)
else bind-spmf (*cond-spmf-fst* (*p* ⤜ (λ*y*. *run-resource y x′*)) *a*) (λ-. *return-spmf False*))) *True* =
spmf ((*q* ⤜ (λ*s*. *map-spmf fst* (*run-resource s x′*))) ⤜
(λ*a*. *if a* = *y′ then cond-spmf-fst* (*q* ⤜ (λ*y*. *run-resource y x′*)) *a* ⤜ *connect* (*d xs*)
else bind-spmf (*cond-spmf-fst* (*q* ⤜ (λ*y*. *run-resource y x′*)) *a*) (λ-. *return-spmf False*))) *True*
by(*rule box-equals*; *use nothing* **in** ‹*rule arg-cong2*[*where f*=*spmf*]›)
(*auto simp add*: *map-bind-spmf bind-map-spmf o-def split-def intro*!: *bind-spmf-cong*)
hence *LINT a*|*measure-spmf* (*p* ⤜ (λ*s*. *map-spmf fst* (*run-resource s x′*))). (*if a* = *y′ then spmf* (*?p* ⤜ *connect* (*d xs*)) *True else 0*) =
LINT a|*measure-spmf* (*q* ⤜ (λ*s*. *map-spmf fst* (*run-resource s x′*))). (*if a* = *y′ then spmf* (*?q* ⤜ *connect* (*d xs*)) *True else 0*)
by(*rule box-equals*; *use nothing* **in** ‹*subst spmf-bind*›)
(*auto intro*!: *Bochner-Integration.integral-cong simp add*: *bind-spmf-const spmf-scale-spmf*)
hence *LINT a*|*measure-spmf* (*p* ⤜ (λ*s*. *map-spmf fst* (*run-resource s x′*))).

indicator $\{y'\}$ *a* $*$ *spmf* (*?p* $\gg=$ *connect* (*d xs*)) *True* =
LINT a|*measure-spmf* (*q* $\gg=$ (*λs. map-spmf fst* (*run-resource s x'*))). *indicator* $\{y'\}$ *a* $*$ *spmf* (*?q* $\gg=$ *connect* (*d xs*)) *True*
by(*rule box-equals*; *use nothing* **in** ‹*rule Bochner-Integration.integral-cong*›) *auto*
hence *spmf* (*p* $\gg=$ (*λs. map-spmf fst* (*run-resource s x'*))) *y'* $*$ *spmf* (*?p* $\gg=$ *connect* (*d xs*)) *True* =
spmf (*q* $\gg=$ (*λs. map-spmf fst* (*run-resource s x'*))) *y'* $*$ *spmf* (*?q* $\gg=$ *connect* (*d xs*)) *True*
by(*simp add*: *spmf-conv-measure-spmf*)
moreover from *Cons.prems*(*3*)[*of* [] *x'*] *Cons.prems*(*4*)
have *spmf* (*p* $\gg=$ (*λs. map-spmf fst* (*run-resource s x'*))) *y'* = *spmf* (*q* $\gg=$ (*λs. map-spmf fst* (*run-resource s x'*))) *y'*
by(*simp*)
ultimately have *spmf* (*?p* $\gg=$ *connect* (*d xs*)) *True* = *spmf* (*?q* $\gg=$ *connect* (*d xs*)) *True*
by(*auto simp add*: *cond-spmf-fst-def*)(*auto 4 3 simp add*: *spmf-eq-0-set-spmf cond-spmf-def o-def bind-UNION intro*: *rev-image-eqI*)
moreover
have *spmf* (*trace-callee run-resource ?p xs x*) *y* ≠ *spmf* (*trace-callee run-resource ?q xs x*) *y*
using *Cons.prems* **by** *simp*
moreover
have *spmf* (*trace-callee run-resource ?p ys x*) *y* = *spmf* (*trace-callee run-resource ?q ys x*) *y*
if *ys*: *strict-prefix ys xs* **and** *x*: *x* ∈ *outs-I I I* **for** *ys x y*
using *Cons.prems*(*3*)[*of xy # ys x y*] *ys x* **by** *simp*
moreover have *set xs* ⊆ *outs-I I I* × *UNIV* **using** *Cons.prems*(*4*) **by** *auto*
ultimately show *?case* **by**(*rule Cons.IH*)
qed
qed

lemma *connect-eq-resource-cong*:
assumes *I* ⊢*g distinguisher* √
and *outs-I I I* ⊢$_R$ *res* ∼ *res'*
and *I* ⊢*res res* √
shows *connect distinguisher res* = *connect distinguisher res'*
unfolding *connect-def*
by(*fold spmf-rel-eq*, *rule map-spmf-parametric*[*THEN rel-funD*, *THEN rel-funD*, *rotated*])
(*auto simp add*: *rel-fun-def intro*: *assms exec-gpv-eq-resource-on*)

lemma *WT-gpv-absorb* [*WT-intro*]:
⟦ *I'* ⊢*g gpv* √; *I'*, *I* ⊢$_C$ *conv* √ ⟧ ⟹ *I* ⊢*g absorb gpv conv* √
by(*simp add*: *absorb-def run-converter.WT-gpv-inline-invar*)

lemma *plossless-gpv-absorb* [*plossless-intro*]:
assumes *gpv*: *plossless-gpv I' gpv*
and *conv*: *plossless-converter I' I conv*

and [*WT-intro*]: $\mathcal{I}' \vdash g$ *gpv* $\surd$ $\mathcal{I}'$, $\mathcal{I} \vdash_C$ *conv* $\surd$
shows *plossless-gpv* $\mathcal{I}$ (*absorb gpv conv*)
by(*auto simp add*: *absorb-def intro*: *run-plossless-converter.plossless-inline-invariant*[*OF gpv*] *WT-intro conv dest*: *plossless-converterD*)

lemma *interaction-any-bounded-by-absorb* [*interaction-bound*]:
assumes *gpv*: *interaction-any-bounded-by gpv bound1*
and *conv*: *interaction-any-bounded-converter conv bound2*
shows *interaction-any-bounded-by* (*absorb gpv conv*) (*bound1* $*$ *bound2*)
unfolding *absorb-def*
by(*rule interaction-bounded-by-map-gpv-id*, *rule interaction-bounded-by-inline-invariant*[*OF gpv*, *rotated 2*])
(*rule conv*, *auto elim*: *interaction-any-bounded-converter.cases*)

end

6 Wiring

theory *Wiring* **imports**
Distinguisher
begin

6.1 Notation

hide-const (**open**) *Resumption.Pause Monomorphic-Monad.Pause Monomorphic-Monad.Done*

no-notation *Sublist.parallel* (**infixl** $\|$ *50*)
no-notation *plus-oracle* (**infix** $\oplus_O$ *500*)

notation *Resource* (§R§)
notation *Converter* (§C§)

alias *RES* = *resource-of-oracle*
alias *CNV* = *converter-of-callee*

alias *id-intercept* = *id-oracle*
notation *id-oracle* (1_I)

notation *plus-oracle* (**infixr** $\oplus_O$ *504*)
notation *parallel-oracle* (**infixr** $\ddagger_O$ *504*)

notation *plus-intercept* (**infixr** $\oplus_I$ *504*)
notation *parallel-intercept* (**infixr** $\ddagger_I$ *504*)

notation *parallel-resource* (**infixr** $\|$ *501*)

notation *parallel-converter* (**infixr** $|_\propto$ *501*)
notation *parallel-converter2* (**infixr** $|_=$ *501*)
notation *comp-converter* (**infixr** $\odot$ *502*)

notation *fail-converter* ($\bot_C$)
notation *id-converter* (1_C)

notation *attach* (**infixr** $\rhd$ *500*)

6.2 Wiring primitives

primrec *swap-sum* :: $'a + 'b \Rightarrow 'b + 'a$ **where**
  swap-sum (*Inl x*) = *Inr x*
| *swap-sum* (*Inr y*) = *Inl y*

definition $swap_C$ :: ($'a + 'b$, $'c + 'd$, $'b + 'a$, $'d + 'c$) *converter* **where**
  $swap_C$ = *map-converter swap-sum swap-sum id id* 1_C

definition $rassocl_C$ :: ($'a + ('b + 'c)$, $'d + ('e + 'f)$, $('a + 'b) + 'c$, $('d + 'e) + 'f$) *converter* **where**
  $rassocl_C$ = *map-converter lsumr rsuml id id* 1_C
definition $lassocr_C$ :: ($('a + 'b) + 'c$, $('d + 'e) + 'f$, $'a + ('b + 'c)$, $'d + ('e + 'f)$) *converter* **where**
  $lassocr_C$ = *map-converter rsuml lsumr id id* 1_C

definition *swap-rassocl* **where** *swap-rassocl* ≡ $lassocr_C \odot (1_C \mid_= swap_C) \odot rassocl_C$
definition *swap-lassocr* **where** *swap-lassocr* ≡ $rassocl_C \odot (swap_C \mid_= 1_C) \odot lassocr_C$

definition *parallel-wiring* :: (($'a + 'b$) + ($'e + 'f$), ($'c + 'd$) + ($'g + 'h$), ($'a + 'e$) + ($'b + 'f$), ($'c + 'g$) + ($'d + 'h$)) *converter* **where**
  parallel-wiring = $lassocr_C \odot (1_C \mid_= swap\text{-}lassocr) \odot rassocl_C$

lemma $WT\text{-}lassocr_C$ [*WT-intro*]: ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I} \mathcal{I}3$, $\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3) \vdash_C lassocr_C \surd$
  by(*coinduction*)(*auto simp add*: $lassocr_C$*-def*)

lemma $WT\text{-}rassocl_C$ [*WT-intro*]: $\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3)$, ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I} \mathcal{I}3 \vdash_C rassocl_C \surd$
  by(*coinduction*)(*auto simp add*: $rassocl_C$*-def*)

lemma $WT\text{-}swap_C$ [*WT-intro*]: $\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$, $\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}1 \vdash_C swap_C \surd$
  by(*coinduction*)(*auto simp add*: $swap_C$*-def*)

lemma *WT-swap-lassocr* [*WT-intro*]: $\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3)$, $\mathcal{I}2 \oplus_\mathcal{I} (\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3) \vdash_C swap\text{-}lassocr \surd$
  unfolding *swap-lassocr-def*
 by(*rule WT-converter-comp* $WT\text{-}lassocr_C$ $WT\text{-}rassocl_C$ *WT-converter-parallel-converter2 WT-converter-id* $WT\text{-}swap_C$)+

lemma *WT-swap-rassocl* [*WT-intro*]: ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I} \mathcal{I}3$, ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3$) $\oplus_\mathcal{I} \mathcal{I}2 \vdash_C$

swap-rassocl $\surd$
unfolding *swap-rassocl-def*
by(*rule WT-converter-comp WT-lassocr*$_C$ *WT-rassocl*$_C$ *WT-converter-parallel-converter2 WT-converter-id WT-swap*$_C$)+

lemma *WT-parallel-wiring* [*WT-intro*]:
($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I}$ ($\mathcal{I}3 \oplus_\mathcal{I} \mathcal{I}4$), ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3$) $\oplus_\mathcal{I}$ ($\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}4$) $\vdash_C$ *parallel-wiring* $\surd$
unfolding *parallel-wiring-def*
by(*rule WT-converter-comp WT-lassocr*$_C$ *WT-rassocl*$_C$ *WT-converter-parallel-converter2 WT-converter-id WT-swap-lassocr*)+

lemma *map-swap-sum-plus-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *swap-sum* −−−> *map-spmf* (*map-prod swap-sum id*)) (*oracle1* $\oplus_O$ *oracle2*) =
(*oracle2* $\oplus_O$ *oracle1*)
proof ((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases q*) (*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp id-def*)
qed

lemma *map-*$\mathcal{I}$*-rsuml-lsumr* [*simp*]: *map-*$\mathcal{I}$ *rsuml lsumr* ($\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3)$) = (($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I} \mathcal{I}3$)
proof(*rule* $\mathcal{I}$*-eqI*[*OF Set.set-eqI*], *goal-cases*)
case (*1 x*)
then show *?case* **by**(*cases x rule*: *rsuml.cases*) *auto*
qed (*auto simp add*: *image-image*)

lemma *map-*$\mathcal{I}$*-lsumr-rsuml* [*simp*]: *map-*$\mathcal{I}$ *lsumr rsuml* (($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I} \mathcal{I}3$) = ($\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3)$)
proof(*rule* $\mathcal{I}$*-eqI*[*OF Set.set-eqI*], *goal-cases*)
case (*1 x*)
then show *?case* **by**(*cases x rule*: *lsumr.cases*) *auto*
qed (*auto simp add*: *image-image*)

lemma *map-*$\mathcal{I}$*-swap-sum* [*simp*]: *map-*$\mathcal{I}$ *swap-sum swap-sum* ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) = $\mathcal{I}2$ $\oplus_\mathcal{I}$ $\mathcal{I}1$
proof(*rule* $\mathcal{I}$*-eqI*[*OF Set.set-eqI*], *goal-cases*)
case (*1 x*)
then show *?case* **by**(*cases x*) *auto*
qed (*auto simp add*: *image-image*)

definition *parallel-resource1-wiring* :: ('*a* + ('*b* + '*c*), '*d* + ('*e* + '*f*), '*b* + ('*a* + '*c*), '*e* + ('*d* + '*f*)) *converter* **where**
parallel-resource1-wiring = *swap-lassocr*

lemma *WT-parallel-resource1-wiring* [*WT-intro*]: $\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3)$, $\mathcal{I}2 \oplus_\mathcal{I} (\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3) \vdash_C$ *parallel-resource1-wiring* $\surd$
unfolding *parallel-resource1-wiring-def* **by**(*rule WT-swap-lassocr*)

lemma *plossless-rassocl$_C$* [*plossless-intro*]: *plossless-converter* (*$\mathcal{I}1$* $\oplus_\mathcal{I}$ (*$\mathcal{I}2$* $\oplus_\mathcal{I}$ *$\mathcal{I}3$*)) ((*$\mathcal{I}1$* $\oplus_\mathcal{I}$ *$\mathcal{I}2$*) $\oplus_\mathcal{I}$ *$\mathcal{I}3$*) *rassocl$_C$*
by *coinduction* (*auto simp add*: *rassocl$_C$-def*)

lemma *plossless-lassocr$_C$* [*plossless-intro*]: *plossless-converter* ((*$\mathcal{I}1$* $\oplus_\mathcal{I}$ *$\mathcal{I}2$*) $\oplus_\mathcal{I}$ *$\mathcal{I}3$*) (*$\mathcal{I}1$* $\oplus_\mathcal{I}$ (*$\mathcal{I}2$* $\oplus_\mathcal{I}$ *$\mathcal{I}3$*)) *lassocr$_C$*
by *coinduction* (*auto simp add*: *lassocr$_C$-def*)

lemma *plossless-swap$_C$* [*plossless-intro*]: *plossless-converter* (*$\mathcal{I}1$* $\oplus_\mathcal{I}$ *$\mathcal{I}2$*) (*$\mathcal{I}2$* $\oplus_\mathcal{I}$ *$\mathcal{I}1$*) *swap$_C$*
by *coinduction* (*auto simp add*: *swap$_C$-def*)

lemma *plossless-swap-lassocr* [*plossless-intro*]:
plossless-converter (*$\mathcal{I}1$* $\oplus_\mathcal{I}$ (*$\mathcal{I}2$* $\oplus_\mathcal{I}$ *$\mathcal{I}3$*)) (*$\mathcal{I}2$* $\oplus_\mathcal{I}$ (*$\mathcal{I}1$* $\oplus_\mathcal{I}$ *$\mathcal{I}3$*)) *swap-lassocr*
unfolding *swap-lassocr-def* **by**(*rule plossless-intro WT-intro*)+

lemma *rsuml-lsumr-parallel-converter2*:
map-converter id id rsuml lsumr ((*conv1* |$_=$ *conv2*) |$_=$ *conv3*) =
map-converter rsuml lsumr id id (*conv1* |$_=$ *conv2* |$_=$ *conv3*)
by(*coinduction arbitrary*: *conv1 conv2 conv3*, *clarsimp split*!: *sum.split simp add*: *rel-fun-def map-gpv-conv-map-gpv'*[*symmetric*])
((*subst left-gpv-map*[**where** *h*=*id*] | *subst right-gpv-map*[**where** *h*=*id*])+
, *simp add*:*gpv.map-comp sum.map-id0 o-def prod.map-comp id-def*[*symmetric*]
, *subst map-gpv'-map-gpv-swap*, (*subst rsuml-lsumr-left-gpv-left-gpv* | *subst rsuml-lsumr-left-gpv-right-gpv* | *subst rsuml-lsumr-right-gpv*)
, *auto 4 4 intro*!: *gpv.rel-refl-strong simp add*: *gpv.rel-map*)+

lemma *comp-lassocr$_C$*: ((*conv1* |$_=$ *conv2*) |$_=$ *conv3*) $\odot$ *lassocr$_C$* = *lassocr$_C$* $\odot$ (*conv1* |$_=$ *conv2* |$_=$ *conv3*)
unfolding *lassocr$_C$-def*
by(*subst comp-converter-map-converter2*)
(*simp add*: *comp-converter-id-right comp-converter-map1-out comp-converter-id-left rsuml-lsumr-parallel-converter2*)

lemmas *comp-lassocr$_C$'* = *comp-converter-eqs*[*OF comp-lassocr$_C$*]

lemma *lsumr-rsuml-parallel-converter2*:
map-converter id id lsumr rsuml (*conv1* |$_=$ (*conv2* |$_=$ *conv3*)) =
map-converter lsumr rsuml id id ((*conv1* |$_=$ *conv2*) |$_=$ *conv3*)
by(*coinduction arbitrary*: *conv1 conv2 conv3*, *clarsimp split*!: *sum.split simp add*: *rel-fun-def map-gpv-conv-map-gpv'*[*symmetric*])
((*subst left-gpv-map*[**where** *h*=*id*] | *subst right-gpv-map*[**where** *h*=*id*])+
, *simp add*:*gpv.map-comp sum.map-id0 o-def prod.map-comp id-def*[*symmetric*]
, *subst map-gpv'-map-gpv-swap*, (*subst lsumr-rsuml-left-gpv* | *subst lsumr-rsuml-right-gpv-left-gpv* | *subst lsumr-rsuml-right-gpv-right-gpv*)
, *auto 4 4 intro*!: *gpv.rel-refl-strong simp add*: *gpv.rel-map*)+

lemma *comp-rassocl$_C$*:

(*conv1* |= *conv2* |= *conv3*) ⊙ *rassocl*$_C$ = *rassocl*$_C$ ⊙ ((*conv1* |= *conv2*) |= *conv3*)
unfolding *rassocl*$_C$*-def*
by(*subst comp-converter-map-converter2*)
(*simp add*: *comp-converter-id-right comp-converter-map1-out comp-converter-id-left lsumr-rsuml-parallel-converter2*)

lemmas *comp-rassocl*$_C$*'* = *comp-converter-eqs*[*OF comp-rassocl*$_C$]

lemma *swap-sum-right-gpv*:
map-gpv' id swap-sum swap-sum (*right-gpv gpv*) = *left-gpv gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*: *sum.split intro*: *exI*[**where** *x*=*Fail*])

lemma *swap-sum-left-gpv*:
map-gpv' id swap-sum swap-sum (*left-gpv gpv*) = *right-gpv gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*: *sum.split intro*: *exI*[**where** *x*=*Fail*])

lemma *swap-sum-parallel-converter2*:
map-converter id id swap-sum swap-sum (*conv1* |= *conv2*) =
map-converter swap-sum swap-sum id id (*conv2* |= *conv1*)
by(*coinduction arbitrary*: *conv1 conv2*, *clarsimp simp add*: *rel-fun-def map-gpv-conv-map-gpv'*[*symmetric*] *split*!: *sum.split*)
(*subst map-gpv'-map-gpv-swap*, (*subst swap-sum-right-gpv* | *subst swap-sum-left-gpv*),

auto 4 4 intro!: *gpv.rel-refl-strong simp add*: *gpv.rel-map*)+

lemma *comp-swap*$_C$: (*conv1* |= *conv2*) ⊙ *swap*$_C$ = *swap*$_C$ ⊙ (*conv2* |= *conv1*)
unfolding *swap*$_C$*-def*
by(*subst comp-converter-map-converter2*)
(*simp add*: *comp-converter-id-right comp-converter-map1-out comp-converter-id-left swap-sum-parallel-converter2*)

lemmas *comp-swap*$_C$*'* = *comp-converter-eqs*[*OF comp-swap*$_C$]

lemma *comp-swap-lassocr*: (*conv1* |= *conv2* |= *conv3*) ⊙ *swap-lassocr* = *swap-lassocr* ⊙ (*conv2* |= *conv1* |= *conv3*)
unfolding *swap-lassocr-def comp-rassocl*$_C$*' comp-converter-assoc comp-converter-parallel2' comp-swap*$_C$*' comp-converter-id-right*
by(*subst* (*9*) *comp-converter-id-left*[*symmetric*], *subst comp-converter-parallel2*[*symmetric*])
(*simp add*: *comp-converter-assoc comp-lassocr*$_C$)

lemmas *comp-swap-lassocr'* = *comp-converter-eqs*[*OF comp-swap-lassocr*]

lemma *comp-parallel-wiring*:
((*C1* |= *C2*) |= (*C3* |= *C4*)) ⊙ *parallel-wiring* = *parallel-wiring* ⊙ ((*C1* |= *C3*) |= (*C2* |= *C4*))

unfolding *parallel-wiring-def comp-lassocr$_C$' comp-converter-assoc comp-converter-parallel2' comp-swap-lassocr'*
by(*subst comp-converter-id-right*[*THEN trans*, *OF comp-converter-id-left*[*symmetric*]], *subst comp-converter-parallel2*[*symmetric*])
(*simp add*: *comp-converter-assoc comp-rassocl$_C$*)

lemmas *comp-parallel-wiring' = comp-converter-eqs*[*OF comp-parallel-wiring*]

lemma *attach-converter-of-resource-conv-parallel-resource*:
converter-of-resource res $|_\propto$ *1$_C$* $\rhd$ *res' = res* $\parallel$ *res'*
by(*coinduction arbitrary*: *res res'*)
(*auto 4 3 simp add*: *rel-fun-def map-lift-spmf spmf.map-comp o-def prod.map-comp spmf-rel-map bind-map-spmf map-spmf-conv-bind-spmf*[*symmetric*] *split-def split*!: *sum.split intro*!: *rel-spmf-reflI*)

lemma *attach-converter-of-resource-conv-parallel-resource2*:
1$_C$ $|_\propto$ *converter-of-resource res* $\rhd$ *res' = res'* $\parallel$ *res*
by(*coinduction arbitrary*: *res res'*)
(*auto 4 3 simp add*: *rel-fun-def map-lift-spmf spmf.map-comp o-def prod.map-comp spmf-rel-map bind-map-spmf map-spmf-conv-bind-spmf*[*symmetric*] *split-def split*!: *sum.split intro*!: *rel-spmf-reflI*)

lemma *plossless-parallel-wiring* [*plossless-intro*]:
plossless-converter (($\mathcal{I}1$ $\oplus_\mathcal{I}$ $\mathcal{I}2$) $\oplus_\mathcal{I}$ ($\mathcal{I}3$ $\oplus_\mathcal{I}$ $\mathcal{I}4$)) (($\mathcal{I}1$ $\oplus_\mathcal{I}$ $\mathcal{I}3$) $\oplus_\mathcal{I}$ ($\mathcal{I}2$ $\oplus_\mathcal{I}$ $\mathcal{I}4$)) *parallel-wiring*
unfolding *parallel-wiring-def* **by**(*rule plossless-intro WT-intro*)+

lemma *run-converter-lassocr* [*simp*]:
run-converter lassocr$_C$ x = Pause (*rsuml x*) (λx. *Done* (*lsumr x*, *lassocr$_C$*))
by(*simp add*: *lassocr$_C$-def o-def*)

lemma *run-converter-rassocl* [*simp*]:
run-converter rassocl$_C$ x = Pause (*lsumr x*) (λx. *Done* (*rsuml x*, *rassocl$_C$*))
by(*simp add*: *rassocl$_C$-def o-def*)

lemma *run-converter-swap* [*simp*]: *run-converter swap$_C$ x = Pause* (*swap-sum x*) (λx. *Done* (*swap-sum x*, *swap$_C$*))
by(*simp add*: *swap$_C$-def o-def*)

definition *lassocr-swap-sum* **where** *lassocr-swap-sum = rsuml* $\circ$ *map-sum swap-sum id* $\circ$ *lsumr*

lemma *run-converter-swap-lassocr* [*simp*]:
run-converter swap-lassocr x = Pause (*lassocr-swap-sum x*) (
case lsumr x of Inl - ⇒ (λy. *case lsumr y of Inl - ⇒ Done* (*lassocr-swap-sum y*, *swap-lassocr*) | *- ⇒ Fail*)
| *Inr - ⇒* (λy. *case lsumr y of Inl - ⇒ Fail* | *Inr - ⇒ Done* (*lassocr-swap-sum y*, *swap-lassocr*)))

by(*subst sum.case-distrib*[**where** *h*=λ*x*. *inline* - *x* -] |
simp add: *bind-rpv-def inline-map-gpv split-def map-gpv-conv-bind*[*symmetric*]
swap-lassocr-def o-def cong del: *sum.case-cong*)+
(*cases x rule*: *lsumr.cases*, *simp-all add*: *o-def lassocr-swap-sum-def gpv.map-comp*
fun-eq-iff cong: *sum.case-cong split*: *sum.split*)

definition *parallel-sum-wiring* **where** *parallel-sum-wiring* = *lsumr* ∘ *map-sum id*
lassocr-swap-sum ∘ *rsuml*

lemma *run-converter-parallel-wiring*:
run-converter parallel-wiring x = *Pause* (*parallel-sum-wiring x*) (
case rsuml x of Inl - ⇒ (λ*y*. *case rsuml y of Inl* - ⇒ *Done* (*parallel-sum-wiring*
y, *parallel-wiring*) | - ⇒ *Fail*)
| *Inr x* ⇒ (*case lsumr x of Inl* - ⇒ (λ*y*. *case rsuml y of Inl* - ⇒ *Fail*
| *Inr x* ⇒ (*case lsumr x of Inl* - ⇒ *Done* (*parallel-sum-wiring y*, *parallel-wiring*) |
Inr - ⇒ *Fail*))
| *Inr* - ⇒ (λ*y*. *case rsuml y of Inl* - ⇒ *Fail*
| *Inr x* ⇒ (*case lsumr x of Inl* - ⇒ *Fail* | *Inr* - ⇒ *Done* (*parallel-sum-wiring y*,
parallel-wiring)))))
by(*simp add*: *parallel-wiring-def o-def cong del*: *sum.case-cong add*: *split-def*
map-gpv-conv-bind[*symmetric*])
(*subst sum.case-distrib*[**where** *h*=λ*x*. *right-rpv x* -] |
subst sum.case-distrib[**where** *h*=λ*x*. *inline* - *x* -] |
subst sum.case-distrib[**where** *h*=*right-gpv*] |
(*auto simp add*: *inline-map-gpv bind-rpv-def gpv.map-comp fun-eq-iff parallel-sum-wiring-def*

parallel-wiring-def[*symmetric*] *sum.case-distrib o-def intro*: *sym cong del*:
sum.case-cong split!: *sum.split*))+

lemma *bound-lassocr*$_C$ [*interaction-bound*]: *interaction-any-bounded-converter lassocr*$_C$ *1*
unfolding *lassocr*$_C$*-def* **by** *interaction-bound-converter simp*

lemma *bound-rassocl*$_C$ [*interaction-bound*]: *interaction-any-bounded-converter rassocl*$_C$ *1*
unfolding *rassocl*$_C$*-def* **by** *interaction-bound-converter simp*

lemma *bound-swap*$_C$ [*interaction-bound*]: *interaction-any-bounded-converter swap*$_C$ *1*
unfolding *swap*$_C$*-def* **by** *interaction-bound-converter simp*

lemma *bound-swap-rassocl* [*interaction-bound*]: *interaction-any-bounded-converter swap-rassocl 1*
unfolding *swap-rassocl-def* **by** *interaction-bound-converter simp*

lemma *bound-swap-lassocr* [*interaction-bound*]: *interaction-any-bounded-converter swap-lassocr 1*
unfolding *swap-lassocr-def* **by** *interaction-bound-converter simp*

lemma *bound-parallel-wiring* [*interaction-bound*]: *interaction-any-bounded-converter parallel-wiring 1*
unfolding *parallel-wiring-def* **by** *interaction-bound-converter simp*

6.3 Characterization of wirings

type-synonym $('a, 'b, 'c, 'd)$ *wiring* $= ('a \Rightarrow 'c) \times ('d \Rightarrow 'b)$

inductive *wiring* :: $('a, 'b)\ \mathcal{I} \Rightarrow ('c, 'd)\ \mathcal{I} \Rightarrow ('a, 'b, 'c, 'd)$ *converter* $\Rightarrow ('a, 'b, 'c, 'd)$ *wiring* $\Rightarrow$ *bool*
for $\mathcal{I}\ \mathcal{I}'$ *cnv*
where
wiring:
wiring $\mathcal{I}\ \mathcal{I}'$ *cnv* (f, g) **if**
$\mathcal{I}$,$\mathcal{I}' \vdash_C$ *cnv* $\sim$ *map-converter id id f g* 1_C
$\mathcal{I}$,$\mathcal{I}' \vdash_C$ *cnv* $\surd$

lemmas *wiringI* = *wiring*
hide-fact *wiring*

lemma *wiringD*:
assumes *wiring* $\mathcal{I}\ \mathcal{I}'$ *cnv* (f, g)
shows *wiringD-eq*: $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *cnv* $\sim$ *map-converter id id f g* 1_C
and *wiringD-WT*: $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *cnv* $\surd$
using *assms* **by**(*cases*, *blast*)+

named-theorems *wiring-intro introduction rules for wiring*

definition *apply-wiring* :: $('a, 'b, 'c, 'd)$ *wiring* $\Rightarrow ('s, 'c, 'd)$ *oracle'* $\Rightarrow ('s, 'a, 'b)$ *oracle'*
where *apply-wiring* $= (\lambda(f, g).$ *map-fun id* (*map-fun f* (*map-spmf* (*map-prod g id*))))

lemma *apply-wiring-simps*: *apply-wiring* (f, g) = *map-fun id* (*map-fun f* (*map-spmf* (*map-prod g id*)))
by(*simp add*: *apply-wiring-def*)

lemma *attach-wiring-resource-of-oracle*:
assumes *wiring*: *wiring* $\mathcal{I}1\ \mathcal{I}2$ *conv fg*
and *WT*: $\mathcal{I}2 \vdash$*res RES res s* $\surd$
and *outs*: *outs-*$\mathcal{I}\ \mathcal{I}1$ = *UNIV*
shows *conv* $\rhd$ *RES res s* = *RES* (*apply-wiring fg res*) *s*
using *wiring*
proof *cases*
case (*wiring f g*)
have $\mathcal{I}$*-full*,$\mathcal{I}2 \vdash_C$ *conv* $\sim$ *map-converter id id f g* 1_C **using** *wiring*(*2*)
by(*rule eq-*$\mathcal{I}$*-converter-mono*)(*simp-all add*: *le-*$\mathcal{I}$*-def outs*)
with *WT* **have** *conv* $\rhd$ *RES res s* = *map-converter id id f g* $1_C \rhd$ *RES res s*

by(*rule eq-ℐ-attach*)
also have ... = *RES* (*apply-wiring fg res*) *s*
by(*simp add*: *attach-map-converter map-resource-resource-of-oracle prod.map-id0 option.map-id0 map-fun-id apply-wiring-def wiring(1)*)
finally show *?thesis* **.**
qed

lemma *wiring-id-converter* [*simp*, *wiring-intro*]: *wiring ℐ ℐ 1*$_C$ (*id*, *id*)
using *wiring.intros*[*of ℐ ℐ 1*$_C$ *id id*]
by(*simp add*: *eq-ℐ-converter-reflI*)

lemma *apply-wiring-id* [*simp*]: *apply-wiring* (*id*, *id*) *res* = *res*
by(*simp add*: *apply-wiring-simps prod.map-id0 option.map-id0 map-fun-id*)

definition *attach-wiring* :: ('*a*, '*b*, '*c*, '*d*) *wiring* ⇒ ('*s* ⇒ '*c* ⇒ ('*d* × '*s*, '*e*, '*f*) *gpv*) ⇒ ('*s* ⇒ '*a* ⇒ ('*b* × '*s*, '*e*, '*f*) *gpv*)
where *attach-wiring* = (λ(*f*, *g*). *map-fun id* (*map-fun f* (*map-gpv* (*map-prod g id*) *id*)))

lemma *attach-wiring-simps*: *attach-wiring* (*f*, *g*) = *map-fun id* (*map-fun f* (*map-gpv* (*map-prod g id*) *id*))
by(*simp add*: *attach-wiring-def*)

lemma *comp-wiring-converter-of-callee*:
assumes *wiring*: *wiring ℐ1 ℐ2 conv w*
and *WT*: *ℐ2*, *ℐ3* ⊢$_C$ *CNV callee s* √
shows *ℐ1*, *ℐ3* ⊢$_C$ *conv* ⊙ *CNV callee s* ∼ *CNV* (*attach-wiring w callee*) *s*
using *wiring*
proof *cases*
case (*wiring f g*)
from *wiring(2)* **have** *ℐ1*,*ℐ3* ⊢$_C$ *conv* ⊙ *CNV callee s* ∼ *map-converter id id f g 1*$_C$ ⊙ *CNV callee s*
by(*rule eq-ℐ-comp-cong*)(*rule eq-ℐ-converter-reflI*[*OF WT*])
also have *map-converter id id f g 1*$_C$ ⊙ *CNV callee s* = *map-converter f g id id* (*CNV callee s*)
by(*subst comp-converter-map-converter1*)(*simp add*: *comp-converter-id-left*)
also have ... = *CNV* (*attach-wiring w callee*) *s*
by(*simp add*: *map-converter-of-callee attach-wiring-simps wiring(1) map-gpv-conv-map-gpv'*)
finally show *?thesis* **.**
qed

definition *comp-wiring* :: ('*a*, '*b*, '*c*, '*d*) *wiring* ⇒ ('*c*, '*d*, '*e*, '*f*) *wiring* ⇒ ('*a*, '*b*, '*e*, '*f*) *wiring* (**infixl** ∘$_w$ *55*)
where *comp-wiring* = (λ(*f*, *g*) (*f'*, *g'*). (*f'* ∘ *f*, *g* ∘ *g'*))

lemma *comp-wiring-simps*: *comp-wiring* (*f*, *g*) (*f'*, *g'*) = (*f'* ∘ *f*, *g* ∘ *g'*)
by(*simp add*: *comp-wiring-def*)

lemma *wiring-comp-converterI* [*wiring-intro*]:

wiring $\mathcal{I}$ $\mathcal{I}''$ (*conv1* $\odot$ *conv2*) (*fg* $\circ_w$ *fg'*) **if** *wiring* $\mathcal{I}$ $\mathcal{I}'$ *conv1 fg wiring* $\mathcal{I}'$ $\mathcal{I}''$ *conv2 fg'*
proof −
from *that*(*1*) **obtain** *f g*
where *conv1*: $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *conv1* $\sim$ *map-converter id id f g* 1_C
and *WT1*: $\mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv1* $\surd$
and [*simp*]: *fg* = (*f*, *g*)
by *cases*
from *that*(*2*) **obtain** *f' g'*
where *conv2*: $\mathcal{I}'$,$\mathcal{I}'' \vdash_C$ *conv2* $\sim$ *map-converter id id f' g'* 1_C
and *WT2*: $\mathcal{I}'$, $\mathcal{I}'' \vdash_C$ *conv2* $\surd$
and [*simp*]: *fg'* = (*f'*, *g'*)
by *cases*
have $*$: (*fg* $\circ_w$ *fg'*) = (*f'* $\circ$ *f*, *g* $\circ$ *g'*) **by**(*simp add*: *comp-wiring-simps*)
have $\mathcal{I}$,$\mathcal{I}'' \vdash_C$ *conv1* $\odot$ *conv2* $\sim$ *map-converter id id f g* 1_C $\odot$ *map-converter id id f' g'* 1_C
using *conv1 conv2* **by**(*rule eq-*$\mathcal{I}$*-comp-cong*)
also have *map-converter id id f g* 1_C $\odot$ *map-converter id id f' g'* 1_C = *map-converter id id* (*f'* $\circ$ *f*) (*g* $\circ$ *g'*) 1_C
by(*simp add*: *comp-converter-map-converter2 comp-converter-id-right*)
also have $\mathcal{I}$,$\mathcal{I}'' \vdash_C$ *conv1* $\odot$ *conv2* $\surd$ **using** *WT1 WT2* **by**(*rule WT-converter-comp*)
ultimately show *?thesis* **unfolding** $*$ **..**
qed

definition *parallel2-wiring*
:: ('*a*, '*b*, '*c*, '*d*) *wiring* $\Rightarrow$ ('*a'*, '*b'*, '*c'*, '*d'*) *wiring*
$\Rightarrow$ ('*a* + '*a'*, '*b* + '*b'*, '*c* + '*c'*, '*d* + '*d'*) *wiring* (**infix** $|_w$ *501*) **where**
parallel2-wiring = (λ(*f*, *g*) (*f'*, *g'*). (*map-sum f f'*, *map-sum g g'*))

lemma *parallel2-wiring-simps*:
parallel2-wiring (*f*, *g*) (*f'*, *g'*) = (*map-sum f f'*, *map-sum g g'*)
by(*simp add*: *parallel2-wiring-def*)

lemma *wiring-parallel-converter2* [*simp*, *wiring-intro*]:
assumes *wiring I1 I1' conv1 fg*
and *wiring I2 I2' conv2 fg'*
shows *wiring* (*I1* $\oplus_{\mathcal{I}}$ *I2*) (*I1'* $\oplus_{\mathcal{I}}$ *I2'*) (*conv1* $|_=$ *conv2*) (*fg* $|_w$ *fg'*)
proof −
from *assms*(*1*) **obtain** *f1 g1*
where *conv1*: *I1*,*I1'* $\vdash_C$ *conv1* $\sim$ *map-converter id id f1 g1* 1_C
and *WT1*: *I1*, *I1'* $\vdash_C$ *conv1* $\surd$
and [*simp*]: *fg* = (*f1*, *g1*)
by *cases*
from *assms*(*2*) **obtain** *f2 g2*
where *conv2*: *I2*,*I2'* $\vdash_C$ *conv2* $\sim$ *map-converter id id f2 g2* 1_C
and *WT2*: *I2*, *I2'* $\vdash_C$ *conv2* $\surd$
and [*simp*]: *fg'* = (*f2*, *g2*)
by *cases*
from *eq-*$\mathcal{I}$*-converterD-WT1*[*OF conv1 WT1*] **have** *I1*: *I1* $\leq$ *map-*$\mathcal{I}$ *f1 g1 I1'*

by(*rule WT-map-converter-idD*)
from *eq-$\mathcal{I}$-converterD-WT1*[*OF conv2 WT2*] **have** $\mathcal{I}2$: $\mathcal{I}2 \leq$ *map-$\mathcal{I}$ f2 g2* $\mathcal{I}2'$
by(*rule WT-map-converter-idD*)
have *WT'*: $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C$ *map-converter id id* (*map-sum f1 f2*) (*map-sum g1 g2*) 1_C $\surd$
by(*auto intro*!: *WT-converter-map-converter WT-converter-mono*[*OF WT-converter-id order-refl*] $\mathcal{I}1$ $\mathcal{I}2$)
have $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C$ *conv1* $|_=$ *conv2* $\sim$ *map-converter id id f1 g1* 1_C $|_=$ *map-converter id id f2 g2* 1_C
using *conv1 conv2* **by**(*rule parallel-converter2-eq-$\mathcal{I}$-cong*)
also have *map-converter id id f1 g1* 1_C $|_=$ *map-converter id id f2 g2* 1_C = (1_C $|_=$ 1_C) $\odot$ *map-converter id id* (*map-sum f1 f2*) (*map-sum g1 g2*) 1_C
by(*simp add*: *parallel-converter2-map2-out parallel-converter2-map1-out map-sum.comp sum.map-id0 comp-converter-map-converter2*[*of - id id*, *simplified*] *comp-converter-id-right*)
also have $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C \ldots \sim 1_C \odot$ *map-converter id id* (*map-sum f1 f2*) (*map-sum g1 g2*) 1_C
by(*rule eq-$\mathcal{I}$-comp-cong*[*OF parallel-converter2-id-id*])(*rule eq-$\mathcal{I}$-converter-reflI*[*OF WT'*])
finally show *?thesis* **using** *WT1 WT2*
by(*auto simp add*: *parallel2-wiring-simps comp-converter-id-left intro*!: *wiringI WT-converter-parallel-converter2*)
qed

lemma *apply-parallel2* [*simp*]:
apply-wiring (*fg* $|_w$ *fg'*) (*res1* $\oplus_O$ *res2*) = (*apply-wiring fg res1* $\oplus_O$ *apply-wiring fg' res2*)
proof −
have[*simp*]: *fg* = (*f1*, *g1*) $\Longrightarrow$ *fg'* = (*f2*, *g2*) $\Longrightarrow$
map-spmf (*map-prod* (*map-sum g1 g2*) *id*) ((*res1* $\oplus_O$ *res2*) *s* (*map-sum f1 f2 q*)) =
((λ*s q*. *map-spmf* (*map-prod g1 id*) (*res1 s* (*f1 q*))) $\oplus_O$ (λ*s q*. *map-spmf* (*map-prod g2 id*) (*res2 s* (*f2 q*)))) *s q* **for** *f1 g1 f2 g2 s q*
by(*cases q*)(*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp split*!:*sum.splits*)

show *?thesis*
by(*cases fg*; *cases fg'*) (*clarsimp simp add*: *parallel2-wiring-simps apply-wiring-simps fun-eq-iff map-fun-def o-def*)
qed

lemma *apply-comp-wiring* [*simp*]: *apply-wiring* (*fg* $\circ_w$ *fg'*) *res* = *apply-wiring fg* (*apply-wiring fg' res*)
by(*cases fg*; *cases fg'*)(*simp add*: *comp-wiring-simps apply-wiring-simps map-fun-def fun-eq-iff spmf.map-comp prod.map-comp o-def id-def*)

definition $lassocr_w$ :: (('*a* + '*b*) + '*c*, ('*d* + '*e*) + '*f*, '*a* + ('*b* + '*c*), '*d* + ('*e* + '*f*)) *wiring*
where $lassocr_w$ = (*rsuml*, *lsumr*)

definition $rassocl_w$ :: ('*a* + ('*b* + '*c*), '*d* + ('*e* + '*f*), ('*a* + '*b*) + '*c*, ('*d* + '*e*) +

$'f$) *wiring*
where $rassocl_w$ = (*lsumr*, *rsuml*)

definition $swap_w$ :: ($'a$ + $'b$, $'c$ + $'d$, $'b$ + $'a$, $'d$ + $'c$) *wiring* **where**
$swap_w$ = (*swap-sum*, *swap-sum*)

lemma *wiring-lassocr* [*simp*, *wiring-intro*]:
wiring ((*I1* $\oplus_I$ *I2*) $\oplus_I$ *I3*) (*I1* $\oplus_I$ (*I2* $\oplus_I$ *I3*)) $lassocr_C$ $lassocr_w$
unfolding $lassocr_C$*-def* $lassocr_w$*-def*
by(*subst map-converter-id-move-right*)(*auto intro*!: *wiringI eq-I-converter-reflI WT-converter-map-converter*)

lemma *wiring-rassocl* [*simp*, *wiring-intro*]:
wiring (*I1* $\oplus_I$ (*I2* $\oplus_I$ *I3*)) ((*I1* $\oplus_I$ *I2*) $\oplus_I$ *I3*) $rassocl_C$ $rassocl_w$
unfolding $rassocl_C$*-def* $rassocl_w$*-def*
by(*subst map-converter-id-move-right*)(*auto intro*!: *wiringI eq-I-converter-reflI WT-converter-map-converter*)

lemma *wiring-swap* [*simp*, *wiring-intro*]: *wiring* (*I1* $\oplus_I$ *I2*) (*I2* $\oplus_I$ *I1*) $swap_C$ $swap_w$
unfolding $swap_C$*-def* $swap_w$*-def*
by(*subst map-converter-id-move-right*)(*auto intro*!: *wiringI eq-I-converter-reflI WT-converter-map-converter*)

lemma $apply\text{-}lassocr_w$ [*simp*]: *apply-wiring* $lassocr_w$ (*res1* $\oplus_O$ *res2* $\oplus_O$ *res3*) = (*res1* $\oplus_O$ *res2*) $\oplus_O$ *res3*
by(*simp add*: *apply-wiring-def* $lassocr_w$*-def map-rsuml-plus-oracle*)

lemma $apply\text{-}rassocl_w$ [*simp*]: *apply-wiring* $rassocl_w$ ((*res1* $\oplus_O$ *res2*) $\oplus_O$ *res3*) = *res1* $\oplus_O$ *res2* $\oplus_O$ *res3*
by(*simp add*: *apply-wiring-def* $rassocl_w$*-def map-lsumr-plus-oracle*)

lemma $apply\text{-}swap_w$ [*simp*]: *apply-wiring* $swap_w$ (*res1* $\oplus_O$ *res2*) = *res2* $\oplus_O$ *res1*
by(*simp add*: *apply-wiring-def* $swap_w$*-def map-swap-sum-plus-oracle*)

end

7 Security

theory *Constructive-Cryptography* **imports**
Wiring
begin

definition *advantage* $\mathcal{A}$ *res1 res2* = |*spmf* (*connect* $\mathcal{A}$ *res1*) *True* − *spmf* (*connect* $\mathcal{A}$ *res2*) *True*|

locale *constructive-security-aux* =
fixes *real-resource* :: *security* ⇒ ($'a$ + $'e$, $'b$ + $'f$) *resource*
and *ideal-resource* :: *security* ⇒ ($'c$ + $'e$, $'d$ + $'f$) *resource*

and *sim* :: *security* ⇒ ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$-*real* :: *security* ⇒ ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$-*ideal* :: *security* ⇒ ('*c*, '*d*) $\mathcal{I}$
and $\mathcal{I}$-*common* :: *security* ⇒ ('*e*, '*f*) $\mathcal{I}$
and *bound* :: *security* ⇒ *enat*
and *lossless* :: *bool*
assumes *WT-real* [*WT-intro*]: ⋀η. $\mathcal{I}$-*real* η ⊕$_\mathcal{I}$ $\mathcal{I}$-*common* η ⊢*res real-resource* η √
and *WT-ideal* [*WT-intro*]: ⋀η. $\mathcal{I}$-*ideal* η ⊕$_\mathcal{I}$ $\mathcal{I}$-*common* η ⊢*res ideal-resource* η √
and *WT-sim* [*WT-intro*]: ⋀η. $\mathcal{I}$-*real* η, $\mathcal{I}$-*ideal* η ⊢$_C$ *sim* η √
and *adv*: ⋀$\mathcal{A}$ :: *security* ⇒ ('*a* + '*e*, '*b* + '*f*) *distinguisher*.
⟦ ⋀η. $\mathcal{I}$-*real* η ⊕$_\mathcal{I}$ $\mathcal{I}$-*common* η ⊢*g* $\mathcal{A}$ η √;
⋀η. *interaction-bounded-by* (λ-. *True*) ($\mathcal{A}$ η) (*bound* η);
⋀η. *lossless* ⟹ *plossless-gpv* ($\mathcal{I}$-*real* η ⊕$_\mathcal{I}$ $\mathcal{I}$-*common* η) ($\mathcal{A}$ η) ⟧
⟹ *negligible* (λη. *advantage* ($\mathcal{A}$ η) (*sim* η |$=$ 1$_C$ ▷ *ideal-resource* η) (*real-resource* η))

locale *constructive-security* =
constructive-security-aux real-resource ideal-resource sim $\mathcal{I}$-*real* $\mathcal{I}$-*ideal* $\mathcal{I}$-*common bound lossless*
for *real-resource* :: *security* ⇒ ('*a* + '*e*, '*b* + '*f*) *resource*
and *ideal-resource* :: *security* ⇒ ('*c* + '*e*, '*d* + '*f*) *resource*
and *sim* :: *security* ⇒ ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$-*real* :: *security* ⇒ ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$-*ideal* :: *security* ⇒ ('*c*, '*d*) $\mathcal{I}$
and $\mathcal{I}$-*common* :: *security* ⇒ ('*e*, '*f*) $\mathcal{I}$
and *bound* :: *security* ⇒ *enat*
and *lossless* :: *bool*
and *w* :: *security* ⇒ ('*c*, '*d*, '*a*, '*b*) *wiring*
+
assumes *correct*: ∃ *cnv*. ∀$\mathcal{D}$ :: *security* ⇒ ('*c* + '*e*, '*d* + '*f*) *distinguisher*.
(∀η. $\mathcal{I}$-*ideal* η ⊕$_\mathcal{I}$ $\mathcal{I}$-*common* η ⊢*g* $\mathcal{D}$ η √)
⟶ (∀η. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound* η))
⟶ (∀η. *lossless* ⟶ *plossless-gpv* ($\mathcal{I}$-*ideal* η ⊕$_\mathcal{I}$ $\mathcal{I}$-*common* η) ($\mathcal{D}$ η))
⟶ (∀η. *wiring* ($\mathcal{I}$-*ideal* η) ($\mathcal{I}$-*real* η) (*cnv* η) (*w* η)) ∧
negligible (λη. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η |$=$ 1$_C$ ▷ *real-resource* η))

locale *constructive-security2* =
constructive-security-aux real-resource ideal-resource sim $\mathcal{I}$-*real* $\mathcal{I}$-*ideal* $\mathcal{I}$-*common bound lossless*
for *real-resource* :: *security* ⇒ ('*a* + '*e*, '*b* + '*f*) *resource*
and *ideal-resource* :: *security* ⇒ ('*c* + '*e*, '*d* + '*f*) *resource*
and *sim* :: *security* ⇒ ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$-*real* :: *security* ⇒ ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$-*ideal* :: *security* ⇒ ('*c*, '*d*) $\mathcal{I}$
and $\mathcal{I}$-*common* :: *security* ⇒ ('*e*, '*f*) $\mathcal{I}$

and *bound* :: *security* ⇒ *enat*
and *lossless* :: *bool*
and *w* :: *security* ⇒ ('*c*, '*d*, '*a*, '*b*) *wiring*
+
assumes *sim*: ∃ *cnv*. ∀ η. *wiring* (*I*-*ideal* η) (*I*-*real* η) (*cnv* η) (*w* η) ∧ *wiring* (*I*-*ideal* η) (*I*-*ideal* η) (*cnv* η ⊙ *sim* η) (*id*, *id*)
begin

lemma *constructive-security*:
constructive-security real-resource ideal-resource sim I-real I-ideal I-common bound lossless w
proof
from *sim* **obtain** *cnv*
where *w*: ⋀η. *wiring* (*I*-*ideal* η) (*I*-*real* η) (*cnv* η) (*w* η)
and *inverse*: ⋀η. *wiring* (*I*-*ideal* η) (*I*-*ideal* η) (*cnv* η ⊙ *sim* η) (*id*, *id*)
by *blast*
show ∃ *cnv*. ∀ *D*. (∀ η. *I*-*ideal* η ⊕*I* *I*-*common* η ⊢*g* *D* η √)
⟶ (∀ η. *interaction-any-bounded-by* (*D* η) (*bound* η))
⟶ (∀ η. *lossless* ⟶ *plossless-gpv* (*I*-*ideal* η ⊕*I* *I*-*common* η) (*D* η))
⟶ (∀ η. *wiring* (*I*-*ideal* η) (*I*-*real* η) (*cnv* η) (*w* η)) ∧
negligible (λη. *advantage* (*D* η) (*ideal-resource* η) (*cnv* η |= 1*C* ▷ *real-resource* η))
proof(*intro strip exI conjI*)
fix *D* :: *security* ⇒ ('*c* + '*e*, '*d* + '*f*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: ∀ η. *I*-*ideal* η ⊕*I* *I*-*common* η ⊢*g* *D* η √
and *bound* [*rule-format*, *interaction-bound*]: ∀ η. *interaction-bounded-by* (λ-. *True*) (*D* η) (*bound* η)
and *lossless* [*rule-format*]: ∀ η. *lossless* ⟶ *plossless-gpv* (*I*-*ideal* η ⊕*I* *I*-*common* η) (*D* η)

show *wiring* (*I*-*ideal* η) (*I*-*real* η) (*cnv* η) (*w* η) **for** η **by** *fact*

let *?A* = λη. *outs-I* (*I*-*ideal* η)
let *?cnv* = λη. *restrict-converter* (*?A* η) (*I*-*real* η) (*cnv* η)
let *?A* = λη. *absorb* (*D* η) (*?cnv* η |= 1*C*)

have *eq*: *advantage* (*D* η) (*ideal-resource* η) (*cnv* η |= 1*C* ▷ *real-resource* η) =
advantage (*?A* η) (*sim* η |= 1*C* ▷ *ideal-resource* η) (*real-resource* η) **for** η
proof −
from *w*[*of* η] **have** [*WT-intro*]: *I*-*ideal* η, *I*-*real* η ⊢*C* *cnv* η √ **by** *cases*
have *I*-*ideal* η, *I*-*ideal* η ⊢*C* *?cnv* η ⊙ *sim* η ∼ *cnv* η ⊙ *sim* η
by(*rule eq-I-comp-cong eq-I-restrict-converter WT-intro order-refl eq-I-converter-reflI*)+
also from *inverse*[*of* η] **have** *I*-*ideal* η, *I*-*ideal* η ⊢*C* *cnv* η ⊙ *sim* η ∼ 1*C*
by *cases simp*
finally have *inverse'*: *I*-*ideal* η, *I*-*ideal* η ⊢*C* *?cnv* η ⊙ *sim* η ∼ 1*C* .
hence *I*-*ideal* η ⊕*I* *I*-*common* η, *I*-*ideal* η ⊕*I* *I*-*common* η ⊢*C* *?cnv* η ⊙ *sim* η |= 1*C* ∼ 1*C* |= 1*C*
by(*rule parallel-converter2-eq-I-cong*)(*intro eq-I-converter-reflI WT-intro*)

also have $\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η, $\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η $\vdash_C$ 1_C $|_=$ $1_C \sim 1_C$
by(*rule parallel-converter2-id-id*)
also
have *eq1*: *connect* ($\mathcal{D}$ η) (*?cnv* η $|_=$ 1_C $\rhd$ *sim* η $|_=$ 1_C $\rhd$ *ideal-resource* η) =
connect ($\mathcal{D}$ η) (1_C $\rhd$ *ideal-resource* η)
unfolding *attach-compose*[*symmetric*] *comp-converter-parallel2 comp-converter-id-right*
by(*rule connect-eq-resource-cong WT-intro eq-*$\mathcal{I}$*-attach-on' calculation*)+(*fastforce intro*: *WT-intro*)+

have $*$: $\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η, $\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η $\vdash_C$ *?cnv* η $|_=$ $1_C \sim$ *cnv* η $|_=$ 1_C
by(*rule parallel-converter2-eq-*$\mathcal{I}$*-cong eq-*$\mathcal{I}$*-restrict-converter*)+(*auto intro*: *WT-intro eq-*$\mathcal{I}$*-converter-reflI*)
have *eq2*: *connect* ($\mathcal{D}$ η) (*?cnv* η $|_=$ 1_C $\rhd$ *real-resource* η) = *connect* ($\mathcal{D}$ η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η)
by(*rule connect-eq-resource-cong WT-intro eq-*$\mathcal{I}$*-attach-on'* $*$)+(*auto intro*: *WT-intro*)
show *?thesis* **unfolding** *advantage-def* **by**(*simp add*: *distinguish-attach*[*symmetric*] *eq1 eq2*)
qed
have $\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η $\vdash g$ *?*$\mathcal{A}$ η $\surd$ **for** η
proof −
from *w* **have** [*WT-intro*]: $\mathcal{I}$-*ideal* η, $\mathcal{I}$-*real* η $\vdash_C$ *cnv* η $\surd$ **by** *cases*
show *?thesis* **by**(*rule WT-intro*)+
qed
moreover
have *interaction-any-bounded-by* (*absorb* ($\mathcal{D}$ η) (*?cnv* η $|_=$ 1_C)) (*bound* η) **for** η
proof −
from *w*[*of* η] **obtain** *f g* **where** [*simp*]: *w* η = (*f*, *g*)
and [*WT-intro*]: $\mathcal{I}$-*ideal* η, $\mathcal{I}$-*real* η $\vdash_C$ *cnv* η $\surd$
and *eq*: $\mathcal{I}$-*ideal* η, $\mathcal{I}$-*real* η $\vdash_C$ *cnv* η $\sim$ *map-converter id id f g* 1_C **by** *cases*
from *eq-*$\mathcal{I}$*-restrict-converter-cong*[*OF eq order-refl*]
have $*$: *restrict-converter* (*?A* η) ($\mathcal{I}$-*real* η) (*cnv* η) =
restrict-converter (*?A* η) ($\mathcal{I}$-*real* η) (*map-converter f g id id* 1_C)
by(*subst map-converter-id-move-right*) *simp*
show *?thesis* **unfolding** $*$ **by** *interaction-bound-converter simp*
qed
moreover have *plossless-gpv* ($\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η) (*?*$\mathcal{A}$ η)
if *lossless* **for** η
proof −
from *w*[*of* η] **obtain** *f g* **where** [*simp*]: *w* η = (*f*, *g*)
and *cnv* [*WT-intro*]: $\mathcal{I}$-*ideal* η, $\mathcal{I}$-*real* η $\vdash_C$ *cnv* η $\surd$
and *eq*: $\mathcal{I}$-*ideal* η, $\mathcal{I}$-*real* η $\vdash_C$ *cnv* η $\sim$ *map-converter id id f g* 1_C **by** *cases*
from *eq-*$\mathcal{I}$*-converterD-WT1*[*OF eq cnv*] **have** $\mathcal{I}$: $\mathcal{I}$-*ideal* η $\leq$ *map-*$\mathcal{I}$ *f g* ($\mathcal{I}$-*real* η)
by(*rule WT-map-converter-idD*)

with *WT-converter-id* **have** [*WT-intro*]: $\mathcal{I}$*-ideal* η, *map-*$\mathcal{I}$ *f g* ($\mathcal{I}$*-real* η) $\vdash_C$ 1_C $\surd$
by(*rule WT-converter-mono*) *simp*
have *id*: *plossless-converter* ($\mathcal{I}$*-ideal* η) (*map-*$\mathcal{I}$ *f g* ($\mathcal{I}$*-real* η)) 1_C
by(*rule plossless-converter-mono*)(*rule plossless-id-converter order-refl* $\mathcal{I}$ *WT-intro*)+
show *?thesis* **unfolding** *eq-*$\mathcal{I}$*-restrict-converter-cong*[*OF eq order-refl*]
by(*rule plossless-gpv-absorb lossless*[*OF that*] *plossless-parallel-converter2 plossless-restrict-converter plossless-map-converter*)+
(*fastforce intro*: *WT-intro id WT-converter-map-converter*)+
qed
ultimately show *negligible* ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))
unfolding *eq* **by**(*rule adv*)
qed
qed

sublocale *constructive-security real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common bound lossless w*
by(*rule constructive-security*)

end

7.1 Composition theorems

theorem *composability*:
fixes *real*
assumes *constructive-security middle ideal sim-inner* $\mathcal{I}$*-middle* $\mathcal{I}$*-inner* $\mathcal{I}$*-common bound-inner lossless-inner w1*
assumes *constructive-security real middle sim-outer* $\mathcal{I}$*-real* $\mathcal{I}$*-middle* $\mathcal{I}$*-common bound-outer lossless-outer w2*
and *bound* [*interaction-bound*]: $\bigwedge\eta$. *interaction-any-bounded-converter* (*sim-outer* η) (*bound-sim* η)
and *bound-le*: $\bigwedge\eta$. *bound-outer* η $*$ *max* (*bound-sim* η) *1* $\leq$ *bound-inner* η
and *lossless-sim* [*plossless-intro*]: $\bigwedge\eta$. *lossless-inner* $\Longrightarrow$ *plossless-converter* ($\mathcal{I}$*-real* η) ($\mathcal{I}$*-middle* η) (*sim-outer* η)
shows *constructive-security real ideal* ($\lambda\eta$. *sim-outer* η $\odot$ *sim-inner* η) $\mathcal{I}$*-real* $\mathcal{I}$*-inner* $\mathcal{I}$*-common bound-outer* (*lossless-outer* $\vee$ *lossless-inner*) ($\lambda\eta$. *w1* η $\circ_w$ *w2* η)
proof
interpret *inner*: *constructive-security middle ideal sim-inner* $\mathcal{I}$*-middle* $\mathcal{I}$*-inner* $\mathcal{I}$*-common bound-inner lossless-inner w1* **by** *fact*
interpret *outer*: *constructive-security real middle sim-outer* $\mathcal{I}$*-real* $\mathcal{I}$*-middle* $\mathcal{I}$*-common bound-outer lossless-outer w2* **by** *fact*

show $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res real* η $\surd$
and $\mathcal{I}$*-inner* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res ideal* η $\surd$
and $\mathcal{I}$*-real* η, $\mathcal{I}$*-inner* η $\vdash_C$ *sim-outer* η $\odot$ *sim-inner* η $\surd$ **for** η **by**(*rule WT-intro*)+

{ fix $\mathcal{A}$ :: *security* $\Rightarrow$ $('g + 'b, 'h + 'd)$ *distinguisher*
assume *WT* [*WT-intro*]: $\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ $\mathcal{A}$ η $\surd$ **for** η
assume *bound-outer* [*interaction-bound*]: *interaction-bounded-by* (λ*-. True*) ($\mathcal{A}$ η) (*bound-outer* η) **for** η
assume *lossless* [*plossless-intro*]:
lossless-outer $\vee$ *lossless-inner* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{A}$ η) **for** η

let $?\mathcal{A} = \lambda\eta.$ *absorb* ($\mathcal{A}$ η) (*sim-outer* η $|_=$ 1_C)
have $\mathcal{I}$*-middle* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ $?\mathcal{A}$ η $\surd$ **for** η **by**(*rule WT-intro*)+
moreover have *interaction-any-bounded-by* ($?\mathcal{A}$ η) (*bound-inner* η) **for** η
by *interaction-bound-converter*(*rule bound-le*)
moreover have *plossless-gpv* ($\mathcal{I}$*-middle* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($?\mathcal{A}$ η) **if** *lossless-inner* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
ultimately have *negligible* ($\lambda\eta.$ *advantage* ($?\mathcal{A}$ η) (*sim-inner* η $|_=$ 1_C $\rhd$ *ideal* η) (*middle* η))
by(*rule inner.adv*)
also have *negligible* ($\lambda\eta.$ *advantage* ($\mathcal{A}$ η) (*sim-outer* η $|_=$ 1_C $\rhd$ *middle* η) (*real* η))
by(*rule outer.adv*[*OF WT bound-outer lossless*]) *simp*
finally (*negligible-plus*)
show *negligible* ($\lambda\eta.$ *advantage* ($\mathcal{A}$ η) (*sim-outer* $\eta \odot$ *sim-inner* η $|_=$ 1_C $\rhd$ *ideal* η) (*real* η))
apply(*rule negligible-mono*)
apply(*simp add*: *bigo-def*)
apply(*rule exI*[**where** *x=1*])
apply *simp*
apply(*rule always-eventually*)
apply(*clarsimp simp add*: *advantage-def*)
apply(*rule order-trans*)
apply(*rule abs-diff-triangle-ineq2*)
apply(*rule add-right-mono*)
apply(*clarsimp simp add*: *advantage-def distinguish-attach*[*symmetric*] *attach-compose*[*symmetric*] *comp-converter-parallel2 comp-converter-id-left*)
done
next
from *inner.correct* **obtain** *cnv-inner*
where *correct-inner*: $\bigwedge\mathcal{D}.$ $[\![$ $\bigwedge\eta.$ $\mathcal{I}$*-inner* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ $\mathcal{D}$ η $\surd$;
$\bigwedge\eta.$ *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound-inner* η);
$\bigwedge\eta.$ *lossless-inner* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-inner* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η) $]\!]$
$\Longrightarrow$ $(\forall\eta.$ *wiring* ($\mathcal{I}$*-inner* η) ($\mathcal{I}$*-middle* η) (*cnv-inner* η) (*w1* η)) $\wedge$
negligible ($\lambda\eta.$ *advantage* ($\mathcal{D}$ η) (*ideal* η) (*cnv-inner* η $|_=$ 1_C $\rhd$ *middle* η))
by *blast*
from *outer.correct* **obtain** *cnv-outer*
where *correct-outer*: $\bigwedge\mathcal{D}.$ $[\![$ $\bigwedge\eta.$ $\mathcal{I}$*-middle* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ $\mathcal{D}$ η $\surd$;

$\bigwedge\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound-outer* η);
$\bigwedge\eta$. *lossless-outer* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-middle* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η) ⟧
$\Longrightarrow$ ($\forall\,\eta$. *wiring* ($\mathcal{I}$*-middle* η) ($\mathcal{I}$*-real* η) (*cnv-outer* η) (*w2* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*middle* η) (*cnv-outer* η $|_=$ 1_C $\rhd$ *real* η))

by *blast*

show $\exists$ *cnv*. $\forall\,\mathcal{D}$. ($\forall\,\eta$. $\mathcal{I}$*-inner* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash g$ $\mathcal{D}$ η $\surd$) $\longrightarrow$
($\forall\,\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound-outer* η)) $\longrightarrow$
($\forall\,\eta$. *lossless-outer* $\vee$ *lossless-inner* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-inner* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η)) $\longrightarrow$
($\forall\,\eta$. *wiring* ($\mathcal{I}$*-inner* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w1* η $\circ_w$ *w2* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal* η) (*cnv* η $|_=$ 1_C $\rhd$ *real* η))

proof(*intro exI strip conjI*)

fix $\mathcal{D}$ :: *security* $\Rightarrow$ ('*e* + '*b*, '*f* + '*d*) *distinguisher*

assume *WT-D* [*rule-format*, *WT-intro*]: $\forall\,\eta$. $\mathcal{I}$*-inner* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash g$ $\mathcal{D}$ η $\surd$

and *bound* [*rule-format*, *interaction-bound*]: $\forall\,\eta$. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound-outer* η)

and *lossless* [*rule-format*]: $\forall\,\eta$. *lossless-outer* $\vee$ *lossless-inner* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-inner* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η)

let *?cnv* = $\lambda\eta$. *cnv-inner* η $\odot$ *cnv-outer* η

have *bound'*: *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound-inner* η) **for** η **using** *bound*[*of* η] *bound-le*[*of* η]

by(*clarsimp elim*!: *interaction-bounded-by-mono order-trans*[*rotated*] *simp add*: *max-def*)
(*metis* (*full-types*) *linorder-linear more-arith-simps*(*6*) *mult-left-mono zero-le*)

from *correct-inner*[*OF WT-D bound' lossless*]

have *w1*: $\bigwedge\eta$. *wiring* ($\mathcal{I}$*-inner* η) ($\mathcal{I}$*-middle* η) (*cnv-inner* η) (*w1* η)

and *adv1*: *negligible* ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal* η) (*cnv-inner* η $|_=$ 1_C $\rhd$ *middle* η))

by *auto*

obtain *f g* **where** *WT-inner* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$*-inner* η, $\mathcal{I}$*-middle* η $\vdash_C$ *cnv-inner* η $\surd$

and *fg* [*simp*]: $\bigwedge\eta$. *w1* η = (*f* η, *g* η)

and *eq1*: $\bigwedge\eta$. $\mathcal{I}$*-inner* η, $\mathcal{I}$*-middle* η $\vdash_C$ *cnv-inner* η $\sim$ *map-converter id id* (*f* η) (*g* η) 1_C

using *w1*

apply(*atomize-elim*)

apply(*fold all-conj-distrib*)

apply(*subst choice-iff*[*symmetric*])+

apply(*fastforce elim*!: *wiring.cases*)

done

from *w1* **have** [*WT-intro*]: $\mathcal{I}$*-inner* η, $\mathcal{I}$*-middle* η $\vdash_C$ *cnv-inner* η $\surd$ **for** η

by *cases*

let *?𝒟 = λη. absorb (𝒟 η) (map-converter id id (f η) (g η) 1_C |= 1_C)*
have *ℐ: ℐ-inner η ≤ map-ℐ (f η) (g η) (ℐ-middle η)* **for** *η*
using *eq-ℐ-converterD-WT1[OF eq1 WT-inner, of η]* **by**(*rule WT-map-converter-idD*)
with *WT-converter-id* **have** [*WT-intro*]: *ℐ-inner η, map-ℐ (f η) (g η) (ℐ-middle η) ⊢_C 1_C √*
for *η* **by**(*rule WT-converter-mono*) *simp*

have *WT-D': ℐ-middle η ⊕_ℐ ℐ-common η ⊢g ?𝒟 η √* **for** *η* **by**(*rule WT-intro | simp*)+
have *bound': interaction-any-bounded-by (?𝒟 η) (bound-outer η)* **for** *η*
by(*subst map-converter-id-move-left*)(*interaction-bound*; *simp*)
have [*simp*]: *plossless-converter (ℐ-inner η) (map-ℐ (f η) (g η) (ℐ-middle η)) 1_C* **for** *η*
using *plossless-id-converter - ℐ[of η]* **by**(*rule plossless-converter-mono*) *auto*
from *lossless*
have *plossless-gpv (ℐ-middle η ⊕_ℐ ℐ-common η) (?𝒟 η)* **if** *lossless-outer* **for** *η*
by(*rule plossless-gpv-absorb*)(*auto simp add*: *that intro*!: *WT-intro plossless-parallel-converter2 plossless-map-converter*)
from *correct-outer[OF WT-D' bound' this]*
have *w2*: *⋀η. wiring (ℐ-middle η) (ℐ-real η) (cnv-outer η) (w2 η)*
and *adv2*: *negligible (λη. advantage (?𝒟 η) (middle η) (cnv-outer η |= 1_C ▷ real η))*
by *auto*
from *w2* **have** [*WT-intro*]: *ℐ-middle η, ℐ-real η ⊢_C cnv-outer η √* **for** *η* **by** *cases*

show *wiring (ℐ-inner η) (ℐ-real η) (?cnv η) (w1 η ∘_w w2 η)* **for** *η*
using *w1 w2* **by**(*rule wiring-comp-converterI*)

have *eq1'*: *connect (𝒟 η) (cnv-inner η |= 1_C ▷ middle η) = connect (?𝒟 η) (middle η)* **for** *η*
unfolding *distinguish-attach[symmetric]*
by(*rule connect-eq-resource-cong WT-intro eq-ℐ-attach-on' parallel-converter2-eq-ℐ-cong eq1 eq-ℐ-converter-reflI order-refl*)+
have *eq2'*: *connect (?𝒟 η) (cnv-outer η |= 1_C ▷ real η) = connect (𝒟 η) (?cnv η |= 1_C ⊙ 1_C ▷ real η)* **for** *η*
unfolding *distinguish-attach[symmetric] attach-compose comp-converter-parallel2[symmetric]*
by(*rule connect-eq-resource-cong WT-intro eq-ℐ-attach-on' parallel-converter2-eq-ℐ-cong eq1[symmetric] eq-ℐ-converter-reflI order-refl|simp*)+

show *negligible (λη. advantage (𝒟 η) (ideal η) (?cnv η |= 1_C ▷ real η))*
using *negligible-plus[OF adv1 adv2]* **unfolding** *advantage-def eq1' eq2' comp-converter-id-right*
by(*rule negligible-le*) *simp*
qed
}

qed

theorem (**in** *constructive-security*) *lifting*:
assumes *WT-conv* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$-*common'* η, $\mathcal{I}$-*common* $\eta \vdash_C$ *conv* η $\surd$
and *bound* [*interaction-bound*]: $\bigwedge\eta$. *interaction-any-bounded-converter* (*conv* η) (*bound-conv* η)
and *bound-le*: $\bigwedge\eta$. *bound'* η $*$ *max* (*bound-conv* η) *1* $\leq$ *bound* η
and *lossless* [*plossless-intro*]: $\bigwedge\eta$. *lossless* $\Longrightarrow$ *plossless-converter* ($\mathcal{I}$-*common'* η) ($\mathcal{I}$-*common* η) (*conv* η)
shows *constructive-security*
($\lambda\eta$. 1_C $\mid_=$ *conv* η $\rhd$ *real-resource* η) ($\lambda\eta$. 1_C $\mid_=$ *conv* η $\rhd$ *ideal-resource* η) *sim*
$\mathcal{I}$-*real* $\mathcal{I}$-*ideal* $\mathcal{I}$-*common'* *bound'* *lossless* *w*
proof
fix $\mathcal{A}$ :: *security* $\Rightarrow$ ('*a* + '*g*, '*b* + '*h*) *distinguisher*
assume *WT-*$\mathcal{A}$ [*WT-intro*]: $\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* $\eta \vdash g$ $\mathcal{A}$ η $\surd$ **for** η
assume *bound-*$\mathcal{A}$ [*interaction-bound*]: *interaction-any-bounded-by* ($\mathcal{A}$ η) (*bound'* η) **for** η
assume *lossless-*$\mathcal{A}$ [*plossless-intro*]: *lossless* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* η) ($\mathcal{A}$ η) **for** η

let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{A}$ η) (1_C $\mid_=$ *conv* η)

have *ideal*: *connect* ($\mathcal{A}$ η) (*sim* η $\mid_=$ 1_C $\rhd$ 1_C $\mid_=$ *conv* η $\rhd$ *ideal-resource* η) = *connect* (*?*$\mathcal{A}$ η) (*sim* η $\mid_=$ 1_C $\rhd$ *ideal-resource* η) **for** η
by(*simp add*: *distinguish-attach*[*symmetric*] *attach-compose*[*symmetric*] *comp-converter-parallel2* *comp-converter-id-left* *comp-converter-id-right*)
have *real*: *connect* ($\mathcal{A}$ η) (1_C $\mid_=$ *conv* η $\rhd$ *real-resource* η) = *connect* (*?*$\mathcal{A}$ η) (*real-resource* η) **for** η
by(*simp add*: *distinguish-attach*[*symmetric*])
have $\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* $\eta \vdash g$ *?*$\mathcal{A}$ η $\surd$ **for** η **by**(*rule WT-intro*)+
moreover have *interaction-any-bounded-by* (*?*$\mathcal{A}$ η) (*bound* η) **for** η
by *interaction-bound-converter*(*use bound-le*[*of* η] **in** ‹*simp add*: *max.commute*›)
moreover have *plossless-gpv* ($\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η) (*absorb* ($\mathcal{A}$ η) (1_C $\mid_=$ *conv* η)) **if** *lossless* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
ultimately show *negligible* ($\lambda\eta$. *advantage* ($\mathcal{A}$ η) (*sim* η $\mid_=$ 1_C $\rhd$ 1_C $\mid_=$ *conv* η $\rhd$ *ideal-resource* η) (1_C $\mid_=$ *conv* η $\rhd$ *real-resource* η))
unfolding *advantage-def ideal real* **by**(*rule adv*[*unfolded advantage-def*])
next
from *correct* **obtain** *cnv*
where *correct'*: $\bigwedge\mathcal{D}$. $[\![$ $\bigwedge\eta$. $\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* $\eta \vdash g$ $\mathcal{D}$ η $\surd$;
$\bigwedge\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound* η);
$\bigwedge\eta$. *lossless* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η) ($\mathcal{D}$ η) $]\!]$
$\Longrightarrow$ ($\forall\eta$. *wiring* ($\mathcal{I}$-*ideal* η) ($\mathcal{I}$-*real* η) (*cnv* η) (*w* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $\mid_=$ 1_C $\rhd$ *real-resource* η))
by *blast*
show $\exists$ *cnv*. $\forall\mathcal{D}$. ($\forall\eta$. $\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* $\eta \vdash g$ $\mathcal{D}$ η $\surd$) $\longrightarrow$

$(\forall \eta.\ interaction\text{-}any\text{-}bounded\text{-}by\ (\mathcal{D}\ \eta)\ (bound'\ \eta)) \longrightarrow$
$(\forall \eta.\ lossless \longrightarrow plossless\text{-}gpv\ (\mathcal{I}\text{-}ideal\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common'\ \eta)\ (\mathcal{D}\ \eta)) \longrightarrow$
$(\forall \eta.\ wiring\ (\mathcal{I}\text{-}ideal\ \eta)\ (\mathcal{I}\text{-}real\ \eta)\ (cnv\ \eta)\ (w\ \eta)) \wedge$
$negligible\ (\lambda\eta.\ advantage\ (\mathcal{D}\ \eta)\ (1_C \mid\!=\ conv\ \eta \rhd ideal\text{-}resource\ \eta)\ (cnv\ \eta \mid\!=\ 1_C \rhd 1_C \mid\!=\ conv\ \eta \rhd real\text{-}resource\ \eta))$

proof(*intro exI conjI strip*)
fix $\mathcal{D}$ :: *security* $\Rightarrow$ ('*c* + '*g*, '*d* + '*h*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall \eta.\ \mathcal{I}\text{-}ideal\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common'\ \eta \vdash g\ \mathcal{D}\ \eta\ \surd$
and *bound* [*rule-format*, *interaction-bound*]: $\forall \eta.$ *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}\ \eta$) (*bound'* η)
and *lossless* [*rule-format*]: $\forall \eta.\ lossless \longrightarrow plossless\text{-}gpv\ (\mathcal{I}\text{-}ideal\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common'\ \eta)\ (\mathcal{D}\ \eta)$

let $?\mathcal{D} = \lambda\eta.\ absorb\ (\mathcal{D}\ \eta)\ (1_C \mid\!=\ conv\ \eta)$
have *WT-D'* [*WT-intro*]: $\mathcal{I}\text{-}ideal\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta \vdash g\ ?\mathcal{D}\ \eta\ \surd$ **for** η **by**(*rule WT-intro*)+
have *bound'*: *interaction-any-bounded-by* ($?\mathcal{D}\ \eta$) (*bound* η) **for** η
by *interaction-bound*(*use bound-le*[*of* η] **in** ‹*auto simp add*: *max-def split*: *if-split-asm*›)
have *plossless-gpv* ($\mathcal{I}\text{-}ideal\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta$) ($?\mathcal{D}\ \eta$) **if** *lossless* **for** η
by(*rule lossless that WT-intro plossless-intro*)+
from *correct'*[*OF WT-D' bound' this*]
have *w1*: *wiring* ($\mathcal{I}\text{-}ideal\ \eta$) ($\mathcal{I}\text{-}real\ \eta$) ($cnv\ \eta$) ($w\ \eta$)
and *adv'*: *negligible* ($\lambda\eta.$ *advantage* ($?\mathcal{D}\ \eta$) (*ideal-resource* η) ($cnv\ \eta \mid\!=\ 1_C \rhd real\text{-}resource\ \eta$)) **for** η
by *auto*
show *wiring* ($\mathcal{I}\text{-}ideal\ \eta$) ($\mathcal{I}\text{-}real\ \eta$) ($cnv\ \eta$) ($w\ \eta$) **for** η **by**(*rule w1*)
have $cnv\ \eta \mid\!=\ 1_C \rhd 1_C \mid\!=\ conv\ \eta \rhd real\text{-}resource\ \eta = 1_C \mid\!=\ conv\ \eta \rhd cnv\ \eta \mid\!=\ 1_C \rhd real\text{-}resource\ \eta$ **for** η
by(*simp add*: *attach-compose*[*symmetric*] *comp-converter-parallel2 comp-converter-id-left comp-converter-id-right*)
with *adv'*
show *negligible* ($\lambda\eta.$ *advantage* ($\mathcal{D}\ \eta$) ($1_C \mid\!=\ conv\ \eta \rhd ideal\text{-}resource\ \eta$) ($cnv\ \eta \mid\!=\ 1_C \rhd 1_C \mid\!=\ conv\ \eta \rhd real\text{-}resource\ \eta$))
by(*simp add*: *advantage-def distinguish-attach*[*symmetric*])
qed
qed(*rule WT-intro*)+

theorem *constructive-security-trivial*:
fixes *res*
assumes [*WT-intro*]: $\bigwedge\eta.\ \mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta \vdash res\ res\ \eta\ \surd$
shows *constructive-security res res* (λ-. 1_C) $\mathcal{I}$ $\mathcal{I}$ $\mathcal{I}$*-common bound lossless* (λ-. (*id*, *id*))
proof
show $\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta \vdash res\ res\ \eta\ \surd$ **and** $\mathcal{I}\ \eta, \mathcal{I}\ \eta \vdash_C 1_C\ \surd$ **for** η **by**(*rule WT-intro*)+

fix $\mathcal{A}$ :: *security* $\Rightarrow$ ('*a* + '*b*, '*c* + '*d*) *distinguisher*

assume *WT* [*WT-intro*]: $\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}\ \eta \vdash g\ \mathcal{A}\ \eta\ \surd$ **for** η
have *connect* $(\mathcal{A}\ \eta)\ (1_C \mid\!=\ 1_C \rhd res\ \eta) = connect\ (\mathcal{A}\ \eta)\ (1_C \rhd res\ \eta)$ **for** η
by(*rule connect-eq-resource-cong*[*OF WT*])(*fastforce intro*: *WT-intro eq-*$\mathcal{I}$*-attach-on*′ *parallel-converter2-id-id*)+
then show *negligible* $(\lambda\eta.\ advantage\ (\mathcal{A}\ \eta)\ (1_C \mid\!=\ 1_C \rhd res\ \eta)\ (res\ \eta))$
unfolding *advantage-def* **by** *simp*
next
show $\exists\, cnv.\ \forall \mathcal{D}.\ (\forall\, \eta.\ \mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}\ \eta \vdash g\ \mathcal{D}\ \eta\ \surd) \longrightarrow$
$(\forall\, \eta.\ interaction\text{-}any\text{-}bounded\text{-}by\ (\mathcal{D}\ \eta)\ (bound\ \eta)) \longrightarrow$
$(\forall\, \eta.\ lossless \longrightarrow plossless\text{-}gpv\ (\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}\ \eta)\ (\mathcal{D}\ \eta)) \longrightarrow$
$(\forall\, \eta.\ wiring\ (\mathcal{I}\ \eta)\ (\mathcal{I}\ \eta)\ (cnv\ \eta)\ (id, id)) \wedge$
$negligible\ (\lambda\eta.\ advantage\ (\mathcal{D}\ \eta)\ (res\ \eta)\ (cnv\ \eta \mid\!=\ 1_C \rhd res\ \eta))$
proof(*intro exI strip conjI*)
fix $\mathcal{D}$ :: *security* ⇒ (′*a* + ′*b*, ′*c* + ′*d*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall\, \eta.\ \mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}\ \eta \vdash g\ \mathcal{D}\ \eta\ \surd$
and *bound* [*rule-format*, *interaction-bound*]: $\forall\, \eta.$ *interaction-bounded-by* (λ-. *True*) $(\mathcal{D}\ \eta)$ $(bound\ \eta)$
and *lossless* [*rule-format*]: $\forall\, \eta.\ lossless \longrightarrow plossless\text{-}gpv\ (\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}\ \eta)\ (\mathcal{D}\ \eta)$
show *wiring* $(\mathcal{I}\ \eta)\ (\mathcal{I}\ \eta)\ 1_C\ (id, id)$ **for** η **by** *simp*
have *connect* $(\mathcal{D}\ \eta)\ (1_C \mid\!=\ 1_C \rhd res\ \eta) = connect\ (\mathcal{D}\ \eta)\ (1_C \rhd res\ \eta)$ **for** η
by(*rule connect-eq-resource-cong*)(*rule WT-intro eq-*$\mathcal{I}$*-attach-on*′ *parallel-converter2-id-id order-refl*)+
then show *negligible* $(\lambda\eta.\ advantage\ (\mathcal{D}\ \eta)\ (res\ \eta)\ (1_C \mid\!=\ 1_C \rhd res\ \eta))$
by(*auto simp add*: *advantage-def*)
qed
qed

theorem *parallel-constructive-security*:
assumes *constructive-security real1 ideal1 sim1* $\mathcal{I}$*-real1* $\mathcal{I}$*-inner1* $\mathcal{I}$*-common1 bound1 lossless1 w1*
assumes *constructive-security real2 ideal2 sim2* $\mathcal{I}$*-real2* $\mathcal{I}$*-inner2* $\mathcal{I}$*-common2 bound2 lossless2 w2*

and *lossless-real1* [*plossless-intro*]: $\bigwedge\eta.\ lossless2 \Longrightarrow lossless\text{-}resource\ (\mathcal{I}\text{-}real1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common1\ \eta)\ (real1\ \eta)$
and *lossless-sim2* [*plossless-intro*]: $\bigwedge\eta.\ lossless1 \Longrightarrow plossless\text{-}converter\ (\mathcal{I}\text{-}real2\ \eta)\ (\mathcal{I}\text{-}inner2\ \eta)\ (sim2\ \eta)$
and *lossless-ideal2* [*plossless-intro*]: $\bigwedge\eta.\ lossless1 \Longrightarrow lossless\text{-}resource\ (\mathcal{I}\text{-}inner2\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common2\ \eta)\ (ideal2\ \eta)$
shows *constructive-security* $(\lambda\eta.\ parallel\text{-}wiring \rhd real1\ \eta \parallel real2\ \eta)$ $(\lambda\eta.\ parallel\text{-}wiring \rhd ideal1\ \eta \parallel ideal2\ \eta)$ $(\lambda\eta.\ sim1\ \eta \mid\!=\ sim2\ \eta)$
$(\lambda\eta.\ \mathcal{I}\text{-}real1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}real2\ \eta)$ $(\lambda\eta.\ \mathcal{I}\text{-}inner1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}inner2\ \eta)$ $(\lambda\eta.\ \mathcal{I}\text{-}common1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common2\ \eta)$
$(\lambda\eta.\ min\ (bound1\ \eta)\ (bound2\ \eta))\ (lossless1 \vee lossless2)\ (\lambda\eta.\ w1\ \eta \mid_w w2\ \eta)$
proof
interpret *sec1*: *constructive-security real1 ideal1 sim1* $\mathcal{I}$*-real1* $\mathcal{I}$*-inner1* $\mathcal{I}$*-common1 bound1 lossless1 w1* **by** *fact*
interpret *sec2*: *constructive-security real2 ideal2 sim2* $\mathcal{I}$*-real2* $\mathcal{I}$*-inner2* $\mathcal{I}$*-common2*

bound2 lossless2 w2 **by** *fact*

show (*$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η) $\vdash_{res}$ *parallel-wiring* $\rhd$ *real1* η $\parallel$ *real2* η $\surd$
and (*$\mathcal{I}$-inner1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-inner2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η) $\vdash_{res}$ *parallel-wiring* $\rhd$ *ideal1* η $\parallel$ *ideal2* η $\surd$
and *$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η, *$\mathcal{I}$-inner1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-inner2* η $\vdash_C$ *sim1* η $\mid_=$ *sim2* η $\surd$ **for** η **by**(*rule WT-intro*)+

fix $\mathcal{A}$ :: *security* $\Rightarrow$ (('*a* + '*g*) + '*b* + '*h*, ('*c* + '*i*) + '*d* + '*j*) *distinguisher*
assume *WT* [*WT-intro*]: (*$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η) $\vdash_g$ $\mathcal{A}$ η $\surd$ **for** η
assume *bound* [*interaction-bound*]: *interaction-any-bounded-by* ($\mathcal{A}$ η) (*min* (*bound1* η) (*bound2* η)) **for** η
assume *lossless* [*plossless-intro*]: *lossless1* $\vee$ *lossless2* $\Longrightarrow$ *plossless-gpv* ((*$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η)) ($\mathcal{A}$ η) **for** η

let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*parallel-wiring* $\odot$ *parallel-converter* (*converter-of-resource* (*real1* η)) 1_C)

have $*$:*$\mathcal{I}$-uniform* (*outs-$\mathcal{I}$* ((*$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η)))
UNIV,(*$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common1* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-inner2* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η) $\vdash_C$
((1_C $\mid_=$ *sim2* η) $\mid_=$ 1_C) $\odot$ *parallel-wiring* $\sim$ ((1_C $\mid_=$ *sim2* η) $\mid_=$ 1_C $\mid_=$ 1_C) $\odot$ *parallel-wiring* **for** η
by(*rule eq-$\mathcal{I}$-comp-cong*, *rule eq-$\mathcal{I}$-converter-mono*)
(*auto simp add*: *le-$\mathcal{I}$-def intro*: *parallel-converter2-eq-$\mathcal{I}$-cong eq-$\mathcal{I}$-converter-reflI WT-converter-parallel-converter2*
WT-converter-id sec2.*WT-sim parallel-converter2-id-id*[*symmetric*] *eq-$\mathcal{I}$-converter-reflI WT-parallel-wiring*)

have $**$: *outs-$\mathcal{I}$* ((*$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η)) $\vdash_R$
((1_C $\mid_=$ *sim2* η) $\mid_=$ 1_C $\mid_=$ 1_C) $\odot$ *parallel-wiring* $\rhd$ *real1* η $\parallel$ *ideal2* η $\sim$
parallel-wiring $\odot$ (*converter-of-resource* (*real1* η) $\mid_\propto$ 1_C) $\rhd$ *sim2* η $\mid_=$ 1_C $\rhd$ *ideal2* η **for** η
unfolding *comp-parallel-wiring*
by(*rule eq-resource-on-trans*, *rule eq-$\mathcal{I}$-attach-on*[**where** *conv'*=*parallel-wiring* $\odot$ (1_C $\mid_=$ *sim2* η $\mid_=$ 1_C)]
, (*rule WT-intro*)+, *rule eq-$\mathcal{I}$-comp-cong*, *rule eq-$\mathcal{I}$-converter-mono*)
(*auto simp add*: *le-$\mathcal{I}$-def attach-compose attach-parallel2 attach-converter-of-resource-conv-parallel-resource intro*: *WT-intro parallel-converter2-eq-$\mathcal{I}$-cong parallel-converter2-id-id eq-$\mathcal{I}$-converter-reflI*)

have *ideal2*:
connect ($\mathcal{A}$ η) ((1_C $\mid_=$ *sim2* η) $\mid_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\parallel$ *ideal2* η)
=
connect (*?*$\mathcal{A}$ η) (*sim2* η $\mid_=$ 1_C $\rhd$ *ideal2* η) **for** η
unfolding *distinguish-attach*[*symmetric*]
proof (*rule connect-eq-resource-cong*[*OF WT*, *rotated*], *goal-cases*)

case *2*
then show *?case*
by(*subst attach-compose*[*symmetric*], *rule eq-resource-on-trans*
, *rule eq-ℐ-attach-on*[**where** *conv′*=((*1*$_C$ |= *sim2* *η*) |= *1*$_C$ |= *1*$_C$) ⊙ *parallel-wiring*])
((*rule WT-intro*)+ | *intro* ∗ | *intro* ∗∗)+
qed (*rule WT-intro*)+
have *real2*: *connect* (*𝒜* *η*) (*parallel-wiring* ▷ *real1* *η* ∥ *real2* *η*) = *connect* (*?𝒜* *η*) (*real2* *η*) **for** *η*
unfolding *distinguish-attach*[*symmetric*]
by(*simp add*: *attach-compose attach-converter-of-resource-conv-parallel-resource*)
have *ℐ-real2* *η* ⊕$_ℐ$ *ℐ-common2* *η* ⊢*g* *?𝒜* *η* √ **for** *η* **by**(*rule WT-intro*)+
moreover have *interaction-any-bounded-by* (*?𝒜* *η*) (*bound2* *η*) **for** *η*
by *interaction-bound-converter simp*
moreover have *plossless-gpv* (*ℐ-real2* *η* ⊕$_ℐ$ *ℐ-common2* *η*) (*?𝒜* *η*) **if** *lossless2* **for** *η*
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
ultimately
have *negl2*: *negligible* (*λη. advantage* (*𝒜* *η*)
((*1*$_C$ |= *sim2* *η*) |= *1*$_C$ ▷ *parallel-wiring* ▷ *real1* *η* ∥ *ideal2* *η*)
(*parallel-wiring* ▷ *real1* *η* ∥ *real2* *η*))
unfolding *advantage-def ideal2 real2* **by**(*rule sec2.adv*[*unfolded advantage-def*])

let *?𝒜* = *λη. absorb* (*𝒜* *η*) (*parallel-wiring* ⊙ *parallel-converter 1*$_C$ (*converter-of-resource* (*sim2* *η* |= *1*$_C$ ▷ *ideal2* *η*)))
have *ideal1*:
connect (*𝒜* *η*) ((*sim1* *η* |= *sim2* *η*) |= *1*$_C$ ▷ *parallel-wiring* ▷ *ideal1* *η* ∥ *ideal2* *η*) =
connect (*?𝒜* *η*) (*sim1* *η* |= *1*$_C$ ▷ *ideal1* *η*) **for** *η*
proof −
have ∗: *ℐ-uniform* ((*outs-ℐ* (*ℐ-real1* *η*) <+> *outs-ℐ* (*ℐ-real2* *η*)) <+> *outs-ℐ* (*ℐ-common1* *η*) <+>
outs-ℐ (*ℐ-common2* *η*)) *UNIV*,(*ℐ-inner1* *η* ⊕$_ℐ$ *ℐ-common1* *η*) ⊕$_ℐ$ (*ℐ-inner2* *η* ⊕$_ℐ$ *ℐ-common2* *η*) ⊢$_C$
((*sim1* *η* |= *sim2* *η*) |= *1*$_C$) ⊙ *parallel-wiring* ∼ ((*sim1* *η* |= *sim2* *η*) |= *1*$_C$ |= *1*$_C$) ⊙ *parallel-wiring*
by(*rule eq-ℐ-comp-cong*, *rule eq-ℐ-converter-mono*)
(*auto simp add*: *le-ℐ-def comp-parallel-wiring′ attach-compose attach-parallel2 attach-converter-of-resource-conv-parallel-resource2*
intro: *WT-intro parallel-converter2-id-id*[*symmetric*] *eq-ℐ-converter-reflI parallel-converter2-eq-ℐ-cong eq-ℐ-converter-mono*)

have ∗∗:((*outs-ℐ* (*ℐ-real1* *η*) <+> *outs-ℐ* (*ℐ-real2* *η*)) <+> *outs-ℐ* (*ℐ-common1* *η*) <+> *outs-ℐ* (*ℐ-common2* *η*)) ⊢$_R$
(*sim1* *η* |= *sim2* *η*) |= *1*$_C$ ▷ *parallel-wiring* ▷ *ideal1* *η* ∥ *ideal2* *η* ∼
parallel-wiring ⊙ (*1*$_C$ |$_\propto$ *converter-of-resource* (*sim2* *η* |= *1*$_C$ ▷ *ideal2* *η*)) ▷ *sim1* *η* |= *1*$_C$ ▷ *ideal1* *η*
unfolding *attach-compose*[*symmetric*]
by(*rule eq-resource-on-trans*, *rule eq-ℐ-attach-on*[**where** *conv′*=((*sim1* *η* |=

sim2 η) $|_=$ 1_C $|_=$ 1_C) $\odot$ *parallel-wiring*])
((*rule WT-intro*)+ | *intro* $*$ | *auto simp add*: *le-$\mathcal{I}$-def comp-parallel-wiring'* *attach-compose*
attach-parallel2 attach-converter-of-resource-conv-parallel-resource2 intro: *WT-intro* $*$)+

show *?thesis*
unfolding *distinguish-attach*[*symmetric*] **using** *WT*
by(*rule connect-eq-resource-cong*) (*simp add*: $**$, (*rule WT-intro*)+)
qed

have *real1*:
connect ($\mathcal{A}$ η) ((1_C $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\parallel$ *ideal2* η)
=
connect (*?*$\mathcal{A}$ η) (*real1* η) **for** η
proof −
have $**$: $\mathcal{I}$*-uniform* (*outs-$\mathcal{I}$* ((*$\mathcal{I}$-real1* η $\oplus_\mathcal{I}$ *$\mathcal{I}$-real2* η) $\oplus_\mathcal{I}$ (*$\mathcal{I}$-common1* η $\oplus_\mathcal{I}$ *$\mathcal{I}$-common2* η)))
UNIV,(*$\mathcal{I}$-real1* η $\oplus_\mathcal{I}$ *$\mathcal{I}$-common1* η) $\oplus_\mathcal{I}$ (*$\mathcal{I}$-inner2* η $\oplus_\mathcal{I}$ *$\mathcal{I}$-common2* η) $\vdash_C$
((1_C $|_=$ *sim2* η) $|_=$ 1_C) $\odot$ *parallel-wiring* $\sim$ ((1_C $|_=$ *sim2* η) $|_=$ 1_C $|_=$ 1_C) $\odot$ *parallel-wiring*
by(*rule eq-$\mathcal{I}$-comp-cong*, *rule eq-$\mathcal{I}$-converter-mono*)
(*auto simp add*: *le-$\mathcal{I}$-def intro*: *WT-intro parallel-converter2-eq-$\mathcal{I}$-cong WT-converter-parallel-converter2*
parallel-converter2-id-id[*symmetric*] *eq-$\mathcal{I}$-converter-reflI WT-parallel-wiring*)

have $*$: *outs-$\mathcal{I}$* ((*$\mathcal{I}$-real1* η $\oplus_\mathcal{I}$ *$\mathcal{I}$-real2* η) $\oplus_\mathcal{I}$ (*$\mathcal{I}$-common1* η $\oplus_\mathcal{I}$ *$\mathcal{I}$-common2* η)) $\vdash_R$
parallel-wiring $\odot$ ((1_C $|_=$ 1_C) $|_=$ *sim2* η $|_=$ 1_C) $\rhd$ *real1* η $\parallel$ *ideal2* η $\sim$
parallel-wiring $\odot$ (1_C $|_\propto$ *converter-of-resource* (*sim2* η $|_=$ 1_C $\rhd$ *ideal2* η)) $\rhd$ *real1* η
by(*rule eq-resource-on-trans*, *rule eq-$\mathcal{I}$-attach-on*[**where** *conv'*=*parallel-wiring* $\odot$ (1_C $|_=$ *sim2* η $|_=$ 1_C)]
, (*rule WT-intro*)+, *rule eq-$\mathcal{I}$-comp-cong*, *rule eq-$\mathcal{I}$-converter-mono*)
(*auto simp add*: *le-$\mathcal{I}$-def attach-compose attach-converter-of-resource-conv-parallel-resource2 attach-parallel2*
intro: *WT-intro parallel-converter2-eq-$\mathcal{I}$-cong parallel-converter2-id-id eq-$\mathcal{I}$-converter-reflI*)

show *?thesis*
unfolding *distinguish-attach*[*symmetric*] **using** *WT*
by(*rule connect-eq-resource-cong*, *fold attach-compose*)
(*rule eq-resource-on-trans*[**where** *res'*=((1_C $|_=$ *sim2* η) $|_=$ 1_C $|_=$ 1_C) $\odot$ *parallel-wiring* $\rhd$ *real1* η $\parallel$ *ideal2* η]
, (*rule eq-$\mathcal{I}$-attach-on*, (*intro* $*$ $**$ | *subst comp-parallel-wiring* | *rule eq-$\mathcal{I}$-attach-on* | (*rule WT-intro eq-$\mathcal{I}$-attach-on*)+)+))
qed

have *$\mathcal{I}$-real1* η $\oplus_\mathcal{I}$ *$\mathcal{I}$-common1* η $\vdash_g$ *?*$\mathcal{A}$ η $\surd$ **for** η **by**(*rule WT-intro*)+

moreover have *interaction-any-bounded-by* (*?*$\mathcal{A}$ η) (*bound1* η) **for** η
by *interaction-bound-converter simp*
moreover have *plossless-gpv* ($\mathcal{I}$*-real1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common1* η) (*?*$\mathcal{A}$ η) **if** *lossless1* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
ultimately
have *negl1*: *negligible* ($\lambda\eta$. *advantage* ($\mathcal{A}$ η)
((*sim1* η $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *ideal1* η $\parallel$ *ideal2* η)
((*1*$_C$ $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\parallel$ *ideal2* η))
unfolding *advantage-def ideal1 real1* **by**(*rule sec1.adv*[*unfolded advantage-def*])

from *negligible-plus*[*OF negl1 negl2*]
show *negligible* ($\lambda\eta$. *advantage* ($\mathcal{A}$ η) ((*sim1* η $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *ideal1* η $\parallel$ *ideal2* η)
(*parallel-wiring* $\rhd$ *real1* η $\parallel$ *real2* η))
by(*rule negligible-mono*) (*auto simp add*: *advantage-def intro*!: *eventuallyI landau-o.big-mono*)
next
interpret *sec1*: *constructive-security real1 ideal1 sim1* $\mathcal{I}$*-real1* $\mathcal{I}$*-inner1* $\mathcal{I}$*-common1 bound1 lossless1 w1* **by** *fact*
interpret *sec2*: *constructive-security real2 ideal2 sim2* $\mathcal{I}$*-real2* $\mathcal{I}$*-inner2* $\mathcal{I}$*-common2 bound2 lossless2 w2* **by** *fact*
from *sec1.correct* **obtain** *cnv1*
where *correct1*: $\bigwedge\mathcal{D}$. $[\![\bigwedge\eta$. $\mathcal{I}$*-inner1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common1* $\eta \vdash g$ $\mathcal{D}$ η $\surd$;
$\bigwedge\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound1* η);
$\bigwedge\eta$. *lossless1* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-inner1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common1* η) ($\mathcal{D}$ η) $]\!]$
$\Longrightarrow$ ($\forall\eta$. *wiring* ($\mathcal{I}$*-inner1* η) ($\mathcal{I}$*-real1* η) (*cnv1* η) (*w1* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal1* η) (*cnv1* η $|_=$ 1_C $\rhd$ *real1* η))
by *blast*
from *sec2.correct* **obtain** *cnv2*
where *correct2*: $\bigwedge\mathcal{D}$. $[\![\bigwedge\eta$. $\mathcal{I}$*-inner2* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* $\eta \vdash g$ $\mathcal{D}$ η $\surd$;
$\bigwedge\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound2* η);
$\bigwedge\eta$. *lossless2* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-inner2* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) ($\mathcal{D}$ η) $]\!]$
$\Longrightarrow$ ($\forall\eta$. *wiring* ($\mathcal{I}$*-inner2* η) ($\mathcal{I}$*-real2* η) (*cnv2* η) (*w2* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal2* η) (*cnv2* η $|_=$ 1_C $\rhd$ *real2* η))
by *blast*
show $\exists$ *cnv*. $\forall\mathcal{D}$. ($\forall\eta$. ($\mathcal{I}$*-inner1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) $\vdash g$ $\mathcal{D}$ η $\surd$) $\longrightarrow$
($\forall\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*min* (*bound1* η) (*bound2* η))) $\longrightarrow$
($\forall\eta$. *lossless1* $\vee$ *lossless2* $\longrightarrow$ *plossless-gpv* (($\mathcal{I}$*-inner1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η)) ($\mathcal{D}$ η)) $\longrightarrow$
($\forall\eta$. *wiring* ($\mathcal{I}$*-inner1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) ($\mathcal{I}$*-real1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) (*cnv* η) (*w1* η $|_w$ *w2* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*parallel-wiring* $\rhd$ *ideal1* η $\parallel$ *ideal2* η) (*cnv* η $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\parallel$ *real2* η))
proof(*intro exI strip conjI*)
fix $\mathcal{D}$ :: *security* $\Rightarrow$ (('*e* + '*k*) + '*b* + '*h*, ('*f* + '*l*) + '*d* + '*j*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall\eta$. ($\mathcal{I}$*-inner1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) $\vdash g$ $\mathcal{D}$ η $\surd$

and *bound* [*rule-format*, *interaction-bound*]: $\forall \eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*min* (*bound1* η) (*bound2* η))
and *lossless* [*rule-format*, *plossless-intro*]: $\forall \eta$. *lossless1* $\vee$ *lossless2* $\longrightarrow$ *plossless-gpv* ((*$\mathcal{I}$-inner1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-inner2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η)) ($\mathcal{D}$ η)

let *?cnv* = $\lambda\eta$. *cnv1* η $|_=$ *cnv2* η

let *?$\mathcal{D}$1* = $\lambda\eta$. *absorb* ($\mathcal{D}$ η) (*parallel-wiring* $\odot$ *parallel-converter* 1_C (*converter-of-resource* (*ideal2* η)))
have *WT1*: *$\mathcal{I}$-inner1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common1* $\eta \vdash g$ *?$\mathcal{D}$1* η $\surd$ **for** η **by**(*rule WT-intro*)+
have *bound1*: *interaction-any-bounded-by* (*?$\mathcal{D}$1* η) (*bound1* η) **for** η **by** *interaction-bound simp*
have *plossless-gpv* (*$\mathcal{I}$-inner1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common1* η) (*?$\mathcal{D}$1* η) **if** *lossless1* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
from *correct1*[*OF WT1 bound1 this*]
have *w1*: *wiring* (*$\mathcal{I}$-inner1* η) (*$\mathcal{I}$-real1* η) (*cnv1* η) (*w1* η)
and *adv1*: *negligible* ($\lambda\eta$. *advantage* (*?$\mathcal{D}$1* η) (*ideal1* η) (*cnv1* η $|_=$ 1_C $\rhd$ *real1* η)) **for** η
by *auto*

from *w1* **obtain** *f g* **where** *fg*: $\bigwedge\eta$. *w1* η = (*f* η, *g* η)
and [*WT-intro*]: $\bigwedge\eta$. *$\mathcal{I}$-inner1* η, *$\mathcal{I}$-real1* $\eta \vdash_C$ *cnv1* η $\surd$
and *eq1*: $\bigwedge\eta$. *$\mathcal{I}$-inner1* η, *$\mathcal{I}$-real1* $\eta \vdash_C$ *cnv1* $\eta \sim$ *map-converter id id* (*f* η) (*g* η) 1_C
apply *atomize-elim*
apply(*fold all-conj-distrib*)
apply(*subst choice-iff*[*symmetric*])+
apply(*fastforce elim*!: *wiring.cases*)
done
have *$\mathcal{I}$1*: *$\mathcal{I}$-inner1* $\eta \leq$ *map-$\mathcal{I}$* (*f* η) (*g* η) (*$\mathcal{I}$-real1* η) **for** η
using *eq-$\mathcal{I}$-converterD-WT1*[*OF eq1*] **by**(*rule WT-map-converter-idD*)(*rule WT-intro*)
with *WT-converter-id order-refl* **have** [*WT-intro*]: *$\mathcal{I}$-inner1* η, *map-$\mathcal{I}$* (*f* η) (*g* η) (*$\mathcal{I}$-real1* η) $\vdash_C$ 1_C $\surd$ **for** η
by(*rule WT-converter-mono*)
have *lossless1* [*plossless-intro*]: *plossless-converter* (*$\mathcal{I}$-inner1* η) (*$\mathcal{I}$-real1* η) (*map-converter id id* (*f* η) (*g* η) 1_C) **for** η
by(*rule plossless-map-converter*)(*rule plossless-intro order-refl $\mathcal{I}$1 WT-intro plossless-converter-mono* | *simp*)+

let *?$\mathcal{D}$2* = $\lambda\eta$. *absorb* ($\mathcal{D}$ η) (*parallel-wiring* $\odot$ *parallel-converter* (*converter-of-resource* (*map-converter id id* (*f* η) (*g* η) 1_C $|_=$ 1_C $\rhd$ *real1* η)) 1_C)
have *WT2*: *$\mathcal{I}$-inner2* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* $\eta \vdash g$ *?$\mathcal{D}$2* η $\surd$ **for** η **by**(*rule WT-intro* | *simp*)+
have *bound2*: *interaction-any-bounded-by* (*?$\mathcal{D}$2* η) (*bound2* η) **for** η **by** *interaction-bound simp*
have *plossless-gpv* (*$\mathcal{I}$-inner2* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η) (*?$\mathcal{D}$2* η) **if** *lossless2* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+

from *correct2*[*OF WT2 bound2 this*]
have *w2*: *wiring* (*I-inner2* η) (*I-real2* η) (*cnv2* η) (*w2* η)
and *adv2*: *negligible* ($\lambda\eta$. *advantage* (*?D2* η) (*ideal2* η) (*cnv2* η $|_=$ 1_C $\rhd$ *real2* η)) **for** η
by *auto*

from *w2* **have** [*WT-intro*]: *I-inner2* η, *I-real2* η $\vdash_C$ *cnv2* η $\surd$ **for** η **by** *cases*

have $*$: *connect* (*D* η) (*?cnv* η $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\|$ *real2* η) =
connect (*?D2* η) (*cnv2* η $|_=$ 1_C $\rhd$ *real2* η) **for** η
proof −
have *outs-I* ((*I-inner1* η $\oplus_I$ *I-inner2* η) $\oplus_I$ (*I-common1* η $\oplus_I$ *I-common2* η)) $\vdash_R$
?cnv η $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\|$ *real2* η $\sim$
(*map-converter id id* (*f* η) (*g* η) 1_C $|_=$ *cnv2* η) $|_=$ (1_C $|_=$ 1_C) $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\|$ *real2* η
by(*rule eq-I-attach-on′ WT-intro parallel-converter2-eq-I-cong eq1 eq-I-converter-reflI parallel-converter2-id-id*[*symmetric*])+ *simp*
also have (*map-converter id id* (*f* η) (*g* η) 1_C $|_=$ *cnv2* η) $|_=$ (1_C $|_=$ 1_C) $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\|$ *real2* η =
parallel-wiring $\rhd$ (*map-converter id id* (*f* η) (*g* η) 1_C $|_=$ 1_C $\rhd$ *real1* η) $\|$ (*cnv2* η $|_=$ 1_C $\rhd$ *real2* η)
by(*simp add*: *comp-parallel-wiring′ attach-compose attach-parallel2*)
finally show *?thesis*
by(*auto intro*!: *connect-eq-resource-cong*[*OF WT-D*] *intro*: *WT-intro simp add*: *distinguish-attach*[*symmetric*] *attach-compose attach-converter-of-resource-conv-parallel-resource*)
qed

have $**$: *connect* (*?D2* η) (*ideal2* η) = *connect* (*?D1* η) (*cnv1* η $|_=$ 1_C $\rhd$ *real1* η) **for** η
proof −
have *connect* (*?D2* η) (*ideal2* η) =
connect (*D* η) (*parallel-wiring* $\rhd$ ((*map-converter id id* (*f* η) (*g* η) 1_C $|_=$ 1_C) $|_=$ 1_C) $\rhd$ (*real1* η $\|$ *ideal2* η))
by(*simp add*: *distinguish-attach*[*symmetric*] *attach-converter-of-resource-conv-parallel-resource attach-compose attach-parallel2*)
also have ... = *connect* (*D* η) (*parallel-wiring* $\rhd$ ((*cnv1* η $|_=$ 1_C) $|_=$ 1_C) $\rhd$ (*real1* η $\|$ *ideal2* η))
unfolding *attach-compose*[*symmetric*] **using** *WT-D*
by(*rule connect-eq-resource-cong*[*symmetric*])
(*rule eq-I-attach-on′ WT-intro eq-I-comp-cong eq-I-converter-reflI parallel-converter2-eq-I-cong eq1* | *simp*)+
also have ... = *connect* (*?D1* η) (*cnv1* η $|_=$ 1_C $\rhd$ *real1* η)
by(*simp add*: *distinguish-attach*[*symmetric*] *attach-converter-of-resource-conv-parallel-resource2 attach-compose attach-parallel2*)
finally show *?thesis* .
qed

have $***$: *connect* (*?D1* η) (*ideal1* η) = *connect* (*D* η) (*parallel-wiring* $\rhd$ *ideal1*

$\eta \parallel$ *ideal2* η) **for** η
by(*auto intro*!: *connect-eq-resource-cong*[*OF WT-D*] *simp add*: *attach-converter-of-resource-conv-parallel-re distinguish-attach*[*symmetric*] *attach-compose intro*: *WT-intro*)

show *wiring* ($\mathcal{I}$-*inner1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*inner2* η) ($\mathcal{I}$-*real1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*real2* η) (*?cnv* η) (*w1* $\eta \mid_w$ *w2* η) **for** η
using *w1 w2* **by**(*rule wiring-parallel-converter2*)
from *negligible-plus*[*OF adv1 adv2*]
show *negligible* ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*parallel-wiring* $\triangleright$ *ideal1* $\eta \parallel$ *ideal2* η) (*?cnv* $\eta \mid_=$ 1_C $\triangleright$ *parallel-wiring* $\triangleright$ *real1* $\eta \parallel$ *real2* η))
by(*rule negligible-le*)(*simp add*: *advantage-def* $*$ $**$ $***$)
qed
qed

theorem (**in** *constructive-security*) *parallel-realisation1*:
assumes *WT-res*: $\bigwedge\eta$. $\mathcal{I}$-*res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* $\eta \vdash$*res res* η $\surd$
and *lossless-res*: $\bigwedge\eta$. *lossless* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$-*res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* η) (*res* η)
shows *constructive-security* ($\lambda\eta$. *parallel-wiring* $\triangleright$ *res* $\eta \parallel$ *real-resource* η) ($\lambda\eta$. *parallel-wiring* $\triangleright$ (*res* $\eta \parallel$ *ideal-resource* η)) ($\lambda\eta$. *parallel-converter2 id-converter* (*sim* η))
($\lambda\eta$. $\mathcal{I}$-*res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*real* η) ($\lambda\eta$. $\mathcal{I}$-*res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*ideal* η) ($\lambda\eta$. $\mathcal{I}$-*common'* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η) *bound lossless* ($\lambda\eta$. (*id*, *id*) $\mid_w$ *w* η)
by(*rule parallel-constructive-security*[*OF constructive-security-trivial*[**where** *lossless*=*False* **and** *bound*=λ-. ∞, *OF WT-res*], *simplified*, *OF - lossless-res*])
unfold-locales

theorem (**in** *constructive-security*) *parallel-realisation2*:
assumes *WT-res*: $\bigwedge\eta$. $\mathcal{I}$-*res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* $\eta \vdash$*res res* η $\surd$
and *lossless-res*: $\bigwedge\eta$. *lossless* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$-*res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* η) (*res* η)
shows *constructive-security* ($\lambda\eta$. *parallel-wiring* $\triangleright$ *real-resource* $\eta \parallel$ *res* η) ($\lambda\eta$. *parallel-wiring* $\triangleright$ (*ideal-resource* $\eta \parallel$ *res* η)) ($\lambda\eta$. *parallel-converter2* (*sim* η) *id-converter*)
($\lambda\eta$. $\mathcal{I}$-*real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*res* η) ($\lambda\eta$. $\mathcal{I}$-*ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*res* η) ($\lambda\eta$. $\mathcal{I}$-*common* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*common'* η) *bound lossless* ($\lambda\eta$. *w* $\eta \mid_w$ (*id*, *id*))
by(*rule parallel-constructive-security*[*OF - constructive-security-trivial*[**where** *lossless*=*False* **and** *bound*=λ-. ∞, *OF WT-res*], *simplified*, *OF - lossless-res*])
unfold-locales

theorem (**in** *constructive-security*) *parallel-resource1*:
assumes *WT-res* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$-*res* $\eta \vdash$*res res* η $\surd$
and *lossless-res* [*plossless-intro*]: $\bigwedge\eta$. *lossless* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$-*res* η) (*res* η)
shows *constructive-security* ($\lambda\eta$. *parallel-resource1-wiring* $\triangleright$ *res* $\eta \parallel$ *real-resource* η)
($\lambda\eta$. *parallel-resource1-wiring* $\triangleright$ *res* $\eta \parallel$ *ideal-resource* η) *sim*
$\mathcal{I}$-*real* $\mathcal{I}$-*ideal* ($\lambda\eta$. $\mathcal{I}$-*res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η) *bound lossless w*
proof

fix $\mathcal{A}$ :: *security* $\Rightarrow$ $('a + 'g + 'e, 'b + 'h + 'f)$ *distinguisher*
assume *WT* [*WT-intro*]: $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) $\vdash g$ $\mathcal{A}$ η $\surd$ **for** η
assume *bound* [*interaction-bound*]: *interaction-any-bounded-by* ($\mathcal{A}$ η) (*bound* η) **for** η
assume *lossless* [*plossless-intro*]: *lossless* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η)) ($\mathcal{A}$ η) **for** η

let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*swap-lassocr* $\odot$ *parallel-converter* (*converter-of-resource* (*res* η)) 1_C)
have *ideal*:
connect ($\mathcal{A}$ η) (*sim* η $|_=$ 1_C $\rhd$ *parallel-resource1-wiring* $\rhd$ *res* η $\parallel$ *ideal-resource* η) =
connect (*?*$\mathcal{A}$ η) (*sim* η $|_=$ 1_C $\rhd$ *ideal-resource* η) **for** η
proof −
have[*intro*]: $\mathcal{I}$*-uniform* (*outs-*$\mathcal{I}$ ($\mathcal{I}$*-real* η) <+> *outs-*$\mathcal{I}$ ($\mathcal{I}$*-res* η) <+> *outs-*$\mathcal{I}$ ($\mathcal{I}$*-common* η))
UNIV,$\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) $\vdash_C$ *sim* η $|_=$ 1_C $\sim$ *sim* η $|_=$ 1_C $|_=$ 1_C
by(*rule eq-*$\mathcal{I}$*-converter-mono*) (*auto simp add*: *le-*$\mathcal{I}$*-def intro*!:
WT-intro parallel-converter2-id-id[*symmetric*] *parallel-converter2-eq-*$\mathcal{I}$*-cong eq-*$\mathcal{I}$*-converter-reflI*)

have $*$: *outs-*$\mathcal{I}$ ($\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η)) $\vdash_R$ (*sim* η $|_=$ 1_C) $\odot$ *parallel-resource1-wiring* $\rhd$ *res* η $\parallel$ *ideal-resource* η $\sim$
swap-lassocr $\odot$ (*converter-of-resource* (*res* η) $|_\propto$ 1_C) $\rhd$ *sim* η $|_=$ 1_C $\rhd$ *ideal-resource* η
by (*rule eq-resource-on-trans*[**where** *res'*=(*sim* η $|_=$ 1_C $|_=$ 1_C) $\odot$ *parallel-resource1-wiring* $\rhd$ *res* η $\parallel$ *ideal-resource* η],
rule eq-$\mathcal{I}$*-attach-on*, (*rule WT-intro*)+, *rule eq-*$\mathcal{I}$*-comp-cong*)
(*auto simp add*: *parallel-resource1-wiring-def comp-swap-lassocr attach-compose attach-parallel2*
attach-converter-of-resource-conv-parallel-resource intro!: *WT-intro eq-*$\mathcal{I}$*-converter-reflI*)

show *?thesis*
unfolding *distinguish-attach*[*symmetric*] **using** *WT*
by(*rule connect-eq-resource-cong*, *subst attach-compose*[*symmetric*])
(*intro* $*$, (*rule WT-intro*)+)
qed
have *real*:
connect ($\mathcal{A}$ η) (*parallel-resource1-wiring* $\rhd$ *res* η $\parallel$ *real-resource* η) =
connect (*?*$\mathcal{A}$ η) (*real-resource* η) **for** η
unfolding *distinguish-attach*[*symmetric*]
by(*simp add*: *attach-compose attach-converter-of-resource-conv-parallel-resource parallel-resource1-wiring-def*)
have $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash g$ *?*$\mathcal{A}$ η $\surd$ **for** η **by**(*rule WT-intro*)+
moreover have *interaction-any-bounded-by* (*?*$\mathcal{A}$ η) (*bound* η) **for** η
by *interaction-bound-converter simp*

moreover have *plossless-gpv* (*I-real* η $\oplus_{\mathcal{I}}$ *I-common* η) (*?A* η) **if** *lossless* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
ultimately show *negligible* ($\lambda\eta$. *advantage* (*A* η) (*sim* η $|_=$ 1_C $\rhd$
parallel-resource1-wiring $\rhd$ *res* η $\|$ *ideal-resource* η)
(*parallel-resource1-wiring* $\rhd$ *res* η $\|$ *real-resource* η))
unfolding *advantage-def ideal real* **by**(*rule adv*[*unfolded advantage-def*])
next
from *correct* **obtain** *cnv*
where *correct'*: $\bigwedge\mathcal{D}$. $[\![$ $\bigwedge\eta$. *I-ideal* η $\oplus_{\mathcal{I}}$ *I-common* η $\vdash g$ $\mathcal{D}$ η $\surd$;
$\bigwedge\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound* η);
$\bigwedge\eta$. *lossless* $\Longrightarrow$ *plossless-gpv* (*I-ideal* η $\oplus_{\mathcal{I}}$ *I-common* η) ($\mathcal{D}$ η) $]\!]$
$\Longrightarrow$ ($\forall\eta$. *wiring* (*I-ideal* η) (*I-real* η) (*cnv* η) (*w* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))
by *blast*
show $\exists$ *cnv*. $\forall\mathcal{D}$. ($\forall\eta$. *I-ideal* η $\oplus_{\mathcal{I}}$ (*I-res* η $\oplus_{\mathcal{I}}$ *I-common* η) $\vdash g$ $\mathcal{D}$ η $\surd$) $\longrightarrow$
($\forall\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound* η)) $\longrightarrow$
($\forall\eta$. *lossless* $\longrightarrow$ *plossless-gpv* (*I-ideal* η $\oplus_{\mathcal{I}}$ (*I-res* η $\oplus_{\mathcal{I}}$ *I-common* η)) ($\mathcal{D}$ η)) $\longrightarrow$
($\forall\eta$. *wiring* (*I-ideal* η) (*I-real* η) (*cnv* η) (*w* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*parallel-resource1-wiring* $\rhd$ *res* η $\|$ *ideal-resource* η)
(*cnv* η $|_=$ 1_C $\rhd$ *parallel-resource1-wiring* $\rhd$ *res* η $\|$ *real-resource* η))
proof(*intro exI conjI strip*)
fix $\mathcal{D}$ :: *security* $\Rightarrow$ ('*c* + '*g* + '*e*, '*d* + '*h* + '*f*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall\eta$. *I-ideal* η $\oplus_{\mathcal{I}}$ (*I-res* η $\oplus_{\mathcal{I}}$ *I-common* η) $\vdash g$ $\mathcal{D}$ η $\surd$
and *bound* [*rule-format*, *interaction-bound*]: $\forall\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound* η)
and *lossless* [*rule-format*, *plossless-intro*]: $\forall\eta$. *lossless* $\longrightarrow$ *plossless-gpv* (*I-ideal* η $\oplus_{\mathcal{I}}$ (*I-res* η $\oplus_{\mathcal{I}}$ *I-common* η)) ($\mathcal{D}$ η)

let *?D* = $\lambda\eta$. *absorb* ($\mathcal{D}$ η) (*swap-lassocr* $\odot$ *parallel-converter* (*converter-of-resource* (*res* η)) 1_C)
have *WT'*: *I-ideal* η $\oplus_{\mathcal{I}}$ *I-common* η $\vdash g$ *?D* η $\surd$ **for** η **by**(*rule WT-intro*)+
have *bound'*: *interaction-any-bounded-by* (*?D* η) (*bound* η) **for** η **by** *interaction-bound simp*
have *lossless'*: *plossless-gpv* (*I-ideal* η $\oplus_{\mathcal{I}}$ *I-common* η) (*?D* η) **if** *lossless* **for** η
by(*rule plossless-intro WT-intro that*)+
from *correct'*[*OF WT' bound' lossless'*]
have *w*: *wiring* (*I-ideal* η) (*I-real* η) (*cnv* η) (*w* η)
and *adv*: *negligible* ($\lambda\eta$. *advantage* (*?D* η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))
for η **by** *auto*
show *wiring* (*I-ideal* η) (*I-real* η) (*cnv* η) (*w* η) **for** η **by**(*rule w*)
from *w* **have** [*WT-intro*]: *I-ideal* η, *I-real* η $\vdash_C$ *cnv* η $\surd$ **for** η **by** *cases*
have *connect* ($\mathcal{D}$ η) (*swap-lassocr* $\rhd$ *res* η $\|$ (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))

=
connect ($\mathcal{D}$ η) (*cnv* η $|_=$ 1_C $\rhd$ *swap-lassocr* $\rhd$ *res* η $\|$ *real-resource* η) **for** η
proof −
have *connect* ($\mathcal{D}$ η) (*cnv* η $|_=$ 1_C $\rhd$ *swap-lassocr* $\rhd$ *res* η $\|$ *real-resource* η)
=
connect ($\mathcal{D}$ η) (*cnv* η $|_=$ 1_C $|_=$ 1_C $\rhd$ *swap-lassocr* $\rhd$ *res* η $\|$ *real-resource* η)
by(*rule connect-eq-resource-cong*[*OF WT-D*])
(*rule eq-$\mathcal{I}$-attach-on′ WT-intro parallel-converter2-eq-$\mathcal{I}$-cong eq-$\mathcal{I}$-converter-reflI parallel-converter2-id-id*[*symmetric*] | *simp*)+
also have ... = *connect* ($\mathcal{D}$ η) (*swap-lassocr* $\rhd$ *res* η $\|$ (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))
by(*simp add*: *comp-swap-lassocr′ attach-compose attach-parallel2*)
finally show *?thesis* **by** *simp*
qed
with *adv* **show** *negligible* ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*parallel-resource1-wiring* $\rhd$ *res* η $\|$ *ideal-resource* η)
(*cnv* η $|_=$ 1_C $\rhd$ *parallel-resource1-wiring* $\rhd$ *res* η $\|$ *real-resource* η))
by(*simp add*: *advantage-def distinguish-attach*[*symmetric*] *attach-compose attach-converter-of-resource-conv-parallel-resource parallel-resource1-wiring-def*)
qed
qed(*rule WT-intro*)+

end

8 Examples

theory *System-Construction* **imports**
../../Constructive-Cryptography
begin

8.1 Random oracle resource

locale *rorc* =
fixes *range* :: *'r set*
begin

fun *rnd-oracle* :: (*'m* ⇒ *'r option*, *'m*, *'r*) *oracle′* **where**
rnd-oracle f m = (*case f m of*
(*Some r*) ⇒ *return-spmf* (*r*, *f*)
| *None* ⇒ *do* {
r ← *spmf-of-set* (*range*);
return-spmf (*r*, *f*(*m* := *Some r*))})

definition *res* = *RES* (*rnd-oracle* $\oplus_O$ *rnd-oracle*) *Map.empty*

end

8.2 Key resource

locale *key* =
  fixes *key-gen* :: *'k spmf*
begin

fun *key-oracle* :: (*'k option*, *unit*, *'k*) *oracle'* **where**
  key-oracle None () = *do* { *k* ← *key-gen*; *return-spmf* (*k*, *Some k*)}
| *key-oracle* (*Some x*) () = *return-spmf* (*x*, *Some x*)

definition *res* = *RES* (*key-oracle* ⊕$_O$ *key-oracle*) *None*

end

8.3 Channel resource

datatype *'a cstate* = *Void* | *Fail* | *Store 'a* | *Collect 'a*

datatype *'a aquery* = *Look* | *ForwardOrEdit* (*forward-or-edit*: *'a*) | *Drop*
type-synonym *'a insec-query* = *'a option aquery*
type-synonym *auth-query* = *unit aquery*

consts *Forward* :: *'a aquery*
abbreviation *Forward-auth* :: *auth-query* **where** *Forward-auth* ≡ *ForwardOrEdit*
()
abbreviation *Forward-insec* :: *'a insec-query* **where** *Forward-insec* ≡ *Forwar-*
dOrEdit None
abbreviation *Edit* :: *'a* ⇒ *'a insec-query* **where** *Edit m* ≡ *ForwardOrEdit* (*Some*
m)
adhoc-overloading *Forward Forward-auth*
adhoc-overloading *Forward Forward-insec*

translations
  (*logic*) *CONST Forward* <= (*logic*) *CONST ForwardOrEdit* (*CONST None*)
  (*logic*) *CONST Forward* <= (*logic*) *CONST ForwardOrEdit* (*CONST Product-Type.Unity*)
  (*type*) *auth-query* <= (*type*) *unit aquery*
  (*type*) *'a insec-query* <= (*type*) *'a option aquery*

8.3.1 Generic channel

locale *channel* =
  fixes *side-oracle* :: (*'m cstate*, *'a*, *'b option*) *oracle'*
begin

fun *send-oracle* :: (*'m cstate*, *'m*, *unit*) *oracle'* **where**
  send-oracle Void m = *return-spmf* ((), *Store m*)
| *send-oracle s m* = *return-spmf* ((), *s*)

fun *recv-oracle* :: (*'m cstate*, *unit*, *'m option*) *oracle'* **where**
  recv-oracle (*Collect m*) () = *return-spmf* (*Some m*, *Fail*)

```
| recv-oracle s          () = return-spmf (None, s)

definition res :: ('a + 'm + unit, 'b option + unit + 'm option) resource where
  res ≡ RES (side-oracle ⊕O send-oracle ⊕O recv-oracle) Void

end
```

8.3.2 Insecure channel

```
locale insec-channel
begin

fun insec-oracle :: ('m cstate, 'm insec-query, 'm option) oracle' where
  insec-oracle Void      (Edit m') = return-spmf (None, Collect m')
| insec-oracle (Store m) (Edit m') = return-spmf (None, Collect m')
| insec-oracle (Store m) Forward   = return-spmf (None, Collect m)
| insec-oracle (Store m) Drop      = return-spmf (None, Fail)
| insec-oracle (Store m) Look      = return-spmf (Some m, Store m)
| insec-oracle s         -         = return-spmf (None, s)

sublocale channel insec-oracle .

end
```

8.3.3 Authenticated channel

```
locale auth-channel
begin

fun auth-oracle :: ('m cstate, auth-query, 'm option) oracle' where
  auth-oracle (Store m) Forward = return-spmf (None, Collect m)
| auth-oracle (Store m) Drop    = return-spmf (None, Fail)
| auth-oracle (Store m) Look    = return-spmf (Some m, Store m)
| auth-oracle s         -       = return-spmf (None, s)

sublocale channel auth-oracle .

end

fun insec-query-of :: auth-query ⇒ 'm insec-query where
  insec-query-of Forward = Forward
| insec-query-of Drop = Drop
| insec-query-of Look = Look

abbreviation (input) auth-response-of :: ('mac × 'm) option ⇒ 'm option
  where auth-response-of ≡ map-option snd

abbreviation insec-auth-wiring :: (auth-query, 'm option, ('mac × 'm) insec-query,
('mac × 'm) option) wiring
  where insec-auth-wiring ≡ (insec-query-of, auth-response-of)
```

8.3.4 Secure channel

locale *sec-channel*
begin

fun *sec-oracle* :: (*'a list cstate*, *auth-query*, *nat option*) *oracle'* **where**
  sec-oracle (*Store m*) *Forward* = *return-spmf* (*None*, *Collect m*)
| *sec-oracle* (*Store m*) *Drop* = *return-spmf* (*None*, *Fail*)
| *sec-oracle* (*Store m*) *Look* = *return-spmf* (*Some* (*length m*), *Store m*)
| *sec-oracle s* - = *return-spmf* (*None*, *s*)

sublocale *channel sec-oracle* **.**

end

abbreviation (*input*) *auth-query-of* :: *auth-query* ⇒ *auth-query*
  where *auth-query-of* ≡ *id*

abbreviation (*input*) *sec-response-of* :: *'a list option* ⇒ *nat option*
  where *sec-response-of* ≡ *map-option length*

abbreviation *auth-sec-wiring* :: (*auth-query*, *nat option*, *auth-query*, *'a list option*) *wiring*
  where *auth-sec-wiring* ≡ (*auth-query-of*, *sec-response-of*)

8.4 Cipher converter

locale *cipher* =
  AUTH: *auth-channel* + *KEY*: *key key-alg*
  for *key-alg* :: *'k spmf* +
  fixes *enc-alg* :: *'k* ⇒ *'m* ⇒ *'c spmf*
    and *dec-alg* :: *'k* ⇒ *'c* ⇒ *'m option*
begin

definition *enc* :: (*'m*, *unit*, *unit* + *'c*, *'k* + *unit*) *converter* **where**
  enc ≡ *CNV* (*stateless-callee* (λ*m*. *do* {
    k ← *Pause* (*Inl* ()) *Done*;
    c ← *lift-spmf* (*enc-alg* (*projl k*) *m*);
    (- :: *'k* + *unit*) ← *Pause* (*Inr c*) *Done*;
    Done (())
  })) ()

definition *dec* :: (*unit*, *'m option*, *unit* + *unit*, *'k* + *'c option*) *converter* **where**
  dec ≡ *CNV* (*stateless-callee* (λ-. *Pause* (*Inr* ()) (λ*c'*.
    case c' of Inr (*Some c*) ⇒ (*do* {
      k ← *Pause* (*Inl* ()) *Done*;
      Done (*dec-alg* (*projl k*) *c*) })
    | - ⇒ *Done None*)
  )) ()

definition πE :: (*auth-query*, *'c option*, *auth-query*, *'c option*) *converter* (π^E)
where
$\pi^E \equiv 1_C$

definition *routing* $\equiv (1_C \mid_= lassocr_C) \odot$ *swap-lassocr* $\odot (1_C \mid_= (1_C \mid_= swap\text{-}lassocr) \odot$ *swap-lassocr*$) \odot rassocl_C$

definition *res* $= (1_C \mid_= enc \mid_= dec) \rhd (1_C \mid_=$ *parallel-wiring*$) \rhd$ *parallel-resource1-wiring* $\rhd$ (*KEY.res* $\parallel$ *AUTH.res*)

lemma *res-alt-def*: *res* $=$ ((*1*$_C$ $\mid_=$ *enc* $\mid_=$ *dec*) $\odot$ (*1*$_C$ $\mid_=$ *parallel-wiring*)) $\rhd$ *parallel-resource1-wiring* $\rhd$ (*KEY.res* $\parallel$ *AUTH.res*)
by(*simp add*: *res-def attach-compose*)

end

8.5 Message authentication converter

locale *macode* =
INSEC: *insec-channel* + *RO*: *rorc range*
for *range* :: *'r set* +
fixes *mac-alg* :: *'r* $\Rightarrow$ *'m* $\Rightarrow$ *'a spmf*
begin

definition *enm* :: (*'m*, *unit*, *'m* + (*'a* $\times$ *'m*), *'r* + *unit*) *converter* **where**
enm $\equiv$ *CNV* (λ*bs m*. *if bs*
then Done ((), *True*)
else do {
r $\leftarrow$ *Pause* (*Inl m*) *Done*;
a $\leftarrow$ *lift-spmf* (*mac-alg* (*projl r*) *m*);
(- :: *'r* + *unit*) $\leftarrow$ *Pause* (*Inr* (*a*, *m*)) *Done*;
Done ((), *True*)
}) *False*

definition *dem* :: (*unit*, *'m option*, *'m* + *unit*, *'r* + (*'a* $\times$ *'m*) *option*) *converter* **where**
dem $\equiv$ *CNV* (*stateless-callee* (λ-. *Pause* (*Inr* ()) (λ*am'*.
case am' of Inr (*Some* (*a*, *m*)) $\Rightarrow$ (*do* {
r $\leftarrow$ *Pause* (*Inl m*) *Done*;
a' $\leftarrow$ *lift-spmf* (*mac-alg* (*projl r*) *m*);
Done (*if a'* $=$ *a then Some m else None*) })
| - $\Rightarrow$ *Done None*)
)) ()

definition πE :: ((*'a* $\times$ *'m*) *insec-query*, (*'a* $\times$ *'m*) *option*, (*'a* $\times$ *'m*) *insec-query*, (*'a* $\times$ *'m*) *option*) *converter* (π^E) **where**
$\pi^E \equiv 1_C$

definition *routing* $\equiv (1_C \mid_= lassocr_C) \odot$ *swap-lassocr* $\odot (1_C \mid_= (1_C \mid_= swap\text{-}lassocr)$

$\odot$ *swap-lassocr*) $\odot$ $rassocl_C$

definition *res* = (1_C $|_=$ *enm* $|_=$ *dem*) $\triangleright$ (1_C $|_=$ *parallel-wiring*) $\triangleright$ *parallel-resource1-wiring* $\triangleright$ (*RO.res* $\parallel$ *INSEC.res*)

end

lemma *interface-wiring*:
(*cnv-adv* $|_=$ *cnv-send* $|_=$ *cnv-recv*) $\triangleright$ (1_C $|_=$ *parallel-wiring*) $\triangleright$ *parallel-resource1-wiring* $\triangleright$
(*RES* (*res2-send* $\oplus_O$ *res2-recv*) *res2-s* $\parallel$ *RES* (*res1-adv* $\oplus_O$ *res1-send* $\oplus_O$ *res1-recv*) *res1-s*)
=
cnv-adv $|_=$ *cnv-send* $|_=$ *cnv-recv* $\triangleright$
RES (†*res1-adv* $\oplus_O$ (*res2-send*† $\oplus_O$ †*res1-send*) $\oplus_O$ *res2-recv*† $\oplus_O$ †*res1-recv*) (*res2-s*, *res1-s*)
(**is** - $\triangleright$ *?L1* $\triangleright$ *?L2* $\triangleright$ *?L3* = - $\triangleright$ *?R*)
proof −
let *?wiring* = (*id*, *id*) $|_w$ ($lassocr_w$ $\circ_w$ ((*id*, *id*) $|_w$ ($rassocl_w$ $\circ_w$ ($swap_w$ $|_w$ (*id*, *id*) $\circ_w$ $lassocr_w$)) $\circ_w$ $rassocl_w$)) $\circ_w$ ($rassocl_w$ $\circ_w$ ($swap_w$ $|_w$ (*id*, *id*) $\circ_w$ $lassocr_w$))

have *?L1* $\triangleright$ *?L2* $\triangleright$ *?L3* = *?L1* $\odot$ *?L2* $\triangleright$
RES ((*res2-send*† $\oplus_O$ *res2-recv*†) $\oplus_O$ †*res1-adv* $\oplus_O$ †*res1-send* $\oplus_O$ †*res1-recv*) (*res2-s*, *res1-s*) (**is** - = - $\triangleright$ *RES*(*?O*) *?S*)
unfolding *attach-compose*[*symmetric*] *resource-of-parallel-oracle*[*symmetric*]
by (*simp only*: *parallel-oracle-conv-plus-oracle extend-state-oracle-plus-oracle extend-state-oracle2-plus-oracle*)
also have ... = *RES*(*apply-wiring ?wiring ?O*) *?S*
by (*rule attach-wiring-resource-of-oracle*, *simp only*: *parallel-wiring-def parallel-resource1-wiring-def swap-lassocr-def*)
((*rule wiring-intro WT-resource-of-oracle WT-plus-oracleI WT-callee-full*)+, *simp-all*)
also have ... = *?R* **by** *simp*
finally show *?thesis* **by** (*rule arg-cong2*[**where** *f*=*attach*, *OF refl*])
qed

definition *id′* **where** *id′* = *id*

end

9 Security of one-time-pad encryption

theory *One-Time-Pad* **imports**
System-Construction
begin

definition *key* :: *security* ⇒ *bool list spmf* **where**
key η ≡ *spmf-of-set* (*nlists UNIV* η)

definition *enc* :: *security* ⇒ *bool list* ⇒ *bool list* ⇒ *bool list spmf* **where**
enc η *k m* ≡ *return-spmf* (*k* [⊕] *m*)

definition *dec* :: *security* ⇒ *bool list* ⇒ *bool list* ⇒ *bool list option* **where**
dec η *k c* ≡ *Some* (*k* [⊕] *c*)

definition *sim* :: *'b list option* ⇒ *'a* ⇒ (*'b list option* × *'b list option*, *'a*, *nat option*) *gpv* **where**
sim c q ≡ (*do* {
lo ← *Pause q Done*;
(*case lo of*
Some n ⇒ *if c* = *None*
then do {
x ← *lift-spmf* (*spmf-of-set* (*nlists UNIV n*));
Done (*Some x*, *Some x*)}
else Done (*c*, *c*)
| *None* ⇒ *Done* (*None*, *c*))})

context
fixes η :: *security*
begin

private definition *key-channel-send* :: *bool list option* × *bool list cstate*
⇒ *bool list* ⇒ (*unit* × *bool list option* × *bool list cstate*) *spmf* **where**
key-channel-send s m ≡ *do* {
(*k*, *s*) ← (*key.key-oracle* (*key* η))† *s* ();
c ← *enc* η *k m*;
(-, *s*) ← †*channel.send-oracle s c*;
return-spmf ((), *s*)}

private definition *key-channel-recv* :: *bool list option* × *bool list cstate*
⇒ *'a* ⇒ (*bool list option* × *bool list option* × *bool list cstate*) *spmf* **where**
key-channel-recv s m ≡*do* {
(*c*, *s*) ← †*channel.recv-oracle s* ();
(*case c of None* ⇒ *return-spmf* (*None*, *s*)
| *Some c'* ⇒ *do* {
(*k*, *s*) ← (*key.key-oracle* (*key* η))† *s* ();
return-spmf (*dec* η *k c'*, *s*)})}

private abbreviation *callee-sec-channel* **where**
callee-sec-channel callee ≡ *lift-state-oracle extend-state-oracle* (*attach-callee callee sec-channel.sec-oracle*)

private inductive *S* :: (*bool list option* × *unit* × *bool list cstate*) *spmf* ⇒
(*bool list option* × *bool list cstate*) *spmf* ⇒ *bool* **where**

```
  S (return-spmf (None, (), Void))
     (return-spmf (None, Void))
| S (return-spmf (None, (), Store plain))
    (map-spmf (λkey. (Some key, Store (key [⊕] plain))) (spmf-of-set (nlists UNIV
η)))
if length plain = id' η
| S (return-spmf (None, (), Collect plain))
      (map-spmf (λkey. (Some key, Collect (key [⊕] plain))) (spmf-of-set (nlists
UNIV η)))
if length plain = id' η
| S (return-spmf (Some (key [⊕] plain), (), Store plain))
     (return-spmf (Some key, Store (key [⊕] plain)))
if length plain = id' η length key = id' η for key
| S (return-spmf (Some (key [⊕] plain), (), Collect plain))
     (return-spmf (Some key, Collect (key [⊕] plain)))
if length plain = id' η length key = id' η for key
| S (return-spmf (None, (), Fail))
     (map-spmf (λx. (Some x, Fail)) (spmf-of-set (nlists UNIV η)))
| S (return-spmf (Some (key [⊕] plain), (), Fail))
     (return-spmf (Some key, Fail))
if length plain = id' η length key = id' η for key plain


lemma resources-indistinguishable:
  shows (UNIV <+> nlists UNIV (id' η) <+> UNIV) ⊢R
   RES (callee-sec-channel sim ⊕O ††channel.send-oracle ⊕O ††channel.recv-oracle)
(None :: bool list option, (), Void)
    ≈
     RES (†auth-channel.auth-oracle ⊕O key-channel-send ⊕O key-channel-recv)
(None :: bool list option, Void)
    (is ?A ⊢R RES (?L1 ⊕O ?L2 ⊕O ?L3) ?SL ≈ RES (?R1 ⊕O ?R2 ⊕O ?R3)
?SR)
proof −
  note [simp] =
   exec-gpv-bind spmf.map-comp o-def map-bind-spmf bind-map-spmf bind-spmf-const
    sec-channel.sec-oracle.simps auth-channel.auth-oracle.simps
    channel.send-oracle.simps key-channel-send-def
    channel.recv-oracle.simps key-channel-recv-def
    key.key-oracle.simps dec-def key-def enc-def

  have ∗: ?A ⊢C ?L1 ⊕O ?L2 ⊕O ?L3(?SL) ≈ ?R1 ⊕O ?R2 ⊕O ?R3(?SR)
 proof(rule trace'-eqI-sim[where S=S], goal-cases Init-OK Output-OK State-OK)
    case Init-OK
    show ?case by (simp add: S.simps)
  next
    case (Output-OK p q query)
    show ?case
    proof (cases query)
      case (Inl adv-query)
```

```
      with Output-OK show ?thesis
      proof (cases adv-query)
        case Look
        with Output-OK Inl show ?thesis
        proof cases
          case Store-State-Channel: (2 plain)

          have∗: length plain = id' η ⟹
            map-spmf (λx. Inl (Some x)) (spmf-of-set (nlists UNIV (id' η))) =
            map-spmf (λx. Inl (Some x)) (map-spmf (λx. x [⊕] plain) (spmf-of-set
(nlists UNIV η))) for η
           unfolding id'-def by (subst xor-list-commute, subst one-time-pad[where
xs=plain, symmetric]) simp-all

          from Store-State-Channel show ?thesis using Output-OK(2−) Inl Look
            by(simp add: sim-def, simp add: map-spmf-conv-bind-spmf[symmetric])
              (subst (2) spmf.map-comp[where f=λx. Inl (Some x), symmetric,
unfolded o-def], simp only: ∗)
        qed (auto simp add: sim-def)
      qed (auto simp add: sim-def id'-def elim: S.cases)
    next
      case Snd-Rcv: (Inr query')
      with Output-OK show ?thesis
      proof (cases query')
        case (Inr rcv-query)
        with Output-OK Snd-Rcv show ?thesis
        proof cases
          case Collect-State-Channel: (3 plain)
          then show ?thesis using Output-OK(2−) Snd-Rcv Inr
            by (simp cong: bind-spmf-cong-simp add: in-nlists-UNIV id'-def)
        qed simp-all
      qed (auto elim: S.cases)
    qed
  next
    case (State-OK p q query state answer state')
    then show ?case
    proof (cases query)
      case (Inl adv-query)
      with State-OK show ?thesis
      proof (cases adv-query)
        case Look
        with State-OK Inl show ?thesis
        proof cases
          case Store-State-Channel: (2 plain)
          have ∗: length plain = id' η ⟹ key ∈ nlists UNIV η ⟹
           S (cond-spmf-fst (map-spmf (λx. (Inl (Some x), Some x, (), Store plain))
              (spmf-of-set (nlists UNIV (id' η)))) (Inl (Some (key [⊕] plain))))
            (cond-spmf-fst (map-spmf (λx. (Inl (Some (x [⊕] plain)), Some x, Store
(x [⊕] plain)))
```

```
          (spmf-of-set (nlists UNIV η))) (Inl (Some (key [⊕] plain)))) for key
        proof(subst (1 2) cond-spmf-fst-map-Pair1, goal-cases)
          case 2
        note inj-onD[OF inj-on-xor-list-nlists, rotated, simplified xor-list-commute]
          with 2 show ?case
            unfolding inj-on-def by (auto simp add: id'-def)
        next
          case 5
        note inj-onD[OF inj-on-xor-list-nlists, rotated, simplified xor-list-commute]
          with 5 show ?case
            by (subst (1 2 3) inv-into-f-f)
                ((clarsimp simp add: inj-on-def), (auto simp add: S.simps id'-def
inj-on-def in-nlists-UNIV ))
        qed (simp-all add: id'-def in-nlists-UNIV min-def inj-on-def)
        from Store-State-Channel show ?thesis using State-OK(2−) Inl Look
       by (clarsimp simp add: sim-def) (simp add: map-spmf-conv-bind-spmf[symmetric]
∗ )
      qed (auto simp add: sim-def map-spmf-conv-bind-spmf[symmetric] S.simps)
    qed (erule S.cases; (simp add: sim-def, auto simp add: map-spmf-conv-bind-spmf[symmetric]
S.simps))+
    next
      case Snd-Rcv: (Inr query')
      with State-OK show ?thesis
      proof (cases query')
        case (Inr rcv-query)
        with State-OK Snd-Rcv show ?thesis
        proof cases
          case Collect-State-Channel: (3 plain)
          then show ?thesis using State-OK(2−) Snd-Rcv Inr
        by clarsimp (simp add: S.simps in-nlists-UNIV id'-def map-spmf-conv-bind-spmf[symmetric]
cong: bind-spmf-cong-simp)
      qed (simp add: sim-def, auto simp add: map-spmf-conv-bind-spmf[symmetric]
S.simps)
      qed (erule S.cases,
          (simp add: sim-def, auto simp add: map-spmf-conv-bind-spmf[symmetric]
S.simps  in-nlists-UNIV))
    qed
  qed

  from ∗ show ?thesis by simp
qed

lemma real-resource-wiring:
  shows cipher.res (key η) (enc η) (dec η)
    = RES (†auth-channel.auth-oracle ⊕O key-channel-send ⊕O key-channel-recv)
(None, Void)
  including lifting-syntax
proof −
  note[simp]= Rel-def prod.rel-eq[symmetric] split-def split-beta o-def exec-gpv-bind
```

bind-map-spmf
resource-of-oracle-rprodl rprodl-extend-state-oracle
conv-callee-parallel[*symmetric*] *extend-state-oracle-plus-oracle*[*symmetric*]
attach-CNV-RES attach-callee-parallel-intercept attach-stateless-callee

show *?thesis*
unfolding *channel.res-def cipher.res-def cipher.routing-def cipher.enc-def cipher.dec-def*
interface-wiring cipher.πE-def key.res-def key-channel-send-def key-channel-recv-def
by (*simp add*: *conv-callee-parallel-id-left*[**where** *s*=(), *symmetric*])
((*auto cong*: *option.case-cong simp add*: *option.case-distrib*[**where** *h*=λ*gpv. exec-gpv - gpv* -]
intro!: *extend-state-oracle-parametric*) | *subst lift-state-oracle-extend-state-oracle*)+
qed

lemma *ideal-resource-wiring*:
shows (*CNV callee s*) |= 1_C ▷ *channel.res sec-channel.sec-oracle*
= *RES* (*callee-sec-channel callee* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*
) (*s*, (), *Void*) (**is** *?L1* ▷ - = *?R*)
proof −
have[*simp*]: *I-full*, *I-full* $\oplus_{\mathcal{I}}$ (*I-full* $\oplus_{\mathcal{I}}$ *I-full*) $\vdash_C$ *?L1* ∼ *?L1* (**is** -, *?I* $\vdash_C$ - ∼ -)
by(*rule eq-I-converter-mono*)
(*rule parallel-converter2-eq-I-cong eq-I-converter-reflI WT-converter-I-full I-full-le-plus-I order-refl plus-I-mono*)+

have[*simp*]: *?I* ⊢*c* (*sec-channel.sec-oracle* $\oplus_O$ *channel.send-oracle* $\oplus_O$ *channel.recv-oracle*) *s* √ **for** *s*
by(*rule WT-plus-oracleI WT-parallel-oracle WT-callee-full*; (*unfold split-paired-all*) *?*)+

have[*simp*]: *?L1* ▷ *RES* (*sec-channel.sec-oracle* $\oplus_O$ *channel.send-oracle* $\oplus_O$ *channel.recv-oracle*) *Void* = *?R*
by(*simp add*: *conv-callee-parallel-id-right*[**where** *s′*=(), *symmetric*] *attach-CNV-RES*

attach-callee-parallel-intercept resource-of-oracle-rprodl extend-state-oracle-plus-oracle)

show *?thesis* **unfolding** *channel.res-def*
by (*subst eq-I-attach*[*OF WT-resource-of-oracle*, **where** *?I′* = *?I* **and** *?conv*=*?L1* **and** *?conv′*=*?L1*]) *simp-all*
qed

end

lemma *eq-I-gpv-Done1*:
eq-I-gpv A I (*Done x*) *gpv* ⟷ *lossless-spmf* (*the-gpv gpv*) ∧ (∀ *a*∈*set-spmf* (*the-gpv gpv*). *eq-I-generat A I* (*eq-I-gpv A I*) (*Pure x*) *a*)
by(*auto intro*: *eq-I-gpv.intros simp add*: *rel-spmf-return-spmf1 elim*: *eq-I-gpv.cases*)

lemma *eq-I-gpv-Done2*:

eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv (Done x) $\longleftrightarrow$ *lossless-spmf (the-gpv gpv)* $\wedge$ ($\forall$ *a*$\in$*set-spmf (the-gpv gpv). eq-$\mathcal{I}$-generat A $\mathcal{I}$ (eq-$\mathcal{I}$-gpv A $\mathcal{I}$) a (Pure x))*
by(*auto intro*: *eq-$\mathcal{I}$-gpv.intros simp add*: *rel-spmf-return-spmf2 elim*: *eq-$\mathcal{I}$-gpv.cases*)

context begin
interpretation *CIPHER*: *cipher key* η *enc* η *dec* η **for** η **.**
interpretation *S-CHAN*: *sec-channel* **.**

lemma *one-time-pad*:
defines *$\mathcal{I}$-real* $\equiv \lambda\eta$. *$\mathcal{I}$-uniform UNIV (insert None (Some ‘ nlists UNIV* η*))*
and *$\mathcal{I}$-ideal* $\equiv \lambda\eta$. *$\mathcal{I}$-uniform UNIV {None, Some* η*}*
and *$\mathcal{I}$-common* $\equiv \lambda\eta$. *$\mathcal{I}$-uniform (nlists UNIV* η*) UNIV* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-uniform UNIV (insert None (Some ‘ nlists UNIV* η*))*
shows
constructive-security2 CIPHER.res (λ-. *S-CHAN.res*) (λ-. *CNV sim None*)
$\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common (λ-. ∞) *False* (λ-. *auth-sec-wiring*)
proof
let *?$\mathcal{I}$-key* = $\lambda\eta$. *$\mathcal{I}$-uniform UNIV (nlists UNIV* η*)*
let *?$\mathcal{I}$-enc* = $\lambda\eta$. *$\mathcal{I}$-uniform (nlists UNIV* η*) UNIV*
let *?$\mathcal{I}$-dec* = $\lambda\eta$. *$\mathcal{I}$-uniform UNIV (insert None (Some ‘ nlists UNIV* η*))*
have *i1* [*WT-intro*]: *$\mathcal{I}$-uniform (nlists UNIV* η*) UNIV*, *?$\mathcal{I}$-key* η $\oplus_{\mathcal{I}}$ *?$\mathcal{I}$-enc* η $\vdash_C$ *CIPHER.enc* η $\surd$ **for** η
unfolding *CIPHER.enc-def* **by**(*rule WT-converter-of-callee*)(*auto simp add*: *stateless-callee-def enc-def in-nlists-UNIV*)
have *i2* [*WT-intro*]: *?$\mathcal{I}$-dec* η, *?$\mathcal{I}$-key* η $\oplus_{\mathcal{I}}$ *?$\mathcal{I}$-dec* η $\vdash_C$ *CIPHER.dec* η $\surd$ **for** η
unfolding *CIPHER.dec-def* **by**(*rule WT-converter-of-callee*)(*auto simp add*: *stateless-callee-def dec-def in-nlists-UNIV*)
have [*WT-intro*]: *$\mathcal{I}$-common* η, (*?$\mathcal{I}$-key* η $\oplus_{\mathcal{I}}$ *?$\mathcal{I}$-enc* η) $\oplus_{\mathcal{I}}$ (*?$\mathcal{I}$-key* η $\oplus_{\mathcal{I}}$ *?$\mathcal{I}$-dec* η) $\vdash_C$ *CIPHER.enc* η $\mid_=$ *CIPHER.dec* η $\surd$ **for** η
unfolding *$\mathcal{I}$-common-def* **by**(*rule WT-intro*)+
have *key*: *callee-invariant-on* (*CIPHER.KEY.key-oracle* η $\oplus_O$ *CIPHER.KEY.key-oracle* η) (*pred-option* (λx. *length x* = η)) (*?$\mathcal{I}$-key* η $\oplus_{\mathcal{I}}$ *?$\mathcal{I}$-key* η) **for** η
apply *unfold-locales*
subgoal for *s x y s′* **by**(*cases s*; *cases x*)(*auto simp add*: *option.pred-set*, *simp-all add*: *key-def in-nlists-UNIV*)
subgoal for *s* **by**(*cases s*)(*auto intro*!: *WT-calleeI*, *simp-all add*: *key-def in-nlists-UNIV*)
done
have *i3*: *?$\mathcal{I}$-key* η $\oplus_{\mathcal{I}}$ *?$\mathcal{I}$-key* η $\vdash$*res CIPHER.KEY.res* η $\surd$ **for** η
unfolding *CIPHER.KEY.res-def* **by**(*rule callee-invariant-on.WT-resource-of-oracle*[*OF key*]) *simp*
let *?I* = $\lambda\eta$. *pred-cstate* (λx. *length x* = η)
have *WT-auth*: *$\mathcal{I}$-real* η $\vdash$*c CIPHER.AUTH.auth-oracle s* $\surd$ **if** *?I* η *s* **for** η *s*
apply(*rule WT-calleeI*)
subgoal for *x y s′* **using** *that*
by(*cases* (*s*, *x*) *rule*: *CIPHER.AUTH.auth-oracle.cases*)(*auto simp add*: *$\mathcal{I}$-real-def in-nlists-UNIV intro*!: *imageI*)

```
    done
  have WT-recv: ?I-dec η ⊢c S-CHAN.recv-oracle s √ if pred-cstate (λx. length
x = η) s for η s
    using that by(cases s)(auto intro!: WT-calleeI simp add: in-nlists-UNIV)
  have WT-send: ?I-enc η ⊢c S-CHAN.send-oracle s √ for η s
    by(rule WT-calleeI)(auto)
 have ∗: callee-invariant-on (CIPHER.AUTH.auth-oracle ⊕O S-CHAN.send-oracle
⊕O S-CHAN.recv-oracle) (?I η)
     (I-real η ⊕I I-common η) for η
    apply unfold-locales
    subgoal for s x y s'
      by(cases x; cases (s, projl x) rule: CIPHER.AUTH.auth-oracle.cases; cases
projr x)(auto simp add: I-common-def in-nlists-UNIV)
   subgoal by(auto simp add: I-common-def WT-auth WT-recv intro: WT-calleeI)
    done
  have i4 [unfolded I-common-def, WT-intro]: I-real η ⊕I I-common η ⊢res
CIPHER.AUTH.res √ for η
   unfolding CIPHER.AUTH.res-def by(rule callee-invariant-on.WT-resource-of-oracle[OF
∗]) simp
  show cipher: I-real η ⊕I I-common η ⊢res CIPHER.res η √ for η
    unfolding CIPHER.res-def by(rule WT-intro i3)+

  show secure: I-ideal η ⊕I I-common η ⊢res S-CHAN.res √ for η
  proof −
    have[simp]:I-ideal η ⊢c S-CHAN.sec-oracle s √ if ?I η s for s
    proof (cases rule: WT-calleeI, goal-cases)
      case (1 call ret s')
    then show ?case using that by (cases (s, call) rule: S-CHAN.sec-oracle.cases)
(simp-all add: I-ideal-def)
    qed
    have[simp]: I-common η ⊢c (S-CHAN.send-oracle ⊕O S-CHAN.recv-oracle)
s √
      if pred-cstate (λx. length x = η) s for s
     unfolding I-common-def by(rule WT-plus-oracleI WT-send WT-recv that)+

    have ∗: callee-invariant-on (S-CHAN.sec-oracle ⊕O S-CHAN.send-oracle ⊕O
S-CHAN.recv-oracle) (?I η) (I-ideal η ⊕I I-common η)
      apply(unfold-locales)
      subgoal for s x y s'
         by(cases (s, projl x) rule: S-CHAN.sec-oracle.cases; cases projr x)(auto
simp add: I-common-def in-nlists-UNIV)
      subgoal by simp
      done

    show ?thesis unfolding S-CHAN.res-def
      by(rule callee-invariant-on.WT-resource-of-oracle[OF ∗]) simp
  qed

  have sim [WT-intro]: I-real η, I-ideal η ⊢C CNV sim s √ if s ≠ None ⟶
```

length (*the s*) = *η* **for** *s η*
using *that* **by**(*coinduction arbitrary*: *s*)(*auto simp add*: *sim-def* $\mathcal{I}$*-ideal-def* $\mathcal{I}$*-real-def in-nlists-UNIV*)
show $\mathcal{I}$*-real η*, $\mathcal{I}$*-ideal η* $\vdash_C$ *CNV sim None* $\surd$ **for** *η* **by**(*rule sim*) *simp*

{ fix $\mathcal{A}$ :: *security* ⇒ (*auth-query* + *bool list* + *unit*, *bool list option* + *unit* + *bool list option*) *distinguisher*
assume *WT*: $\bigwedge\eta$. $\mathcal{I}$*-real η* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common η* $\vdash g$ $\mathcal{A}$ *η* $\surd$
and *bound*: $\bigwedge\eta$. *interaction-bounded-by* (λ-. *True*) ($\mathcal{A}$ *η*) ∞
have *connect* ($\mathcal{A}$ *η*) (*CNV sim None* $|=$ 1_C ▷ *S-CHAN.res*) = *connect* ($\mathcal{A}$ *η*) (*CIPHER.res η*) **for** *η*
using - *WT*
proof(*rule connect-cong-trace*)
show (*UNIV* <+> *nlists UNIV* (*id' η*) <+> *UNIV*) $\vdash_R$ *CNV sim None* $|=$ 1_C ▷ *S-CHAN.res* ≈ *CIPHER.res η*
unfolding *ideal-resource-wiring real-resource-wiring*
by(*rule resources-indistinguishable*)
show *outs-gpv* ($\mathcal{I}$*-real η* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common η*) ($\mathcal{A}$ *η*) ⊆ *UNIV* <+> *nlists UNIV* (*id' η*) <+> *UNIV*
using *WT*[*of η*, *THEN WT-gpv-outs-gpv*]
by(*auto simp add*: $\mathcal{I}$*-real-def* $\mathcal{I}$*-common-def id'-def*)
show $\mathcal{I}$*-real η* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common η* $\vdash$*res CIPHER.res η* $\surd$ **by**(*rule cipher*)
show $\mathcal{I}$*-real η* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common η* $\vdash$*res CNV sim None* $|=$ 1_C ▷ *S-CHAN.res* $\surd$
by(*rule WT-intro secure* | *simp*)+
qed
then show *negligible* (λ*η*. *advantage* ($\mathcal{A}$ *η*) (*CNV sim None* $|=$ 1_C ▷ *S-CHAN.res*) (*CIPHER.res η*))
by(*simp add*: *advantage-def*)
next
let *?cnv* = *map-converter id id auth-query-of sec-response-of* 1_C :: (*auth-query*, *nat option*, *auth-query*, *bool list option*) *converter*
have [*simp*]: $\mathcal{I}$*-full*, *map-*$\mathcal{I}$ *id* (*map-option length*) $\mathcal{I}$*-full* $\vdash_C$ 1_C $\surd$
using *WT-converter-id order-refl*
by(*rule WT-converter-mono*)(*simp add*: *le-*$\mathcal{I}$*-def*)
have *WT* [*WT-intro*]: $\mathcal{I}$*-ideal η*, $\mathcal{I}$*-real η* $\vdash_C$ *?cnv* $\surd$ **for** *η*
by(*rule WT-converter-map-converter*)(*auto simp add*: $\mathcal{I}$*-ideal-def* $\mathcal{I}$*-real-def intro*!: *WT-converter-mono*[*OF WT-converter-id order-refl*] *simp add*: *le-*$\mathcal{I}$*-def in-nlists-UNIV*)
with *eq-*$\mathcal{I}$*-converter-reflI*[*OF this*]
have *wiring* ($\mathcal{I}$*-ideal η*) ($\mathcal{I}$*-real η*) *?cnv auth-sec-wiring* **for** *η* **..**
moreover
have *eq*: $\mathcal{I}$*-ideal η*, $\mathcal{I}$*-ideal η* $\vdash_C$ *map-converter id* (*map-option length*) *id id* (*CNV sim s*) ∼ 1_C
if *s* ≠ *None* ⟶ *length* (*the s*) = *η* **for** *η* **and** *s* :: *bool list option*
unfolding *map-converter-of-callee map-gpv-conv-map-gpv'*[*symmetric*] **using** *that*
by(*coinduction arbitrary*: *s*)
(*fastforce intro*!: *eq-*$\mathcal{I}$*-gpv-Pause simp add*: $\mathcal{I}$*-ideal-def in-nlists-UNIV eq-*$\mathcal{I}$*-gpv-Done2 gpv.map-sel eq-onp-same-args sim-def map-gpv-conv-bind*[*symmetric*]

id-def[*symmetric*] *split*!: *option.split if-split-asm*)
have $\mathcal{I}$-*ideal* η, $\mathcal{I}$-*ideal* $\eta \vdash_C$ *?cnv* $\odot$ *CNV sim None* $\surd$ **for** η **by**(*rule WT WT-intro*)+ *simp*
with - **have** *wiring* ($\mathcal{I}$-*ideal* η) ($\mathcal{I}$-*ideal* η) (*?cnv* $\odot$ *CNV sim None*) (*id*, *id*) **for** η
by(*rule wiring.intros*)(*auto simp add*: *comp-converter-map-converter1 comp-converter-id-left eq*)
ultimately show $\exists$ *cnv*. $\forall \eta$. *wiring* ($\mathcal{I}$-*ideal* η) ($\mathcal{I}$-*real* η) (*cnv* η) *auth-sec-wiring* $\wedge$
wiring ($\mathcal{I}$-*ideal* η) ($\mathcal{I}$-*ideal* η) (*cnv* $\eta \odot$ *CNV sim None*) (*id*, *id*)
by *meson*
}
qed

end

end

10 Security of message authentication

theory *Message-Authentication-Code* **imports**
System-Construction
begin

definition *rnd* :: *security* $\Rightarrow$ *bool list set* **where**
rnd $\eta \equiv$ *nlists UNIV* η

definition *mac* :: *security* $\Rightarrow$ *bool list* $\Rightarrow$ *bool list* $\Rightarrow$ *bool list spmf* **where**
mac η *r m* $\equiv$ *return-spmf r*

definition *vld* :: *security* $\Rightarrow$ *bool list set* **where**
vld $\eta \equiv$ *nlists UNIV* η

fun *valid-mac-query* :: *security* $\Rightarrow$ (*bool list* $\times$ *bool list*) *insec-query* $\Rightarrow$ *bool* **where**
valid-mac-query η (*ForwardOrEdit* (*Some* (*a*, *m*))) $\longleftrightarrow$ *a* $\in$ *vld* $\eta \wedge$ *m* $\in$ *vld* η
| *valid-mac-query* η - = *True*

fun *sim* :: (*'b list* $\times$ *'b list*) *option* + *unit* $\Rightarrow$ (*'b list* $\times$ *'b list*) *insec-query*
$\Rightarrow$ ((*'b list* $\times$ *'b list*) *option* $\times$ ((*'b list* $\times$ *'b list*) *option* + *unit*), *auth-query* , *'b list option*) *gpv* **where**
sim (*Inr* ()) - = *Done* (*None*, *Inr*())
| *sim* (*Inl None*) (*Edit* (*a'*, *m'*)) = *do* { - $\leftarrow$ *Pause Drop Done*; *Done* (*None*, *Inr* ())}
| *sim* (*Inl* (*Some* (*a*, *m*))) (*Edit* (*a'*, *m'*)) = (*if a* = *a'* $\wedge$ *m* = *m'*
then do { - $\leftarrow$ *Pause Forward Done*; *Done* (*None*, *Inl* (*Some* (*a*, *m*)))}
else do { - $\leftarrow$ *Pause Drop Done*; *Done* (*None*, *Inr* ())})
| *sim* (*Inl None*) *Forward* = *do* {
Pause Forward Done;
Done (*None*, *Inl None*) }

```
| sim (Inl (Some -))       Forward          = do {
    Pause Forward Done;
    Done (None, Inr ()) }
| sim (Inl None)           Drop             = do {
    Pause Drop Done;
    Done (None, Inl None) }
| sim (Inl (Some -))       Drop             = do {
    Pause Drop Done;
    Done (None, Inr ()) }
| sim (Inl (Some (a, m)))  Look             = do {
    lo ← Pause Look Done;
    (case lo of
      Some m ⇒ Done (Some (a, m), Inl (Some (a, m)))
    | None   ⇒ Done (None, Inl (Some (a, m))))}
| sim (Inl None)           Look             = do {
    lo ← Pause Look Done;
    (case lo of
      Some m ⇒ do {
        a ← lift-spmf (spmf-of-set (nlists UNIV (length m)));
        Done (Some (a, m), Inl (Some (a, m)))}
    | None   ⇒ Done (None, Inl None))}


context
  fixes η :: security
begin

private definition rorc-channel-send :: ((bool × unit) × (bool list ⇒ bool list
option) × (bool list × bool list) cstate, bool list, unit) oracle′ where
  rorc-channel-send s m ≡ (if fst (fst s)
    then return-spmf ((), (True, ()), snd s)
    else do {
      (r, s) ← (rorc.rnd-oracle (rnd η))† (snd s) m;
      a ← mac η r m;
      (-, s) ← †channel.send-oracle s (a, m);
      return-spmf ((), (True, ()), s)
    })

private definition rorc-channel-recv :: ((bool × unit) × (bool list ⇒ bool list
option) × (bool list × bool list) cstate, unit, bool list option) oracle′ where
  rorc-channel-recv s q ≡ do {
    (m, s) ← ††channel.recv-oracle s ();
    (case m of
      None ⇒ return-spmf (None, s)
    | Some (a, m) ⇒ do {
        (r, s) ← †(rorc.rnd-oracle (rnd η))† s m;
        a′ ← mac η r m;
        return-spmf (if a′ = a then Some m else None, s)})
  }
```

```
private definition rorc-channel-recv-f :: ((bool list ⇒ bool list option) × (bool list
× bool list) cstate, unit, bool list option) oracle′ where
  rorc-channel-recv-f s q ≡ do {
    (am, (as, ams)) ← †channel.recv-oracle s ();
    (case am of
      None ⇒ return-spmf (None, (as, ams))
    | Some (a, m) ⇒ (case as m of
        None ⇒ do {
          a′′ :: bool list ← spmf-of-set (nlists UNIV η − {a});
          a′ ← spmf-of-set (nlists UNIV η);
          (if a′ = a
            then return-spmf (None, as(m := Some a′′), ams)
            else return-spmf (None, as(m := Some a′), ams)) }
      | Some a′ ⇒ return-spmf (if a′ = a then Some m else None, as, ams)))}

private fun lazy-channel-send :: (bool list cstate × (bool list × bool list) option ×
(bool list ⇒ bool list option), bool list, unit) oracle′ where
  lazy-channel-send (Void, es) m = return-spmf ((), (Store m, es))
| lazy-channel-send s          m = return-spmf ((), s)

private fun lazy-channel-recv :: (bool list cstate × (bool list × bool list) option ×
(bool list ⇒ bool list option), unit, bool list option) oracle′ where
  lazy-channel-recv (Collect m, None, as)    () = return-spmf (Some m, (Fail,
None, as))
| lazy-channel-recv (ms, Some (a′, m′), as) () = (case as m′ of
    None ⇒ do {
      a ← spmf-of-set (rnd η);
      return-spmf (if a = a′ then Some m′ else None, cstate.Fail, None, as (m′ :=
Some a))}
  | Some a ⇒ return-spmf (if a = a′ then Some m′ else None, Fail, None, as))
| lazy-channel-recv s                          () = return-spmf (None, s)

private fun lazy-channel-insec :: (bool list cstate × (bool list × bool list) option ×
(bool list ⇒ bool list option),
  (bool list × bool list) insec-query, (bool list × bool list) option) oracle′ where
  lazy-channel-insec (Void, -, as)      (Edit (a′, m′)) = return-spmf (None, (Collect
m′, Some (a′, m′), as))
| lazy-channel-insec (Store m, -, as)      (Edit (a′, m′)) = return-spmf (None,
(Collect m′, Some (a′, m′), as))
| lazy-channel-insec (Store m, es)         Forward         = return-spmf (None,
(Collect m, es))
| lazy-channel-insec (Store m, es)       Drop            = return-spmf (None, (Fail,
es))
| lazy-channel-insec (Store m, None, as) Look            = (case as m of
    None ⇒ do {
      a ← spmf-of-set (rnd η);
      return-spmf (Some (a, m), Store m, None, as (m := Some a))}
  | Some a ⇒ return-spmf (Some (a, m), Store m, None, as))
```

```
| lazy-channel-insec s                    -                    = return-spmf (None, s)

private fun lazy-channel-recv-f :: (bool list  cstate × (bool list × bool list) option
× (bool list ⇒ bool list option), unit, bool list option) oracle' where
  lazy-channel-recv-f (Collect m, None, as)    () = return-spmf (Some m, (Fail,
None, as))
| lazy-channel-recv-f (ms, Some (a', m'), as) () = (case as m' of
    None ⇒ do {
      a ← spmf-of-set (rnd η);
      return-spmf (None, Fail, None, as (m' := Some a))}
  | Some a ⇒ return-spmf (if a = a' then Some m' else None, Fail, None, as))
| lazy-channel-recv-f s                          () = return-spmf (None, s)

private abbreviation callee-auth-channel where
  callee-auth-channel callee ≡ lift-state-oracle extend-state-oracle (attach-callee callee
auth-channel.auth-oracle)

private abbreviation
  valid-insecQ ≡ {(x :: (bool list × bool list) insec-query). case x of
    ForwardOrEdit (Some (a, m)) ⇒ length a = id' η ∧ length m = id' η
  | - ⇒ True}

private inductive S :: (bool list cstate × (bool list × bool list) option × (bool list
⇒ bool list option)) spmf
  ⇒ ((bool × unit) × (bool list ⇒ bool list option) × (bool list × bool list)  cstate)
spmf ⇒ bool where
  S (return-spmf (Void, None, Map.empty))
     (return-spmf ((False, ()), Map.empty, Void))
| S (return-spmf (Store m, None, Map.empty))
    (map-spmf (λa. ((True, ()), [m ↦ a], Store (a, m))) (spmf-of-set (nlists UNIV
η)))
if length m = id' η
| S (return-spmf (Collect m, None, Map.empty))
      (map-spmf (λa. ((True, ()), [m ↦ a], Collect (a, m))) (spmf-of-set (nlists
UNIV η)))
if length m = id' η
| S (return-spmf (Store m, None, [m ↦ a]))
     (return-spmf ((True, ()), [m ↦ a], Store (a, m)))
if length m = id' η and length a = id' η
| S (return-spmf (Collect m, None, [m ↦ a]))
     (return-spmf ((True, ()), [m ↦ a], Collect (a, m)))
if length m = id' η and length a = id' η
| S (return-spmf (Fail, None, Map.empty))
     (map-spmf (λa. ((True, ()), [m ↦ a], Fail)) (spmf-of-set (nlists UNIV η)))
if length m = id' η
| S (return-spmf (Fail, None, [m ↦ a]))
     (return-spmf ((True, ()), [m ↦ a], Fail))
if length m = id' η and length a = id' η
| S (return-spmf (Collect m', Some (a', m'), Map.empty))
```

(*return-spmf* ((*False*, ()), *Map.empty*, *Collect* (*a'*, *m'*)))
if *length m' = id' η* **and** *length a' = id' η*
| *S* (*return-spmf* (*Collect m'*, *Some* (*a'*, *m'*), [*m* ↦ *a*]))
(*return-spmf* ((*True*, ()), [*m* ↦ *a*], *Collect* (*a'*, *m'*)))
if *length m = id' η* **and** *length a = id' η* **and** *length m' = id' η* **and** *length a' = id' η*
| *S* (*return-spmf* (*Collect m'*, *Some* (*a'*, *m'*), *Map.empty*))
(*map-spmf* (λ*x*. ((*True*, ()), [*m* ↦ *x*], *Collect* (*a'*, *m'*))) (*spmf-of-set* (*nlists UNIV η*)))
if *length m = id' η* **and** *length m' = id' η* **and** *length a' = id' η*
| *S* (*map-spmf* (λ*x*. (*Fail*, *None*, *as*(*m'* ↦ *x*))) *spmf-s*)
(*map-spmf* (λ*x*. ((*False*, ()), *as*(*m'* ↦ *x*), *Fail*)) *spmf-s*)
if *length m' = id' η* **and** *lossless-spmf spmf-s*
| *S* (*map-spmf* (λ*x*. (*Fail*, *None*, *as*(*m'* ↦ *x*))) *spmf-s*)
(*map-spmf* (λ*x*. ((*True*, ()), *as*(*m'* ↦ *x*), *Fail*)) *spmf-s*)
if *length m' = id' η* **and** *lossless-spmf spmf-s*
| *S* (*return-spmf* (*Fail*, *None*, [*m'* ↦ *a'*]))
(*map-spmf* (λ*x*. ((*True*, ()), [*m* ↦ *x*, *m'* ↦ *a'*], *Fail*)) (*spmf-of-set* (*nlists UNIV η*)))
if *length m = id' η* **and** *length m'= id' η* **and** *length a' = id' η*
| *S* (*map-spmf* (λ*x*. (*Fail*, *None*, [*m'* ↦ *x*])) (*spmf-of-set* (*nlists UNIV η* ∩ {*x*. *x* ≠ *a'*})))
(*map-spmf* (λ*x*. ((*True*, ()), [*m* ↦ *fst x*, *m'* ↦ *snd x*], *Fail*)) (*spmf-of-set* (*nlists UNIV η* × *nlists UNIV η* ∩ {*x*. *snd x* ≠ *a'*})))
if *length m = id' η* **and** *length m'= id' η*
| *S* (*map-spmf* (λ*x*. (*Fail*, *None*, *as*(*m'* ↦ *x*))) *spmf-s*)
(*map-spmf* (λ*p*. ((*True*, ()), *as*(*m'* ↦ *fst p*, *m* ↦ *snd p*), *Fail*)) (*mk-lossless* (*pair-spmf spmf-s* (*spmf-of-set* (*nlists UNIV η*)))))
if *length m = id' η* **and** *length m'= id' η* **and** *lossless-spmf spmf-s*

private lemma *trace-eq-lazy*:
assumes *η > 0*
shows (*valid-insecQ* <+> *nlists UNIV* (*id' η*) <+> *UNIV*) ⊢$_R$
RES (*lazy-channel-insec* ⊕$_O$ *lazy-channel-send* ⊕$_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*)
≈
RES (††*insec-channel.insec-oracle* ⊕$_O$ *rorc-channel-send* ⊕$_O$ *rorc-channel-recv*) ((*False*, ()), *Map.empty*, *Void*)
(**is** *?A* ⊢$_R$ *RES* (*?L1* ⊕$_O$ *?L2* ⊕$_O$ *?L3*) *?SL* ≈ *RES* (*?R1* ⊕$_O$ *?R2* ⊕$_O$ *?R3*) *?SR*)
proof −
note [*simp*] =
spmf.map-comp o-def map-bind-spmf bind-map-spmf bind-spmf-const exec-gpv-bind
insec-channel.insec-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps
rorc-channel-send-def rorc-channel-recv-def rorc.rnd-oracle.simps mac-def rnd-def

have *aux1*: *nlists* (*UNIV*::*bool set*) *η* × *nlists* (*UNIV*::*bool set*) *η* ∩ {*x*. *snd x* ≠ *a'*} ≠ {} (**is** *?aux1*)

```
  and aux2: nlists (UNIV::bool set) η ∩ {x. x ≠ a'} ≠ {} (is ?aux2) for a'
 proof −
  have ∃ a. a ∈ nlists (UNIV::bool set) η ∧ a ≠ a' for a'
  proof (cases a' ∈ nlists (UNIV::bool set) η)
   case True
   show ?thesis
   proof (rule ccontr)
    have 2 ^ η = card (nlists (UNIV :: bool set) η) by (simp add: card-nlists)
    also assume ∄ a. a ∈ nlists UNIV η ∧ a ≠ a'
    then have nlists UNIV η = {a'} using True by blast
    then have fct:card (nlists (UNIV :: bool set) η) = card {a'} by simp
    also have  card {a'} = 1 by simp
    finally have η = 0 by simp
    then show False using assms by blast
   qed
  next
   case False
   obtain a where obt:a ∈ nlists (UNIV::bool set) η using assms by auto
   then have a ≠ a' using False by blast
   then show ?thesis using obt by auto
  qed
  then obtain a where o1: a ∈ nlists (UNIV::bool set) η and o2: a ≠ a' by
blast

  obtain m where m ∈ nlists (UNIV::bool set) η by blast
  with o1 o2 have (m, a) ∈ {x. snd x ≠ a'} and (m, a) ∈ nlists UNIV η ×
nlists UNIV η by auto
  then show ?aux1 by blast

  from o1 o2 have a ∈ {x. x ≠ a'} and a ∈ nlists UNIV η by auto
  then show ?aux2 by blast
 qed

 have ∗: ?A ⊢C ?L1 ⊕O ?L2 ⊕O ?L3(?SL) ≈ ?R1 ⊕O ?R2 ⊕O ?R3(?SR)
 proof(rule trace'-eqI-sim[where S=S], goal-cases Init-OK Output-OK State-OK)
  case Init-OK
  then show ?case by (simp add: S.simps)
 next
  case (Output-OK p q query)
  show ?case
  proof (cases query)
   case (Inl adv-query)
   with Output-OK show ?thesis
   proof cases
    case (14 m m' a')
    then show ?thesis using Output-OK(2) Inl
     by(cases adv-query)(simp; subst (1 2) weight-spmf-of-set-finite; auto simp
add: assms aux1 aux2)+
   qed (auto simp add: weight-mk-lossless lossless-spmf-def split: aquery.splits
```

```
option.splits)
    next
      case Snd-Rcv: (Inr query')
      show ?thesis
      proof (cases query')
        case (Inl snd-query)
        with Output-OK Snd-Rcv show ?thesis
        proof cases
          case (11 m' - as)
          with Output-OK(2) Snd-Rcv Inl show ?thesis
            by(cases snd-query = m'; cases as snd-query)(simp-all)
        next
          case (14 m m' a')
          with Output-OK(2) Snd-Rcv Inl show ?thesis
           by(simp; subst (1 2) weight-spmf-of-set-finite; auto simp add: assms aux1
aux2)
        qed (auto simp add: weight-mk-lossless lossless-spmf-def)
      next
        case (Inr rcv-query)
        with Output-OK Snd-Rcv show ?thesis
        proof cases
          case (10 m m' a')
        with Output-OK(2) Snd-Rcv Inr show ?thesis by(cases m = m')(simp-all)
        next
          case (14 m m' a')
          with Output-OK(2) Snd-Rcv Inr show ?thesis
           by(simp; subst (1 2) weight-spmf-of-set-finite; auto simp add: assms aux1
aux2)
        qed (auto simp add: weight-mk-lossless lossless-spmf-def split:option.splits)
      qed
    qed
  next
    case (State-OK p q query state answer state')
    show ?case
    proof (cases query)
      case (Inl adv-query)
      show ?thesis
      proof (cases adv-query)
        case Look
        with State-OK Inl show ?thesis
        proof cases
          case Store-State-Channel: (2 m)
          have[simp]:  a ∈ nlists UNIV η ⟹
          S (cond-spmf-fst (map-spmf (λx. (Inl (Some (x, m)), Store m, None, [m
↦ x]))
              (spmf-of-set (nlists UNIV η))) (Inl (Some (a, m))))
            (cond-spmf-fst (map-spmf (λx. (Inl (Some (x, m)), (True, ()), [m ↦ x],
Store (x, m)))
              (spmf-of-set (nlists UNIV η))) (Inl (Some (a, m)))) for a
```

```
        proof(subst (1 2) cond-spmf-fst-map-Pair1, goal-cases)
          case 4
          then show ?case
            by(subst (1 2 3) inv-into-f-f, simp-all add: inj-on-def)
                 (rule S.intros, simp-all add: Store-State-Channel in-nlists-UNIV
id'-def)
        qed (simp-all add: id'-def inj-on-def)

        from Store-State-Channel show ?thesis using State-OK Inl Look
          by(clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])

      qed (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro: S.intros)
    next
      case (ForwardOrEdit foe)
      with State-OK Inl show ?thesis
      proof (cases foe)
        case (Some am')
        obtain a' m' where am' = (a', m') by (cases am') simp
        with State-OK Inl ForwardOrEdit Some show ?thesis
          by cases (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro:
S.intros elim:S.cases)
       qed (erule S.cases, auto simp add: map-spmf-conv-bind-spmf[symmetric]
intro: S.intros)
    next
      case Drop
      with State-OK Inl show ?thesis
        by cases (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro:
S.intros)
    qed
  next
    case Snd-Rcv: (Inr query')
    show ?thesis
    proof (cases query')
      case (Inl snd-query)
      with State-OK Snd-Rcv show ?thesis
      proof cases
        case 1
        with State-OK Snd-Rcv Inl show ?thesis
          by(clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
            (rule S.intros, simp add: in-nlists-UNIV)
      next
        case (8 m' a')
        with State-OK Snd-Rcv Inl show ?thesis
          by(clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
            (rule S.intros, simp add: in-nlists-UNIV)
      next
        case (11 m' spmf-s as)
        with State-OK Snd-Rcv Inl show ?thesis
          by(auto simp add: bind-bind-conv-pair-spmf split-def split: if-splits
```

```
option.splits  intro!: S.intros)
          ((auto simp add: map-spmf-conv-bind-spmf[symmetric] intro!: S.intros),
simp only: id'-def in-nlists-UNIV)
       qed (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro: S.intros)
     next
       case (Inr rcv-query)
       with State-OK Snd-Rcv show ?thesis
       proof(cases)
         case (8 m' a')
         then show ?thesis using State-OK(2−) Snd-Rcv Inr
           by (auto simp add: map-spmf-conv-bind-spmf[symmetric] image-def
               cond-spmf-fst-def vimage-def aux1 aux2 assms intro:S.intros)
       next
         case (9 m a m' a')
         then show ?thesis using State-OK(2−) Snd-Rcv Inr
          by (cases m = m') (auto simp add: map-spmf-conv-bind-spmf[symmetric]
cond-spmf-fst-def
              vimage-def aux1 aux2 assms intro:S.intros split!: if-splits)
       next
         case (10 m m' a')
         show ?thesis
         proof (cases m = m')
           case True
           with 10 show ?thesis using State-OK(2−) Snd-Rcv Inr
         by (auto simp add: map-spmf-conv-bind-spmf[symmetric] cond-spmf-fst-def

                vimage-def aux1 aux2 assms split!: if-split intro:S.intros)
         next
           case False
          have[simp]: a' ∈ nlists UNIV η ⟹ nlists (UNIV :: bool set) η × nlists
UNIV η ∩ {x. snd x = a'} = nlists UNIV η × {a'}
            by auto

          from 10 False show ?thesis using State-OK(2−) Snd-Rcv Inr
            by(simp add: bind-bind-conv-pair-spmf)
                  ((auto simp add: bind-bind-conv-pair-spmf split-def image-def
map-spmf-conv-bind-spmf[symmetric]
               cond-spmf-fst-def vimage-def cond-spmf-spmf-of-set pair-spmf-of-set
)
               , (auto simp add: pair-spmf-of-set[symmetric] spmf-of-set-singleton
pair-spmf-return-spmf2
               map-spmf-of-set-inj-on[symmetric] simp del: map-spmf-of-set-inj-on
intro: S.intros))
         qed
       qed (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro: S.intros)
     qed
   qed
 qed
```

from ∗ **show** *?thesis* **by** *simp*
qed

private lemma *game-difference*:
defines $\mathcal{I} \equiv \mathcal{I}$*-uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert None* (*Some* ‘ (*nlists UNIV* η × *nlists UNIV* η))) $\oplus_{\mathcal{I}}$
($\mathcal{I}$*-uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *vld* η)))
assumes *bound*: *interaction-bounded-by'* (λ-. *True*) $\mathcal{A}$ *q*
and *lossless*: *plossless-gpv* $\mathcal{I}$ $\mathcal{A}$
and *WT*: $\mathcal{I} \vdash g$ $\mathcal{A}$ √
shows
|*spmf* (*connect* $\mathcal{A}$ (*RES* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv-f*) (*Void*, *None*, *Map.empty*))) *True* −
spmf (*connect* $\mathcal{A}$ (*RES* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*))) *True*|
≤ *q* / *real* (*2* ^ η) (**is** *?LHS* ≤ -)
proof −

define *lazy-channel-insec'* **where**
lazy-channel-insec' = (†*lazy-channel-insec* :: (*bool* × *bool list cstate* × (*bool list* × *bool list*) *option* × (*bool list* ⇒ *bool list option*),
(*bool list* × *bool list*) *insec-query*, (*bool list* × *bool list*) *option*) *oracle'*)

define *lazy-channel-send'* **where**
lazy-channel-send' = (†*lazy-channel-send* :: (*bool* × *bool list cstate* × (*bool list* × *bool list*) *option* × (*bool list* ⇒ *bool list option*),
bool list, *unit*) *oracle'*)

define *lazy-channel-recv'* **where**
lazy-channel-recv' = (λ (*s* :: *bool* × *bool list cstate* × (*bool list* × *bool list*) *option* × (*bool list* ⇒ *bool list option*)) (*q* :: *unit*).
(*case s of*
(*flg*, *Collect m*, *None*, *as*) ⇒ *return-spmf* (*Some m*, (*flg*, *Fail*, *None*, *as*))
| (*flg*, *ms*, *Some* (*a'*, *m'*), *as*) ⇒ (*case as m' of*
None ⇒ *do* {
a ← *spmf-of-set* (*rnd* η);
return-spmf (*if a = a' then Some m' else None*, *flg* ∨ *a = a'*, *Fail*, *None*, *as* (*m'* := *Some a*))}
| *Some a* ⇒ *return-spmf* (*if a = a' then Some m' else None*, *flg*, *Fail*, *None*, *as*))
| - ⇒ *return-spmf* (*None*, *s*)))

define *lazy-channel-recv-f'* **where**
lazy-channel-recv-f' = (λ (*s* :: *bool* × *bool list cstate* × (*bool list* × *bool list*) *option* × (*bool list* ⇒ *bool list option*)) (*q* :: *unit*).
(*case s of*
(*flg*, *Collect m*, *None*, *as*) ⇒ *return-spmf* (*Some m*, (*flg*, *Fail*, *None*, *as*))
| (*flg*, *ms*, *Some* (*a'*, *m'*), *as*) ⇒ (*case as m' of*

```
      None ⇒ do {
        a ← spmf-of-set (rnd η);
        return-spmf (None, flg ∨ a = a', Fail, None, as (m' := Some a))}
    | Some a ⇒ return-spmf (if a = a' then Some m' else None, flg, Fail, None,
as))
  | - ⇒ return-spmf (None, s)))

 define game where
  game = (λ(A :: ((bool list × bool list) insec-query + bool list + unit, (bool list
× bool list) option + unit + bool list option) distinguisher) recv-oracle. do {
  (b :: bool, (flg, ams, es, as))← (exec-gpv (lazy-channel-insec' ⊕O lazy-channel-send'
⊕O recv-oracle) A (False, Void, None, Map.empty));
  return-spmf (b, flg) })

 have fact1: spmf (connect A (RES (lazy-channel-insec ⊕O lazy-channel-send
⊕O lazy-channel-recv-f) (Void, None, Map.empty))) True
    = spmf (connect A (RES (lazy-channel-insec' ⊕O lazy-channel-send' ⊕O
lazy-channel-recv-f') (False, Void, None, Map.empty))) True
 proof −
  let ?orc-lft = lazy-channel-insec ⊕O lazy-channel-send ⊕O lazy-channel-recv-f
 let ?orc-rgt = lazy-channel-insec' ⊕O lazy-channel-send' ⊕O lazy-channel-recv-f'

  have[simp]: rel-spmf (rel-prod (=) (λx y. x = snd y))
    (lazy-channel-insec s q) (lazy-channel-insec' (flg, s) q) for s flg q
     by (cases s) (simp add: lazy-channel-insec'-def spmf-rel-map rel-prod-sel
split-def option.rel-refl pmf.rel-refl split:option.split)

  have[simp]: rel-spmf (rel-prod (=) (λx y. x = snd y))
    (lazy-channel-send s q) (lazy-channel-send' (flg, s) q) for s flg q
  proof −
    obtain ams es as where s = (ams, es, as) by (cases s)
   then show ?thesis by (cases ams) (auto simp add: spmf-rel-map map-spmf-conv-bind-spmf
split-def lazy-channel-send'-def)
  qed

  have[simp]: rel-spmf (rel-prod (=) (λx y. x = snd y))
    (lazy-channel-recv-f s q) (lazy-channel-recv-f' (flg, s) q) for s flg q
  proof −
    obtain ams es as where ∗: s = (ams, es, as) by (cases s)
    show ?thesis
    proof (cases es)
      case None
     with ∗ show ?thesis by (cases ams) (simp-all add:lazy-channel-recv-f'-def)
    next
      case (Some am)
      obtain a m where am = (a, m) by (cases am)
      with ∗ show ?thesis by (cases ams) (simp-all add: lazy-channel-recv-f'-def
rel-spmf-bind-reflI split:option.split)
    qed
```

qed

have[*simp*]: *rel-spmf* (*rel-prod* (=) (λ*x y*. *x* = *snd y*))
((*lazy-channel-insec* ⊕$_O$ *lazy-channel-send* ⊕$_O$ *lazy-channel-recv-f*) (*ams*, *es*, *as*) *query*)
((*lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv-f'*) (*flg*, *ams*, *es*, *as*) *query*) **for** *flg ams es as query*
proof (*cases query*)
case (*Inl adv-query*)
then show *?thesis*
by(*clarsimp simp add*: *spmf-rel-map map-spmf-conv-bind-spmf*[*symmetric*] *apfst-def map-prod-def split-beta*)
((*rule rel-spmf-mono*[**where** *A*=*rel-prod* (=) (λ*x y*. *x* = *snd y*)]), *auto*)
next
case (*Inr query'*)
note *Snd-Rcv* = *this*
then show *?thesis*
by (*cases query'*, *auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf*[*symmetric*] *split-beta*)
((*rule rel-spmf-mono*[**where** *A*=*rel-prod* (=) (λ*x y*. *x* = *snd y*)]); *auto*)+
qed

have[*simp*]: *rel-spmf* (λ*x y*. *fst x* = *fst y*)
(*exec-gpv ?orc-lft* 𝒜 (*Void*, *None*, *Map.empty*)) (*exec-gpv ?orc-rgt* 𝒜 (*False*, *Void*, *None*, *Map.empty*))
by(*rule rel-spmf-mono*[**where** *A*=*rel-prod* (=) (λ*x y*. *x* = *snd y*)])
(*auto simp add*: *gpv.rel-eq split-def intro*!: *rel-funI*
exec-gpv-parametric[**where** *CALL*=(=), *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])

show *?thesis*
unfolding *map-spmf-conv-bind-spmf exec-gpv-resource-of-oracle connect-def spmf-conv-measure-spmf*
by(*rule measure-spmf-parametric*[**where** *A*=(=), *THEN rel-funD*, *THEN rel-funD*])
(*auto simp add*: *rel-pred-def spmf-rel-map map-spmf-conv-bind-spmf*[*symmetric*] *gpv.rel-eq split-def intro*!: *rel-funI*)
qed

have *fact2*: |*spmf* (*connect* 𝒜 (*RES* (*lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv-f'*) (*False*, *Void*, *None*, *Map.empty*))) *True* −
spmf (*connect* 𝒜 (*RES* (*lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv'*) (*False*, *Void*, *None*, *Map.empty*))) *True*|
≤ *Sigma-Algebra.measure* (*measure-spmf* (*game* 𝒜 *lazy-channel-recv-f'*)) {*x*. *snd x*} (**is** |*spmf ?L* - − *spmf ?R* - | ≤ -)
proof −
let *?orc-lft* = *lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv-f'*

let *?orc-rgt* = *lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv'*

have[*simp*]: *callee-invariant-on lazy-channel-insec' fst* (*ℐ-uniform* (*Set.Collect* (*valid-mac-query η*)) *UNIV*)
proof (*unfold-locales*, *goal-cases*)
case (*1 state query answer state'*)
then show *?case*
by(*cases state*; *cases fst* (*snd state*))(*auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf split-def lazy-channel-insec'-def*)
qed (*auto intro*: *WT-calleeI*)

have[*simp*]: *callee-invariant-on lazy-channel-send' fst* (*ℐ-uniform* (*vld η*) *UNIV*)
proof (*unfold-locales*, *goal-cases*)
case (*1 state query answer state'*)
then show *?case*
by(*cases state*; *cases fst* (*snd state*))(*auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf split-def lazy-channel-send'-def*)
qed (*auto intro*: *WT-calleeI*)

have[*simp*]: *callee-invariant lazy-channel-recv' fst*
proof (*unfold-locales*, *goal-cases*)
case (*1 state query answer state'*)
then show *?case*
by(*cases state*; *cases fst* (*snd state*))(*auto simp add*: *lazy-channel-recv'-def split*:*option.splits*)
qed (*auto split*:*option.splits*)

have[*simp*]: *callee-invariant lazy-channel-recv-f' fst*
proof (*unfold-locales*, *goal-cases*)
case (*1 state query answer state'*)
then show *?case*
by(*cases state*; *cases fst* (*snd state*))(*auto simp add*: *lazy-channel-recv-f'-def split*:*option.splits*)
qed (*auto split*:*option.splits*)

have[*simp*]: *lossless-spmf* (*lazy-channel-insec s q*) **for** *s q*
by(*cases* (*s*, *q*) *rule*: *lazy-channel-insec.cases*)(*auto simp add*: *rnd-def split*!: *option.split*)

have[*simp*]: *lossless-spmf* (*lazy-channel-send' s q*) **for** *s q*
by(*cases s*; *cases fst* (*snd s*))(*simp-all add*: *lazy-channel-send'-def*)

have[*simp*]: *lossless-spmf* (*lazy-channel-recv' s* ()) **for** *s*
by(*auto simp add*: *lazy-channel-recv'-def rnd-def split*: *prod.split cstate.split option.split*)

have[*simp*]: *lossless-spmf* (*lazy-channel-recv-f' s* ()) **for** *s*
by(*auto simp add*: *lazy-channel-recv-f'-def rnd-def split*: *prod.split cstate.split option.split*)

have[*simp*]: *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). (*fst s2'* ⟶ *fst s1'*) ∧ (¬ *fst s2'* ⟶ ¬ *fst s1'* ∧ *a* = *b* ∧ *s1'* = *s2'*))
(*?orc-lft* (*flg*, *ams*, *es*, *as*) *query*) (*?orc-rgt* (*flg*, *ams*, *es*, *as*) *query*) **for** *flg ams es as query*
proof (*cases query*)
case (*Inl adv-query*)
then show *?thesis* **by** (*auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf intro*: *rel-spmf-bind-reflI*)
next
case (*Inr query'*)
note *Snd-Rcv* = *this*
show *?thesis*
proof (*cases query'*)
case (*Inl snd-query*)
with *Snd-Rcv* **show** *?thesis*
by (*auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf intro*: *rel-spmf-bind-reflI*)
next
case (*Inr rcv-query*)
with *Snd-Rcv* **show** *?thesis*
proof (*cases es*)
case (*Some am'*)
obtain *a' m'* **where** *am'* = (*a'*, *m'*) **by** (*cases am'*)
with *Snd-Rcv Some Inr* **show** *?thesis*
by (*cases ams*) (*auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf lazy-channel-recv'-def lazy-channel-recv-f'-def rel-spmf-bind-reflI split*:*option.splits*)
qed (*cases ams*; *auto simp add*: *lazy-channel-recv'-def lazy-channel-recv-f'-def split*:*option.splits*)
qed
qed
let *?𝓘* = *𝓘-uniform* (*Set.Collect* (*valid-mac-query* η)) *UNIV* ⊕*𝓘* (*𝓘-uniform* (*vld* η) *UNIV* ⊕*𝓘* *𝓘-full*)
let *?𝓐* = *mk-lossless-gpv* (*responses-𝓘 𝓘*) *True* *𝓐*

have *plossless-gpv ?𝓘 ?𝓐* **using** *lossless WT*
by(*rule plossless-gpv-mk-lossless-gpv*)(*simp add*: *𝓘-def*)
moreover have *?𝓘* ⊢*g ?𝓐* √ **using** *WT* **by**(*rule WT-gpv-mk-lossless-gpv*)(*simp add*: *𝓘-def*)
ultimately have *rel-spmf* (λ*x y*. *fst* (*snd x*) = *fst* (*snd y*) ∧ (¬ *fst* (*snd y*) ⟶ (*fst x* ⟷ *fst y*)))
(*exec-gpv ?orc-lft ?𝓐* (*False*, *Void*, *None*, *Map.empty*)) (*exec-gpv ?orc-rgt ?𝓐* (*False*, *Void*, *None*, *Map.empty*))
by(*auto simp add*: *𝓘-def intro*!: *exec-gpv-oracle-bisim-bad-plossless*[**where** *X*=(=) **and**
X-bad=λ - -. *True* **and** *?bad1.0*=*fst* **and** *?bad2.0*=*fst* **and** *𝓘* = *?𝓘*, *THEN rel-spmf-mono*])
(*auto simp add*: *lazy-channel-insec'-def intro*: *WT-calleeI*)
also let *?I* = (λ(-, *s1*, *s2*, *s3*). *pred-cstate* (λ*x*. *length x* = η) *s1* ∧ *pred-option* (λ(*x*, *y*). *length x* = η ∧ *length y* = η) *s2* ∧ *ran s3* ⊆ *nlists UNIV* η)

have *callee-invariant-on* (*lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv-f′*) *?I* $\mathcal{I}$
apply(*unfold-locales*)
subgoal for *s x y s′*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *elim PlusE*; *clarsimp simp add*: *lazy-channel-insec′-def lazy-channel-send′-def lazy-channel-recv-f′-def*)
subgoal for - - - - *x′*
by(*cases* (*snd s*, *x′*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def split*: *option.split-asm*)
subgoal by(*cases* (*snd s*, *projl* (*projr x*)) *rule*: *lazy-channel-send.cases*)(*auto simp add*: *vld-def in-nlists-UNIV*)
subgoal by(*cases* (*snd s*, ()) *rule*: *lazy-channel-recv-f.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm cstate.split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def option.pred-set*)
done
subgoal for *s*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *intro conjI WT-calleeI*; *clarsimp simp add*: *lazy-channel-insec′-def lazy-channel-send′-def lazy-channel-recv-f′-def*)
subgoal for - - - - *x*
by(*cases* (*snd s*, *x*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def ran-def split*: *option.split-asm*)
subgoal by(*cases* (*snd s*, ()) *rule*: *lazy-channel-recv-f.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm cstate.split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def*)
done
done
then have *exec-gpv ?orc-lft ?*$\mathcal{A}$ (*False*, *Void*, *None*, *Map.empty*) = *exec-gpv ?orc-lft* $\mathcal{A}$ (*False*, *Void*, *None*, *Map.empty*)
using *WT* **by**(*rule callee-invariant-on.exec-gpv-mk-lossless-gpv*) *simp*
also have *callee-invariant-on* (*lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv′*) *?I* $\mathcal{I}$
apply(*unfold-locales*)
subgoal for *s x y s′*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *elim PlusE*; *clarsimp simp add*: *lazy-channel-insec′-def lazy-channel-send′-def lazy-channel-recv′-def*)
subgoal for - - - - *x′*
by(*cases* (*snd s*, *x′*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def split*: *option.split-asm*)
subgoal by(*cases* (*snd s*, *projl* (*projr x*)) *rule*: *lazy-channel-send.cases*)(*auto simp add*: *vld-def in-nlists-UNIV*)
subgoal by(*cases* (*snd s*, ()) *rule*: *lazy-channel-recv.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm cstate.split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def option.pred-set*)
done
subgoal for *s*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *intro conjI WT-calleeI*; *clarsimp simp add*: *lazy-channel-insec′-def lazy-channel-send′-def lazy-channel-recv′-def*)
subgoal for - - - - *x*

by(*cases* (*snd s*, *x*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def ran-def split*: *option.split-asm*)
subgoal by(*cases* (*snd s*, ()) *rule*: *lazy-channel-recv.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm cstate.split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def*)
done
done
then have *exec-gpv ?orc-rgt ?*$\mathcal{A}$ (*False*, *Void*, *None*, *Map.empty*) = *exec-gpv ?orc-rgt* $\mathcal{A}$ (*False*, *Void*, *None*, *Map.empty*)
using *WT* **by**(*rule callee-invariant-on.exec-gpv-mk-lossless-gpv*) *simp*
finally have |*Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv-f'*)) {*x*. *fst x*}
− *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv'*)) {*x*. *fst x*}|
≤ *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv-f'*)) {*x*. *snd x*}
unfolding *game-def*
by − (*rule fundamental-lemma*[**where** *?bad2.0*=*snd*]; *auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf*[*symmetric*] *split-def*)

moreover have *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv-f'*)) {*x*. *fst x*} = *spmf ?L True*
unfolding *game-def*
by(*auto simp add*: *map-spmf-conv-bind-spmf exec-gpv-resource-of-oracle connect-def spmf-conv-measure-spmf*)
(*auto simp add*: *rel-pred-def intro*!: *rel-spmf-bind-reflI measure-spmf-parametric*[**where** *A*=λ*l r*. *fst l* ⟷ *r*, *THEN rel-funD*, *THEN rel-funD*])

moreover have *spmf ?R True* = *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv'*)) {*x*. *fst x*}
unfolding *game-def*
by(*auto simp add*: *map-spmf-conv-bind-spmf exec-gpv-resource-of-oracle connect-def spmf-conv-measure-spmf*)
(*auto simp add*: *rel-pred-def intro*!: *rel-spmf-bind-reflI measure-spmf-parametric*[**where** *A*=λ*l r*. *l* ⟷ *fst r*, *THEN rel-funD*, *THEN rel-funD*])

ultimately show *?thesis* **by** *simp*
qed

have *fact3*: *spmf* (*connect* $\mathcal{A}$ (*RES* (*lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv'*) (*False*, *Void*, *None*, *Map.empty*))) *True*
= *spmf* (*connect* $\mathcal{A}$ (*RES* (*lazy-channel-insec* ⊕$_O$ *lazy-channel-send* ⊕$_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*))) *True*
proof −
let *?orc-lft* = *lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv'*
let *?orc-rgt* = *lazy-channel-insec* ⊕$_O$ *lazy-channel-send* ⊕$_O$ *lazy-channel-recv*

have[*simp*]: *rel-spmf* (*rel-prod* (=) (λ*x y*. *y* = *snd x*))

```
    (lazy-channel-insec' (flg, s) q) (lazy-channel-insec s q) for s flg q
      by (cases s) (simp add: lazy-channel-insec'-def spmf-rel-map rel-prod-sel
split-def option.rel-refl pmf.rel-refl split:option.split)

  have[simp]: rel-spmf (rel-prod (=) (λx y. y = snd x))
    (lazy-channel-send' (flg, s) q) (lazy-channel-send s q) for s flg q
     by(cases (s, q) rule: lazy-channel-send.cases)(auto simp add: spmf-rel-map
map-spmf-conv-bind-spmf split-def lazy-channel-send'-def)

  have[simp]: rel-spmf (rel-prod (=) (λx y. y = snd x))
    (lazy-channel-recv' (flg, s) q) (lazy-channel-recv s q) for s flg q
   by(cases (s, q) rule: lazy-channel-recv.cases)(auto simp add: lazy-channel-recv'-def
rel-spmf-bind-reflI split:option.split cstate.split)

  have[simp]: rel-spmf (rel-prod (=) (λx y. y = snd x))
   ((lazy-channel-insec' ⊕O lazy-channel-send' ⊕O lazy-channel-recv') (flg, ams,
es, as) query)
   ((lazy-channel-insec ⊕O lazy-channel-send ⊕O lazy-channel-recv) (ams, es, as)
query) for flg ams es as query
  proof (cases query)
    case (Inl adv-query)
    then show ?thesis
        by(auto simp add: spmf-rel-map map-spmf-conv-bind-spmf[symmetric]
apfst-def map-prod-def split-beta intro: rel-spmf-mono[where A=rel-prod (=) (λx
y. y = snd x)])
  next
    case (Inr query')
    note Snd-Rcv = this
    then show ?thesis
   by (cases query', auto simp add: spmf-rel-map map-spmf-conv-bind-spmf[symmetric]
split-beta)
       ((rule rel-spmf-mono[where A=rel-prod (=) (λx y. y = snd x)]); auto)+
  qed

  have[simp]: rel-spmf (λx y. fst x = fst y)
    (exec-gpv ?orc-lft A (False, Void, None, Map.empty)) (exec-gpv ?orc-rgt A
(Void, None, Map.empty))
    by(rule rel-spmf-mono[where A=rel-prod (=) (λx y. y = snd x)])
     (auto simp add: gpv.rel-eq split-def intro!: rel-funI
      exec-gpv-parametric[where CALL=(=), THEN rel-funD, THEN rel-funD,
THEN rel-funD])

  show ?thesis
    unfolding map-spmf-conv-bind-spmf exec-gpv-resource-of-oracle connect-def
spmf-conv-measure-spmf
     by(rule measure-spmf-parametric[where A=(=), THEN rel-funD, THEN
rel-funD])
    (auto simp add: rel-pred-def spmf-rel-map map-spmf-conv-bind-spmf[symmetric]
gpv.rel-eq split-def intro!: rel-funI)
```

qed

have *fact4*: *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv-f'*))
{*x*. *snd x*} ≤ *q* / *real* (2 ^ *η*)
proof −
let *?orc-sum* = *lazy-channel-insec'* ⊕$_O$ *lazy-channel-send'* ⊕$_O$ *lazy-channel-recv-f'*

have *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv-f'*))
{*x*. *snd x*}
= *spmf* (*map-spmf* (*fst* ∘ *snd*) (*exec-gpv ?orc-sum* $\mathcal{A}$ (*False*, *Void*, *None*,
Map.empty))) *True*
unfolding *game-def*
by (*simp add*: *split-def map-spmf-conv-bind-spmf*[*symmetric*] *measure-map-spmf*)
(*simp add*: *spmf-conv-measure-spmf measure-map-spmf vimage-def*)
also have ... ≤ (*1* / *real* (*card* (*nlists* (*UNIV* :: *bool set*) *η*))) ∗ *real q*
proof −
have ∗: *spmf* (*map-spmf* (*fst* ∘ *snd*) (*?orc-sum* (*False*, *ams*, *es*, *as*) *query*))
True
≤ *1* / *real* (*card* (*nlists* (*UNIV* :: *bool set*) *η*)) **for** *ams es as*
query
proof (*cases query*)
case (*Inl adv-query*)
then show *?thesis*
by(*cases* ((*ams*, *es*, *as*), *adv-query*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *lazy-channel-insec'-def rnd-def map-spmf-conv-bind-spmf*
bind-spmf-const split: *option.split*)
next
case (*Inr query'*)
note *Snd-Rcv* = *this*
then show *?thesis*
proof (*cases query'*)
case (*Inr rcv-query*)
with *Snd-Rcv* **show** *?thesis*
by (*cases ams*, *auto simp add*: *lazy-channel-recv-f'-def map-spmf-conv-bind-spmf*
split-def split:*option.split*)
(*auto simp add*: *spmf-of-set map-spmf-conv-bind-spmf*[*symmetric*] *rnd-def*
spmf-map vimage-def spmf-conv-measure-spmf[*symmetric*] *split*: *split-indicator*)
qed (*cases ams*, *simp-all add*: *lazy-channel-send'-def*)
qed

show *?thesis* **by** (*rule oi-True.interaction-bounded-by-exec-gpv-bad*[**where**
bad=*fst*]) (*auto simp add*: ∗ *assms*)
qed

also have ... = *1* / *real* (2 ^ *η*) ∗ *real q* **by** (*simp add*: *card-nlists*)
finally show *?thesis* **by** *simp*
qed

have *?LHS* ≤ *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv-f'*))

$\{x.\ snd\ x\}$
  using *fact1 fact2 fact3* **by** *arith*
 also note *fact4*
 finally show *?thesis* **.**
qed

private inductive S' :: $(((bool\ list \times bool\ list)\ option + unit) \times unit \times bool\ list\ cstate)\ spmf \Rightarrow$
  $(bool\ list\ cstate \times (bool\ list \times bool\ list)\ option \times (bool\ list \Rightarrow bool\ list\ option))\ spmf \Rightarrow bool$ **where**
  $S'\ (return\text{-}spmf\ (Inl\ None,\ (),\ Void))$
    $(return\text{-}spmf\ (Void,\ None,\ Map.empty))$
| $S'\ (return\text{-}spmf\ (Inl\ None,\ (),\ Store\ m))$
    $(return\text{-}spmf\ (Store\ m,\ None,\ Map.empty))$
if $length\ m = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Collect\ m))$
    $(return\text{-}spmf\ (Collect\ m,\ None,\ Map.empty))$
if $length\ m = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inl\ (Some\ (a,\ m)),\ (),\ Store\ m))$
    $(return\text{-}spmf\ (Store\ m,\ None,\ [m \mapsto a]))$
if $length\ m = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Collect\ m))$
    $(return\text{-}spmf\ (Collect\ m,\ None,\ [m \mapsto a]))$
if $length\ m = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Fail))$
    $(return\text{-}spmf\ (Fail,\ None,\ Map.empty))$
| $S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Fail))$
    $(return\text{-}spmf\ (Fail,\ None,\ [m \mapsto x]))$
if $length\ m = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Void))$
    $(return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ Map.empty))$
if $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Fail))$
    $(return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ Map.empty))$
if $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Store\ m))$
    $(return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ Map.empty))$
if $length\ m = id'\ \eta$ **and** $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inl\ (Some\ (a',\ m')),\ (),\ Collect\ m'))$
    $(return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ [m' \mapsto a']))$
if $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$

| $S'\ (return\text{-}spmf\ (Inl\ None,\ (),\ cstate.Collect\ m))$
    $(return\text{-}spmf\ (cstate.Collect\ m,\ None,\ Map.empty))$
if $length\ m = id'\ \eta$
| $S'\ (return\text{-}spmf\ (Inl\ None,\ (),\ cstate.Fail))$
    $(return\text{-}spmf\ (cstate.Fail,\ None,\ Map.empty))$

| S' (*return-spmf* (*Inr* (), (), *Fail*))
(*return-spmf* (*Collect* m', *Some* (a', m'), [$m \mapsto a$]))
if *length* m = *id'* η **and** *length* m' = *id'* η **and** *length* a' = *id'* η **and** $m \neq m'$
| S' (*return-spmf* (*Inr* (), (), *Fail*))
(*return-spmf* (*Collect* m', *Some* (a', m'), [$m \mapsto a$]))
if *length* m = *id'* η **and** *length* m' = *id'* η **and** *length* a' = *id'* η **and** $a \neq a'$
| S' (*return-spmf* (*Inl None*, (), *Collect* m'))
(*return-spmf* (*Collect* m', *Some* (a', m'), [$m' \mapsto a'$]))
if *length* m' = *id'* η **and** *length* a' = *id'* η
| S' (*return-spmf* (*Inr* (), (), *Collect* m'))
(*return-spmf* (*Collect* m', *Some* (a', m'), [$m' \mapsto a'$]))
if *length* m' = *id'* η **and** *length* a' = *id'* η
| S' (*return-spmf* (*Inr* (), (), *Void*))
(*map-spmf* ($\lambda a'$. (*Fail*, *None*, [$m' \mapsto a'$])) (*spmf-of-set* (*nlists UNIV* η)))
if *length* m' = *id'* η
| S' (*return-spmf* (*Inr* (), (), *Fail*))
(*map-spmf* ($\lambda a'$. (*Fail*, *None*, [$m' \mapsto a'$])) (*spmf-of-set* (*nlists UNIV* η)))
if *length* m' = *id'* η
| S' (*return-spmf* (*Inr* (), (), *Store* m))
(*map-spmf* ($\lambda a'$. (*Fail*, *None*, [$m' \mapsto a'$])) (*spmf-of-set* (*nlists UNIV* η)))
if *length* m = *id'* η **and** *length* m' = *id'* η
| S' (*return-spmf* (*Inr* (), (), *Fail*))
(*map-spmf* ($\lambda a'$. (*Fail*, *None*, [$m \mapsto a$, $m' \mapsto a'$])) (*spmf-of-set* (*nlists UNIV* η)))
if *length* m = *id'* η **and** *length* m' = *id'* η **and** $m \neq m'$
| S' (*return-spmf* (*Inl* (*Some* (a', m')), (), *Fail*))
(*return-spmf* (*Fail*, *None*, [$m' \mapsto a'$]))
if *length* m' = *id'* η **and** *length* a' = *id'* η
| S' (*return-spmf* (*Inl None*, (), *Fail*))
(*return-spmf* (*Fail*, *None*, [$m' \mapsto a'$]))
if *length* m' = *id'* η **and** *length* a' = *id'* η

private lemma *trace-eq-sim*:
shows (*valid-insecQ* <+> *nlists UNIV* (*id'* η) <+> *UNIV*) $\vdash_R$
RES (*callee-auth-channel sim* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*) (*Inl None*, (), *Void*)
$\approx$
RES (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv-f*) (*Void*, *None*, *Map.empty*)
(**is** *?A* $\vdash_R$ *RES* (*?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*) *?SL* $\approx$ *RES* (*?R1* $\oplus_O$ *?R2* $\oplus_O$ *?R3*) *?SR*)
proof −
note [*simp*] =
spmf.map-comp o-def map-bind-spmf bind-map-spmf bind-spmf-const exec-gpv-bind auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps rorc-channel-send-def rorc-channel-recv-def rnd-def

have ∗: *?A* $\vdash_C$ *?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*(*?SL*) $\approx$ *?R1* $\oplus_O$ *?R2* $\oplus_O$ *?R3*(*?SR*)

```
proof(rule trace'-eqI-sim[where S=S'], goal-cases Init-OK Output-OK State-OK)
  case Init-OK
  then show ?case by (rule S'.intros)
next
  case (Output-OK p q query)
  show ?case
  proof (cases query)
    case (Inl adv-query)
    with Output-OK show ?thesis
    proof (cases adv-query)
      case (ForwardOrEdit foe)
      with Output-OK Inl show ?thesis
      proof (cases foe)
        case (Some am')
        obtain a' m' where am' = (a', m') by (cases am') simp
        with Output-OK Inl ForwardOrEdit Some show ?thesis
          by cases (simp-all add: id'-def)
      qed (erule S'.cases, simp-all add: id'-def)
    qed (erule S'.cases, simp-all add: id'-def)+
  next
    case (Inr query')
    with Output-OK show ?thesis by (cases; cases query') (simp-all)
  qed
next
  case (State-OK p q query state answer state')
  show ?case
  proof (cases query)
    case (Inl adv-query)
    show ?thesis
    proof (cases adv-query)
      case Look
      with State-OK Inl show ?thesis
      proof cases
        case (2 m)
          have S' (return-spmf (Inl (Some (x, m)), (), Store m)) (return-spmf
(Store m, None, [m ↦ x])) for x
          by (rule S'.intros) (simp only: 2)
        with 2 show ?thesis using State-OK(2−) Inl Look
            by clarsimp (simp add: cond-spmf-spmf-of-set spmf-of-set-singleton
map-spmf-conv-bind-spmf[symmetric]
            split-beta cond-spmf-fst-def image-def vimage-def id'-def)
    qed (auto simp add: map-spmf-const[symmetric] map-spmf-conv-bind-spmf[symmetric]
image-def intro: S'.intros; simp add: id'-def)+
    next
      case (ForwardOrEdit foe)
      show ?thesis
      proof (cases foe)
        case None
        with State-OK Inl ForwardOrEdit show ?thesis
```

```
      by cases(auto simp add: map-spmf-const[symmetric] map-spmf-conv-bind-spmf[symmetric]
image-def S'.intros)
    next
      case (Some am')
      obtain a' m' where [simp]: am' = (a', m') by (cases am')
      from State-OK Inl ForwardOrEdit Some show ?thesis
      proof cases
        case (4 m a)
        then show ?thesis using State-OK(2−) Inl ForwardOrEdit Some
            by (auto simp add: if-distrib-exec-gpv if-distrib-map-spmf split-def
image-def if-distrib[symmetric] intro: S'.intros cong: if-cong)
      qed (auto simp add: map-spmf-const[symmetric] map-spmf-conv-bind-spmf[symmetric]
image-def intro:S'.intros)
    qed
  next
    case Drop
    with State-OK Inl show ?thesis
    by cases (auto simp add: map-spmf-const[symmetric] map-spmf-conv-bind-spmf[symmetric]
image-def intro:S'.intros; simp add: id'-def)+
  qed
next
  case Snd-Rcv: (Inr query')
  with State-OK show ?thesis
    by(cases; cases query')
   (auto simp add: map-spmf-const[symmetric] map-spmf-conv-bind-spmf[symmetric]
image-def;
      (rule S'.intros; simp add: in-nlists-UNIV id'-def))+
 qed
qed

from ∗ show ?thesis by simp
qed

private lemma real-resource-wiring: macode.res (rnd η) (mac η) =
  RES (††insec-channel.insec-oracle ⊕O rorc-channel-send ⊕O rorc-channel-recv)
((False, ()), Map.empty, Void)
  (is ?L = ?R) including lifting-syntax
proof −
  have ∗: (λx y. map-spmf (map-prod id lprodr) ((A ⊕O B) (rprodl x) y))
        = (λx yl. map-spmf (λp. ((), lprodr (snd p))) (A (rprodl x) yl)) ⊕O
          (λx yr. map-spmf (λp. (fst p, lprodr (snd p))) (B (rprodl x) yr)) for A
B C
  proof −
    have aux: map-prod id g ∘ apfst h = apfst h ∘ map-prod id g for f g h by auto
    show ?thesis
      by (auto simp add: aux plus-oracle-def spmf.map-comp[where f=apfst -,
symmetric]
          spmf.map-comp[where f=map-prod id lprodr] sum.case-distrib[where
h=map-spmf -] cong:sum.case-cong split!:sum.splits)
```

```
    (subst plus-oracle-def[symmetric], simp add: map-prod-def split-def)
  qed

  have ?L = RES (††insec-channel.insec-oracle ⊕O (rprodl ---> id --->
map-spmf (map-prod id lprodr))
        (lift-state-oracle extend-state-oracle (attach-callee
         (λs m. if s
          then Done ((), True)
          else do {
           r ← Pause (Inl m) Done;
           a ← lift-spmf (mac η (projl r) m);
           - ← Pause (Inr (a, m)) Done;
         ( Done ((), True))}) ((rorc.rnd-oracle (rnd η))† ⊕O †channel.send-oracle))
⊕O
         ††(λs q. do {
          (amo, s') ← †channel.recv-oracle s ();
          case amo of
           None ⇒ return-spmf (None, s')
          | Some (a, m) ⇒ do {
             (r, s'') ← (rorc.rnd-oracle (rnd η))† s' m;
             a'← mac η r m;
            return-spmf (if a' = a then Some m else None, s'')}}))) ((False, ()),
Map.empty, Void)
  proof −
   note[simp] = attach-CNV-RES attach-callee-parallel-intercept attach-stateless-callee
    resource-of-oracle-rprodl Rel-def prod.rel-eq[symmetric] extend-state-oracle-plus-oracle[symmetric]
     conv-callee-parallel[symmetric] conv-callee-parallel-id-left[where s=(), sym-
metric]
    o-def bind-map-spmf exec-gpv-bind split-def option.case-distrib[where h=λgpv.
exec-gpv - gpv -]

    show ?thesis
     unfolding channel.res-def rorc.res-def macode.res-def macode.routing-def
      macode.πE-def macode.enm-def macode.dem-def interface-wiring
    by (subst lift-state-oracle-extend-state-oracle | auto cong del: option.case-cong-weak
intro: extend-state-oracle-parametric)+
  qed

  also have ... = ?R
   unfolding rorc-channel-send-def rorc-channel-recv-def extend-state-oracle-def
  by(simp add: ∗ split-def o-def map-fun-def spmf.map-comp extend-state-oracle-def
lift-state-oracle-def
      exec-gpv-bind if-distrib[where f=λgpv. exec-gpv - gpv -] cong: if-cong)
   (simp add: split-def o-def rprodl-def Pair-fst-Unity bind-map-spmf map-spmf-bind-spmf

      if-distrib[where f=map-spmf -] option.case-distrib[where h=map-spmf -]
cong: if-cong option.case-cong)

  finally show ?thesis .
```

qed

private lemma *ideal-resource-wiring*: (*CNV callee s*) $|=_{1_C} \triangleright$ *channel.res auth-channel.auth-oracle* =
RES (*callee-auth-channel callee* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*) (*s*, (), *Void*) (**is** *?L1* $\triangleright$ - = *?R*)
proof −
have[*simp*]: $\mathcal{I}$*-full*, $\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\vdash_C$ *?L1* $\sim$ *?L1* (**is** -, *?I* $\vdash_C$ - $\sim$ -)
by(*rule eq-*$\mathcal{I}$*-converter-mono*)
(*rule parallel-converter2-eq-*$\mathcal{I}$*-cong eq-*$\mathcal{I}$*-converter-reflI WT-converter-*$\mathcal{I}$*-full* $\mathcal{I}$*-full-le-plus-*$\mathcal{I}$ *order-refl plus-*$\mathcal{I}$*-mono*)+

have[*simp*]: *?I* $\vdash c$ (*auth-channel.auth-oracle* $\oplus_O$ *channel.send-oracle* $\oplus_O$ *channel.recv-oracle*) *s* $\surd$ **for** *s*
by(*rule WT-plus-oracleI WT-parallel-oracle WT-callee-full*; (*unfold split-paired-all*) *?*)+

have[*simp*]: *?L1* $\triangleright$ *RES* (*auth-channel.auth-oracle* $\oplus_O$ *channel.send-oracle* $\oplus_O$ *channel.recv-oracle*) *Void* = *?R*
by(*simp add*: *conv-callee-parallel-id-right*[**where** *s'*=(), *symmetric*] *attach-CNV-RES*

attach-callee-parallel-intercept resource-of-oracle-rprodl extend-state-oracle-plus-oracle)

show *?thesis* **unfolding** *channel.res-def*
by (*subst eq-*$\mathcal{I}$*-attach*[*OF WT-resource-of-oracle*, **where** $\mathcal{I}'$ = *?I* **and** *conv*=*?L1* **and** *conv'*=*?L1*]) *simp-all*
qed

lemma *all-together*:
defines $\mathcal{I} \equiv \mathcal{I}$*-uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert None* (*Some* ‘ (*nlists UNIV* η × *nlists UNIV* η))) $\oplus_{\mathcal{I}}$
($\mathcal{I}$*-uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *vld* η)))
assumes $\eta > 0$
and *interaction-bounded-by'* (λ-. *True*) ($\mathcal{A}$ η) *q*
and *lossless*: *plossless-gpv* $\mathcal{I}$ ($\mathcal{A}$ η)
and *WT*: $\mathcal{I} \vdash g$ $\mathcal{A}$ η $\surd$
shows
|*spmf* (*connect* ($\mathcal{A}$ η) (*CNV sim* (*Inl None*) $|=_{1_C} \triangleright$ *channel.res auth-channel.auth-oracle*)) *True* −
spmf (*connect* ($\mathcal{A}$ η) (*macode.res* (*rnd* η) (*mac* η))) *True*| ≤ *q* / *real* (2 ^ η)
proof −
have ∗: ∀ *a b*. *ma* = *Edit* (*a*, *b*) ⟶ *length a* = η ∧ *length b* = η ⟹ *valid-mac-query* η *ma* **for** *ma a b*
by(*cases* (η, *ma*) *rule*: *valid-mac-query.cases*)(*auto simp add*: *vld-def in-nlists-UNIV*)

have *assm4-alt*: *outs-gpv* $\mathcal{I}$ ($\mathcal{A}$ η) ⊆ *valid-insecQ* <+> *nlists UNIV* (*id'* η) <+> *UNIV*
using *assms*(*5*)[*THEN WT-gpv-outs-gpv*] **unfolding** *id'-def*

by(*rule ord-le-eq-trans*) (*auto simp add*: ∗ *split*: *aquery.split option.split simp add*: *in-nlists-UNIV vld-def* $\mathcal{I}$*-def*)

have *callee-invariant-on* (*callee-auth-channel sim* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*)
(λ(*s1*, -, *s3*). (∀ *x y*. *s1* = *Inl* (*Some* (*x*, *y*)) ⟶ *length x* = η ∧ *length y* = η) ∧ *pred-cstate* (λ*x*. *length x* = η) *s3*) $\mathcal{I}$
apply *unfold-locales*
subgoal for *s x y s'*
apply(*cases* (*fst s*, *projl x*) *rule*: *sim.cases*; *cases snd* (*snd s*))
apply(*fastforce simp*: *channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce simp*: *channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce simp*: *channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce simp*: *channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*auto split*: *if-split-asm simp add*: *exec-gpv-bind auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*auto split*: *if-split-asm simp add*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)[*1*]
apply(*fastforce split*: *if-split-asm simp add*: *exec-gpv-bind auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)+
done
subgoal for *s*
apply(*rule WT-calleeI*)
subgoal for *x*
by(*cases* (*fst s*, *projl x*) *rule*: *sim.cases*; *cases snd* (*snd s*))
(*auto split*: *if-split-asm simp add*: *exec-gpv-bind auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* $\mathcal{I}$*-def*)
done
done
then have *WT1*: $\mathcal{I}$ ⊢*res RES* (*callee-auth-channel sim* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*) (*Inl None*, (), *Void*) √
by(*rule callee-invariant-on.WT-resource-of-oracle*) *simp*

have *callee-invariant-on* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv-f*)
(λ(*s1*, *s2*, *s3*). *pred-cstate* (λ*x*. *length x* = η) *s1* ∧ *pred-option* (λ(*x*, *y*). *length x* = η ∧ *length y* = η) *s2* ∧ *ran s3* ⊆ *nlists UNIV* η)
$\mathcal{I}$

apply *unfold-locales*
subgoal for *s x y s'*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *elim PlusE*; *clarsimp*)
subgoal for - - - *x'*
by(*cases* (*s*, *x'*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def split*: *option.split-asm*)
subgoal by(*cases* (*s*, *projl* (*projr x*)) *rule*: *lazy-channel-send.cases*)(*auto simp add*: *vld-def in-nlists-UNIV*)
subgoal by(*cases* (*s*, ()) *rule*: *lazy-channel-recv-f.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def*)
done
subgoal for *s*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *intro conjI WT-calleeI*; *clarsimp*)
subgoal for - - - *x*
by(*cases* (*s*, *x*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def ran-def split*: *option.split-asm*)
subgoal by(*cases* (*s*, ()) *rule*: *lazy-channel-recv-f.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def*)
done
done
then have *WT2*: $\mathcal{I} \vdash_{res}$ *RES* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv-f*) (*Void*, *None*, *Map.empty*) $\surd$
by(*rule callee-invariant-on.WT-resource-of-oracle*) *simp*

have *callee-invariant-on* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv*)
(λ(*s1*, *s2*, *s3*). *pred-cstate* (λx. *length x* = η) *s1* $\wedge$ *pred-option* (λ(*x*, *y*). *length x* = η $\wedge$ *length y* = η) *s2* $\wedge$ *ran s3* $\subseteq$ *nlists UNIV* η)
$\mathcal{I}$
apply *unfold-locales*
subgoal for *s x y s'*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *elim PlusE*; *clarsimp*)
subgoal for - - - *x'*
by(*cases* (*s*, *x'*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def split*: *option.split-asm*)
subgoal by(*cases* (*s*, *projl* (*projr x*)) *rule*: *lazy-channel-send.cases*)(*auto simp add*: *vld-def in-nlists-UNIV*)
subgoal by(*cases* (*s*, ()) *rule*: *lazy-channel-recv.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def option.pred-set*)
done
subgoal for *s*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *intro conjI WT-calleeI*; *clarsimp*)
subgoal for - - - *x*
by(*cases* (*s*, *x*) *rule*: *lazy-channel-insec.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def ran-def split*: *option.split-asm*)
subgoal by(*cases* (*s*, ()) *rule*: *lazy-channel-recv-f.cases*)(*auto 4 3 split*: *option.split-asm if-split-asm simp add*: *in-nlists-UNIV vld-def ran-def rnd-def*)
done
done

then have *WT3*: $\mathcal{I} \vdash_{res} RES$ (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*) $\surd$
by(*rule callee-invariant-on.WT-resource-of-oracle*) *simp*

have *callee-invariant-on* (††*insec-channel.insec-oracle* $\oplus_O$ *rorc-channel-send* $\oplus_O$ *rorc-channel-recv*)
(λ(-, *m*, *s*). *ran m* $\subseteq$ *nlists UNIV* η $\wedge$ *pred-cstate* (λ(*x*, *y*). *length x* = η $\wedge$ *length y* = η) *s*) $\mathcal{I}$
apply(*unfold-locales*)
subgoal for *s x y s'*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *elim PlusE*; *clarsimp*)
subgoal for - - - *x'*
by(*cases* (*snd* (*snd s*), *x'*) *rule*: *insec-channel.insec-oracle.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def insec-channel.insec-oracle.simps split*: *option.split-asm*)
subgoal
by(*cases snd* (*snd s*))
(*auto 4 3 simp add*: *channel.send-oracle.simps rorc-channel-send-def rorc.rnd-oracle.simps in-nlists-UNIV rnd-def vld-def mac-def ran-def split*: *option.split-asm if-split-asm*)
subgoal
by(*cases snd* (*snd s*))
(*auto 4 4 simp add*: *rorc-channel-recv-def channel.recv-oracle.simps rorc.rnd-oracle.simps rnd-def mac-def ran-def iff*: *in-nlists-UNIV split*: *option.split-asm if-split-asm*)
done
subgoal for *s*
apply(*clarsimp simp add*: $\mathcal{I}$*-def*; *intro conjI WT-calleeI*; *clarsimp*)
subgoal for - - - *x'*
by(*cases* (*snd* (*snd s*), *x'*) *rule*: *insec-channel.insec-oracle.cases*)
(*auto simp add*: *vld-def in-nlists-UNIV rnd-def insec-channel.insec-oracle.simps split*: *option.split-asm*)
subgoal
by(*cases snd* (*snd s*))
(*auto simp add*: *rorc-channel-recv-def channel.recv-oracle.simps rorc.rnd-oracle.simps rnd-def mac-def vld-def ran-def iff*: *in-nlists-UNIV split*: *option.split-asm if-split-asm*)
done
done
then have *WT4*: $\mathcal{I} \vdash_{res} RES$ (††*insec-channel.insec-oracle* $\oplus_O$ *rorc-channel-send* $\oplus_O$ *rorc-channel-recv*) ((*False*, ()), *Map.empty*, *Void*) $\surd$
by(*rule callee-invariant-on.WT-resource-of-oracle*) *simp*

note *game-difference*[*OF assms*(*3*), *folded* $\mathcal{I}$*-def*, *OF assms*(*4*,*5*)]
also note *connect-cong-trace*[*OF trace-eq-sim WT assm4-alt WT1 WT2*, *symmetric*]
also note *connect-cong-trace*[*OF trace-eq-lazy*, *OF assms*(*2*), *OF WT assm4-alt WT3 WT4*]
also note *ideal-resource-wiring*[*of sim*, *of Inl None*, *symmetric*]
also note *real-resource-wiring*[*symmetric*]

finally show *?thesis* **by** *simp*
qed

end

context begin
interpretation *MAC*: *macode rnd η mac η* **for** *η* **.**
interpretation *A-CHAN*: *auth-channel* **.**

lemma *WT-enm*:
X ≠ {} ⟹ *I-uniform* (*vld η*) *UNIV*, *I-uniform* (*vld η*) *X* ⊕*I* *I-uniform* (*X* × *vld η*) *UNIV* ⊢*C* *MAC.enm η* √
unfolding *MAC.enm-def*
by(*rule WT-converter-of-callee*) (*auto simp add*: *mac-def*)

lemma *WT-dem*: *I-uniform UNIV* (*insert None* (*Some* ‘ *vld η*)), *I-full* ⊕*I* *I-uniform UNIV* (*insert None* (*Some* ‘ (*nlists UNIV η* × *nlists UNIV η*))) ⊢*C* *MAC.dem η* √
unfolding *MAC.dem-def*
by (*rule WT-converter-of-callee*) (*auto simp add*: *vld-def stateless-callee-def mac-def split*: *option.split-asm*)

lemma *valid-insec-query-of* [*simp*]: *valid-mac-query η* (*insec-query-of x*)
by(*cases x*) *simp-all*

lemma *secure-mac*:
defines *I-real* ≡ λ*η*. *I-uniform* {*x*. *valid-mac-query η x*} (*insert None* (*Some* ‘ (*nlists UNIV η* × *nlists UNIV η*)))
and *I-ideal* ≡ λ*η*. *I-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV η*))
and *I-common* ≡ λ*η*. *I-uniform* (*vld η*) *UNIV* ⊕*I* *I-uniform UNIV* (*insert None* (*Some* ‘ *vld η*))
shows
constructive-security MAC.res (λ-. *A-CHAN.res*) (λ-. *CNV sim* (*Inl None*))
I-real I-ideal I-common (λ-. *enat q*) *True* (λ-. *insec-auth-wiring*)
proof
show *WT-res* [*WT-intro*]: *I-real η* ⊕*I* *I-common η* ⊢*res MAC.res η* √ **for** *η*
proof −
let *?I* = *pred-cstate* (λ(*x*, *y*). *length x* = *η* ∧ *length y* = *η*)

have ∗: *callee-invariant-on* (*MAC.RO.rnd-oracle η* ⊕*O* *MAC.RO.rnd-oracle η*) (λ*m*. *ran m* ⊆ *vld η*) (*I-uniform* (*vld η*) (*vld η*) ⊕*I* *I-full*) **for** *η*
apply *unfold-locales*
subgoal for *s x y s′* **by**(*cases x*; *clarsimp split*: *option.split-asm*; *auto simp add*: *rnd-def vld-def*)
subgoal by(*clarsimp intro*!: *WT-calleeI split*: *option.split-asm*; *auto simp add*: *rnd-def in-nlists-UNIV vld-def ran-def*)
done
have [*WT-intro*]: *I-uniform* (*vld η*) (*vld η*) ⊕*I* *I-full* ⊢*res MAC.RO.res η* √ **for** *η*

unfolding *MAC.RO.res-def* **by**(*rule callee-invariant-on.WT-resource-of-oracle*[*OF* ∗]) *simp*
have [*simp*]: $\mathcal{I}$*-real* $\eta \vdash c$ *MAC.INSEC.insec-oracle* $s \surd$ **if** *?I* s **for** s
apply(*rule WT-calleeI*)
subgoal for x **using** *that* **by**(*cases* (s, x) *rule*: *MAC.INSEC.insec-oracle.cases*)(*auto simp add*: $\mathcal{I}$*-real-def in-nlists-UNIV*)
done
have [*simp*]: $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ (*nlists UNIV* $\eta \times$ *nlists UNIV* η))) $\vdash c$ *A-CHAN.recv-oracle* $s \surd$
if *?I* s **for** s :: (*bool list* × *bool list*) *cstate* **using** *that*
by(*cases* s)(*auto intro*!: *WT-calleeI simp add*: *in-nlists-UNIV*)
have ∗: *callee-invariant-on* (*MAC.INSEC.insec-oracle* $\oplus_O$ *A-CHAN.send-oracle* $\oplus_O$ *A-CHAN.recv-oracle*) *?I*
($\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-uniform* (*vld* η × *vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ (*nlists UNIV* η × *nlists UNIV* η)))))
apply *unfold-locales*
subgoal for s x y s'
by(*cases* s; *cases* $(s,$ *projl* $x)$ *rule*: *MAC.INSEC.insec-oracle.cases*)(*auto simp add*: $\mathcal{I}$*-real-def vld-def in-nlists-UNIV*)
subgoal by(*auto intro*: *WT-calleeI*)
done
have [*WT-intro*]: $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-uniform* (*vld* η × *vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ (*nlists UNIV* η × *nlists UNIV* η)))) $\vdash$*res MAC.INSEC.res* $\surd$
unfolding *MAC.INSEC.res-def*
by(*rule callee-invariant-on.WT-resource-of-oracle*[*OF* ∗]) *simp*
show *?thesis*
unfolding $\mathcal{I}$*-common-def MAC.res-def*
by(*rule WT-intro WT-enm*[**where** *X*=*vld* η] *WT-dem* | (*simp add*: *vld-def*; *fail*))+
qed
let *?I* = $\lambda\eta$. *pred-cstate* (λx. *length* $x = \eta$)
have *WT-auth*: $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η)) $\vdash c$ *A-CHAN.auth-oracle* $s \surd$
if *?I* η s **for** η s
apply(*rule WT-calleeI*)
subgoal for x **using** *that* **by**(*cases* $(s,$ $x)$ *rule*: *A-CHAN.auth-oracle.cases*; *auto simp add*: *in-nlists-UNIV*)
done
have *WT-recv*: $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *vld* η)) $\vdash c$ *A-CHAN.recv-oracle* $s \surd$
if *?I* η s **for** η s **using** *that*
by(*cases* s)(*auto intro*!: *WT-calleeI simp add*: *vld-def in-nlists-UNIV*)
have ∗: *callee-invariant-on* (*A-CHAN.auth-oracle* $\oplus_O$ *A-CHAN.send-oracle* $\oplus_O$ *A-CHAN.recv-oracle*)
(*?I* η) ($\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) **for** η
apply(*unfold-locales*)
subgoal for s x y s'
by(*cases* $(s,$ *projl* $x)$ *rule*: *A-CHAN.auth-oracle.cases*; *cases projr* x)(*auto*

simp add: $\mathcal{I}$*-common-def vld-def in-nlists-UNIV*)
subgoal for *s* **using** *WT-auth WT-recv* **by**(*auto simp add*: $\mathcal{I}$*-common-def* $\mathcal{I}$*-ideal-def intro*: *WT-calleeI*)
done
show *WT-auth*: $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res A-CHAN.res* $\surd$ **for** η
unfolding *A-CHAN.res-def* **by**(*rule callee-invariant-on.WT-resource-of-oracle*[*OF* $*$]) *simp*

let *?I-sim* = $\lambda\eta$ (*s* :: (*bool list* × *bool list*) *option* + *unit*). $\forall$ *x y*. *s* = *Inl* (*Some* (*x*, *y*)) $\longrightarrow$ *length x* = η $\wedge$ *length y* = η

have $\mathcal{I}$*-real* η, $\mathcal{I}$*-ideal* η $\vdash_C$ *CNV sim s* $\surd$ **if** *?I-sim* η *s* **for** η *s* **using** *that*
apply(*coinduction arbitrary*: *s*)
subgoal for *q r conv*′ *s* **by**(*cases* (*s*, *q*) *rule*: *sim.cases*)(*auto simp add*: $\mathcal{I}$*-real-def* $\mathcal{I}$*-ideal-def vld-def in-nlists-UNIV*)
done
then show [*WT-intro*]: $\mathcal{I}$*-real* η, $\mathcal{I}$*-ideal* η $\vdash_C$ *CNV sim* (*Inl None*) $\surd$ **for** η **by** *simp*

{ fix $\mathcal{A}$ :: *security* $\Rightarrow$ ((*bool list* × *bool list*) *insec-query* + *bool list* + *unit*, (*bool list* × *bool list*) *option* + *unit* + *bool list option*) *distinguisher*
assume *WT*: $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash g$ $\mathcal{A}$ η $\surd$ **for** η
assume *bounded*[*simplified*]: *interaction-bounded-by*′ (λ-. *True*) ($\mathcal{A}$ η) *q* **for** η
assume *lossless*[*simplified*]: *True* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{A}$ η) **for** η
show *negligible* ($\lambda\eta$. *advantage* ($\mathcal{A}$ η) (*CNV sim* (*Inl None*) $|_=$ 1_C $\rhd$ *A-CHAN.res*) (*MAC.res* η))
proof −
have *f1*: *negligible* ($\lambda\eta$. *q* $*$ (*1* / *real* (*2* ^ η))) (**is** *negligible ?ov*)
by(*rule negligible-poly-times*[**where** *n*=*0*]) (*simp add*: *negligible-inverse-powerI*)+

have *f2*: ($\lambda\eta$. *spmf* (*connect* ($\mathcal{A}$ η) (*CNV sim* (*Inl None*) $|_=$ 1_C $\rhd$ *A-CHAN.res*)) *True* −
spmf (*connect* ($\mathcal{A}$ η) (*MAC.res* η)) *True*) $\in$ *O*(*?ov*) (**is** *?L* $\in$ -)
by (*auto simp add*: *bigo-def intro*!: *all-together*[*simplified*] *bounded eventually-at-top-linorderI*[**where** *c*=*1*]
exI[**where** *x*=*1*] *lossless*[*unfolded* $\mathcal{I}$*-real-def* $\mathcal{I}$*-common-def*] *WT*[*unfolded* $\mathcal{I}$*-real-def* $\mathcal{I}$*-common-def*])
have *negligible ?L* **using** *f1 f2* **by** (*rule negligible-mono*[*of ?ov*])
then show *?thesis* **by** (*simp add*: *advantage-def*)
qed
next
let *?cnv* = *map-converter id id insec-query-of auth-response-of* 1_C
show $\exists$ *cnv*. $\forall\mathcal{D}$. ($\forall\eta$. $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash g$ $\mathcal{D}$ η $\surd$) $\longrightarrow$
($\forall\eta$. *interaction-bounded-by*′ (λ-. *True*) ($\mathcal{D}$ η) *q*) $\longrightarrow$
($\forall\eta$. *True* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η)) $\longrightarrow$
($\forall\eta$. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*insec-query-of*, *map-option snd*)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) *A-CHAN.res* (*cnv* η $|_=$ 1_C $\rhd$ *MAC.res*

```
η))
  proof(intro exI[where x=λ-. ?cnv] strip conjI)
    fix 𝒟 :: security ⇒ (auth-query + bool list + unit, bool list option + unit +
bool list option) distinguisher
    assume WT-D [rule-format, WT-intro]: ∀η. ℐ-ideal η ⊕ℐ ℐ-common η ⊢g
𝒟 η √

    let ?A = λη. UNIV <+> nlists UNIV η <+> UNIV
    have WT1: ℐ-ideal η, map-ℐ insec-query-of (map-option snd) (ℐ-real η) ⊢C
1C √ for η
      using WT-converter-id order-refl by(rule WT-converter-mono)(auto simp
add: le-ℐ-def ℐ-ideal-def ℐ-real-def)
    have WT0: ℐ-ideal η ⊕ℐ ℐ-common η ⊢res map-converter id id insec-query-of
(map-option snd) 1C |= 1C ▷ MAC.res η √ for η
      by(rule WT1 WT-intro | simp)+

    have WT2: ℐ-ideal η, map-ℐ insec-query-of (map-option snd) (ℐ-real η) ⊢C
1C √ for η
      using WT-converter-id order-refl
     by(rule WT-converter-mono)(auto simp add: le-ℐ-def ℐ-ideal-def ℐ-real-def)
    have WT-cnv: ℐ-ideal η, ℐ-real η ⊢C ?cnv √ for η
      by(rule WT-converter-map-converter)(simp-all add: WT2)

    from eq-ℐ-converter-reflI[OF this] this
    show wiring (ℐ-ideal η) (ℐ-real η) ?cnv insec-auth-wiring for η ..

    define res' :: security ⇒ - ⇒ auth-query + (bool list + bool list × bool list)
+ bool list + unit ⇒ -
      where res' η =
        map-fun id (map-fun insec-query-of (map-spmf (map-prod (map-option
snd) id))) †MAC.INSEC.insec-oracle ⊕O
     ((MAC.RO.rnd-oracle η)† ⊕O †A-CHAN.send-oracle) ⊕O (MAC.RO.rnd-oracle
η)† ⊕O †A-CHAN.recv-oracle
      for η

     have push: map-resource (map-sum f id) (map-sum g id) ((1C |= conv) ▷
res) =
       (1C |= conv) ▷ map-resource (map-sum f id) (map-sum g id) res
      for f g conv res
    proof −
     have map-resource (map-sum f id) (map-sum g id) ((1C |= conv) ▷ res) =
map-converter f g id id 1C |= conv ▷ res
     by(simp add: attach-map-converter parallel-converter2-map1-out sum.map-id0)
     also have ... = (1C |= conv) ▷ map-resource (map-sum f id) (map-sum g
id) res
     by(subst map-converter-id-move-right)(simp add: parallel-converter2-map1-out
sum.map-id0 attach-map-converter)
     finally show ?thesis .
    qed
```

have *res'*: *map-resource* (*map-sum insec-query-of id*) (*map-sum* (*map-option snd*) *id*) (*MAC.res* η) =
1$_C$ |= *MAC.enm* η |= *MAC.dem* η ▷ *RES* (*res'* η) (*Map.empty*, *Void*) **for** η
unfolding *MAC.res-def MAC.RO.res-def MAC.INSEC.res-def interface-wiring push*
by(*simp add*: *parallel-converter2-map1-out sum.map-id0 attach-map-converter map-resource-resource-of-oracle map-sum-plus-oracle prod.map-id0 option.map-id0 map-fun-id res'-def*)

define *res''* :: *security* ⇒ (*unit* × *bool* × *unit*) × (*bool list* ⇒ *bool list option*) × - *cstate* ⇒ *auth-query* + *bool list* + *unit* ⇒ -
where *res''* η = *map-fun rprodl* (*map-fun id* (*map-spmf* (*map-prod id lprodr*)))
(*lift-state-oracle extend-state-oracle* †(*map-fun id* (*map-fun insec-query-of* (*map-spmf* (*map-prod* (*map-option snd*) *id*))) †*MAC.INSEC.insec-oracle*) ⊕$_O$
†(*map-fun rprodl* (*map-fun id* (*map-spmf* (*map-prod id lprodr*)))
(*lift-state-oracle extend-state-oracle*
(*attach-callee*
(λ*bs m*. *if bs then Done* ((), *True*) *else Pause* (*Inl m*) *Done* ⤜ (λ*r*. *lift-spmf* (*mac* η (*projl r*) *m*) ⤜ (λ*a*. *Pause* (*Inr* (*a*, *m*)) *Done* ⤜ (λ-. *Done* ((), *True*)))))
((*MAC.RO.rnd-oracle* η)† ⊕$_O$ †*A-CHAN.send-oracle*)) ⊕$_O$
††(λ*s q*. †*A-CHAN.recv-oracle s* () ⤜
(λ*x*. *case x of* (*None*, *s'*) ⇒ *return-spmf* (*None*, *s'*)
| (*Some* (*x1*, *x2a*), *s'*) ⇒ (*MAC.RO.rnd-oracle* η)† *s' x2a* ⤜ (λ(*x*, *s'*). *mac* η *x x2a* ⤜ (λ*y*. *return-spmf* (*if y = x1 then Some x2a else None*, *s'*))))))))
for η
have *?cnv* |= *1*$_C$ ▷ *MAC.res* η = *1*$_C$ |= *MAC.enm* η |= *MAC.dem* η ▷ *RES* (*res'* η) (*Map.empty*, *Void*) **for** η
by(*simp add*: *parallel-converter2-map1-out attach-map-converter sum.map-id0 res' attach-compose*[*symmetric*] *comp-converter-parallel2 comp-converter-id-left*)
also have ... η = *RES* (*res''* η) ((((), *False*, ()), *Map.empty*, *Void*) **for** η
unfolding *MAC.enm-def MAC.dem-def conv-callee-parallel*[*symmetric*] *conv-callee-parallel-id-left*[**where** *s*=(), *symmetric*] *attach-CNV-RES*
unfolding *res'-def res''-def attach-callee-parallel-intercept attach-stateless-callee attach-callee-id-oracle prod.rel-eq*[*symmetric*]
by(*simp add*: *extend-state-oracle-plus-oracle*[*symmetric*] *rprodl-extend-state-oracle sum.case-distrib*[**where** *h*=λ*x*. *exec-gpv - x* -]
option.case-distrib[**where** *h*=λ*x*. *exec-gpv - x* -] *prod.case-distrib*[**where** *h*=λ*x*. *exec-gpv - x* -] *exec-gpv-bind bind-map-spmf o-def*
cong: *sum.case-cong option.case-cong*)
also
define *S* :: *security* ⇒ *bool list cstate* ⇒ (*unit* × *bool* × *unit*) × (*bool list* ⇒ *bool list option*) × (*bool list* × *bool list*) *cstate* ⇒ *bool*
where *S* η *l r* = (*case* (*l*, *r*) *of*
(*Void*, ((-, *False*, -), *m*, *Void*)) ⇒ *m* = *Map.empty*
| (*Store x*, ((-, *True*, -), *m*, *Store* (*y*, *z*))) ⇒ *length x* = η ∧ *length y* = η ∧

length $z = \eta \wedge m = [z \mapsto y] \wedge x = z$
| (*Collect* x, ((-, *True*, -), m, *Collect* (y, z))) $\Rightarrow$ *length* $x = \eta \wedge$ *length* $y = \eta \wedge$ *length* $z = \eta \wedge m = [z \mapsto y] \wedge x = z$
| (*Fail*, ((-, *True*, -), m, *Fail*)) $\Rightarrow$ *True*
| - $\Rightarrow$ *False*) **for** η l r

note [*simp*] = *spmf-rel-map bind-map-spmf exec-gpv-bind*
have *connect* ($\mathcal{D}$ η) (*?cnv* $|_=$ 1_C $\triangleright$ *MAC.res* η) = *connect* ($\mathcal{D}$ η) *A-CHAN.res* **for** η
unfolding *calculation* **using** *WT-D - WT-auth*
apply(*rule connect-eq-resource-cong*[*symmetric*])
unfolding *A-CHAN.res-def*
apply(*rule eq-resource-on-resource-of-oracleI*[**where** S=S η])
apply(*rule rel-funI*)+
subgoal for s s' q q'
by(*cases q*; *cases projl q*; *cases projr q*; *clarsimp simp add*: *eq-on-def S-def res''-def split*: *cstate.split-asm bool.split-asm*; *clarsimp simp add*: *rel-spmf-return-spmf1 rnd-def mac-def bind-UNION* $\mathcal{I}$*-common-def vld-def in-nlists-UNIV S-def*)+
subgoal by(*simp add*: *S-def*)
done
then show *adv*: *negligible* ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) *A-CHAN.res* (*?cnv* $|_=$ 1_C $\triangleright$ *MAC.res* η))
by(*simp add*: *advantage-def*)
qed
}
qed

end

end

11 Secure composition: Encrypt then MAC

theory *Secure-Channel* **imports**
One-Time-Pad
Message-Authentication-Code
begin

context begin
interpretation *INSEC*: *insec-channel* **.**

interpretation *MAC*: *macode rnd* η *mac* η **for** η **.**
interpretation *AUTH*: *auth-channel* **.**

interpretation *CIPHER*: *cipher key* η *enc* η *dec* η **for** η **.**
interpretation *SEC*: *sec-channel* **.**

lemma *plossless-enc* [*plossless-intro*]:
plossless-converter ($\mathcal{I}$*-uniform* (*nlists UNIV* η) *UNIV*) ($\mathcal{I}$*-uniform UNIV* (*nlists*

UNIV η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*nlists UNIV* η) *UNIV*) (*CIPHER.enc* η)
unfolding *CIPHER.enc-def*
by(*rule plossless-converter-of-callee*) (*auto simp add*: *stateless-callee-def enc-def in-nlists-UNIV*)

lemma *plossless-dec* [*plossless-intro*]:
plossless-converter ($\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η))) ($\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η))) (*CIPHER.dec* η)
unfolding *CIPHER.dec-def*
by(*rule plossless-converter-of-callee*) (*auto simp add*: *stateless-callee-def dec-def in-nlists-UNIV split*: *option.split*)

lemma *callee-invariant-on-key-oracle*:
callee-invariant-on
(*CIPHER.KEY.key-oracle* η $\oplus_O$ *CIPHER.KEY.key-oracle* η)
(λx. *case x of None* $\Rightarrow$ *True* | *Some x'* $\Rightarrow$ *length x'* $= \eta$)
($\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*)
proof(*unfold-locales*, *goal-cases*)
case (*1 s x y s'*)
then show *?case* **by**(*cases x*; *clarsimp split*: *option.splits*; *simp add*: *key-def in-nlists-UNIV*)
next
case (*2 s*)
then show *?case* **by**(*clarsimp intro*!: *WT-calleeI split*: *option.split-asm*)(*simp-all add*: *in-nlists-UNIV key-def*)
qed

interpretation *key*: *callee-invariant-on*
CIPHER.KEY.key-oracle η $\oplus_O$ *CIPHER.KEY.key-oracle* η
λx. *case x of None* $\Rightarrow$ *True* | *Some x'* $\Rightarrow$ *length x'* $= \eta$
$\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full* **for** η
by(*rule callee-invariant-on-key-oracle*)

lemma *WT-enc* [*WT-intro*]: $\mathcal{I}$*-uniform* (*nlists UNIV* η) *UNIV*,
$\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*vld* η) *UNIV* $\vdash_C$ *CIPHER.enc* η $\surd$
unfolding *CIPHER.enc-def*
by (*rule WT-converter-of-callee*) (*simp-all add*: *stateless-callee-def vld-def enc-def in-nlists-UNIV*)

lemma *WT-dec* [*WT-intro*]: $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η)),
$\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *vld* η)) $\vdash_C$
CIPHER.dec η $\surd$
unfolding *CIPHER.dec-def*
by(*rule WT-converter-of-callee*)(*auto simp add*: *stateless-callee-def dec-def vld-def in-nlists-UNIV*)

lemma *bound-enc* [*interaction-bound*]: *interaction-any-bounded-converter* (*CIPHER.enc* η) (*enat 2*)
unfolding *CIPHER.enc-def*
by (*rule interaction-any-bounded-converter-of-callee*)
(*auto simp add*: *stateless-callee-def map-gpv-id-bind-gpv zero-enat-def one-enat-def*)

lemma *bound-dec* [*interaction-bound*]: *interaction-any-bounded-converter* (*CIPHER.dec* η) (*enat 2*)
unfolding *CIPHER.dec-def*
by (*rule interaction-any-bounded-converter-of-callee*)
(*auto simp add*: *stateless-callee-def map-gpv-id-bind-gpv zero-enat-def one-enat-def split*: *sum.split option.split*)

theorem *mac-otp*:
defines $\mathcal{I}$*-real* $\equiv \lambda\eta.$ $\mathcal{I}$*-uniform* {*x. valid-mac-query* η *x*} *UNIV*
and $\mathcal{I}$*-ideal* $\equiv \lambda$*-.* $\mathcal{I}$*-full*
and $\mathcal{I}$*-common* $\equiv \lambda\eta.$ $\mathcal{I}$*-uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*
shows
constructive-security
($\lambda\eta.$ 1_C $\models$ (*CIPHER.enc* η $\models$ *CIPHER.dec* η) $\odot$ *parallel-wiring* $\rhd$
parallel-resource1-wiring $\rhd$
CIPHER.KEY.res η $\|$
(1_C $\models$ *MAC.enm* η $\models$ *MAC.dem* η $\rhd$
1_C $\models$ *parallel-wiring* $\rhd$
parallel-resource1-wiring $\rhd$ *MAC.RO.res* η $\|$ *INSEC.res*))
(λ*-. SEC.res*)
($\lambda\eta.$ *CNV Message-Authentication-Code.sim* (*Inl None*) $\odot$ *CNV One-Time-Pad.sim None*)
($\lambda\eta.$ $\mathcal{I}$*-uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert None* (*Some* ‘ (*nlists UNIV* η × *nlists UNIV* η))))
($\lambda\eta.$ $\mathcal{I}$*-uniform UNIV* {*None*, *Some* η})
($\lambda\eta.$ $\mathcal{I}$*-uniform* (*nlists UNIV* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η)))
(λ*-. enat q*) *True* ($\lambda\eta.$ (*id*, *map-option length*) $\circ_w$ (*insec-query-of*, *map-option snd*))
proof(*rule composability*[*OF one-time-pad*[*THEN constructive-security2.constructive-security*, *unfolded CIPHER.res-alt-def comp-converter-parallel2 comp-converter-id-left*]
secure-mac[*unfolded MAC.res-def*,
THEN constructive-security.parallel-resource1,
THEN constructive-security.lifting],
where *?*$\mathcal{I}$*-res2*=$\lambda\eta.$ $\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) **and** *?bound-conv1*=λ*-. 2* **and** *?q3* = *2* ∗ *q* **and** *bound-sim* = λ*-.* ∞,
simplified]
, *goal-cases*)
case (*1* η)
have [*simp*]: $\mathcal{I}$*-uniform UNIV* (*nlists UNIV* η) $\vdash c$ *CIPHER.KEY.key-oracle* η *s* $\surd$

if *pred-option* ($\lambda x.$ *length* $x = \eta$) s **for** s η
apply(*rule WT-calleeI*)
subgoal for *call* **using** *that* **by**(*cases s*; *cases call*; *clarsimp*; *auto simp add*: *key-def in-nlists-UNIV*)
done
have $*$: *callee-invariant-on* (*CIPHER.KEY.key-oracle* $\eta \oplus_O$ *CIPHER.KEY.key-oracle* η) (*pred-option* ($\lambda x.$ *length* $x = \eta$))
(*I-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ *I-uniform UNIV* (*nlists UNIV* η)) **for** η
apply(*unfold-locales*)
subgoal for s x y s' **by**(*cases s*; *cases x*; *clarsimp*; *auto simp add*: *key-def in-nlists-UNIV*)
subgoal for s **by** *auto*
done
then show *?case* **unfolding** *CIPHER.KEY.res-def*
by(*rule callee-invariant-on.WT-resource-of-oracle*) *simp*

case (*2* η)
show *?case*
unfolding *CIPHER.KEY.res-def*
apply(*rule callee-invariant-on.lossless-resource-of-oracle*[*OF* $*$])
subgoal for s x **by**(*cases s*; *cases x*)(*auto split*: *plus-oracle-split*; *simp add*: *key-def*)+
subgoal by *simp*
done

case (*3* η)
show *?case* **by**(*rule WT-intro*)+

case (*4* η)
show *?case* **by** *interaction-bound-converter code-simp*

case (*5* η)
show *?case* **by** (*simp add*: *mult-2-right*)

case (*6* η)
show *?case* **unfolding** *vld-def* **by**(*rule plossless-intro WT-intro*[*unfolded vld-def*])+
qed

end

end
theory *Examples* **imports**
Secure-Channel/Secure-Channel
begin

end

References

[1] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *IACR Cryptology ePrint Archive*, 2017:753, 2017.

[2] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science (FOCS 2001), Proceedings*, pages 136–145, 2001.

[3] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[4] A. Lochbihler. CryptHOL. *Archive of Formal Proofs*, 2017. http://isa-afp.org/entries/CryptHOL.shtml, Formal proof development.

[5] U. Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In *Theory of Security and Applications - Joint Workshop (TOSCA 2011), Revised Selected Papers*, pages 33–56, 2011.

[6] U. Maurer and R. Renner. Abstract cryptography. In *Innovations in Computer Science (ICS 2010), Proceedings*, pages 1–21, 2011.

[7] U. M. Maurer. Indistinguishability of random systems. In *Advances in Cryptology (EUROCRYPT 2002), Proceedings*, pages 110–132, 2002.