Constructive Cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar

March 17, 2025

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 *Resource* **imports**
CryptHOL.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 $A' \leq A$ $B \leq B'$
shows *rel-resource A B* $\leq$ *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* ⇒ *′a2* ⇒ *bool* **and** *B* :: *′c1* ⇒ *′c2* ⇒ *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* (λ*x*. *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* (λ*x 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 (λb (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′ $\Longrightarrow$ 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 ⋀*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*; $\mathcal{I} \leq \mathcal{I}'$ ⟧ ⟹ *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* ⇒ -) *conversep* (*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-ℐ ℐ*; *I s* ⟧ ⟹ *lossless-spmf* (*callee s x*)
and *I s*
shows *lossless-resource 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 $\bigwedge$*res q r res'*. ⟦ *X res*; *q* ∈ *outs-ℐ ℐ*; (*r*, *res'*) ∈ *set-spmf* (*run-resource res q*) ⟧
⟹ *r* ∈ *responses-ℐ ℐ q* ∧ (*X res'* ∨ *ℐ* ⊢*res res'* √)
shows *ℐ* ⊢*res res* √
using *assms*(*1*) **by**(*rule WT-resource.coinduct*)(*blast dest*: *assms*(*2*))

lemma *WT-resourceD*:
assumes *ℐ* ⊢*res res* √ *q* ∈ *outs-ℐ ℐ* (*r*, *res'*) ∈ *set-spmf* (*run-resource res q*)
shows *r* ∈ *responses-ℐ ℐ q* ∧ *ℐ* ⊢*res res'* √
using *assms* **by**(*auto elim*: *WT-resource.cases*)

lemma *WT-resource-of-oracle* [*simp*]:
assumes $\bigwedge$*s*. *ℐ* ⊢*c oracle s* √
shows *ℐ* ⊢*res resource-of-oracle oracle s* √
by(*coinduction arbitrary*: *s*)(*auto dest*: *WT-calleeD*[*OF assms*])

lemma *WT-resource-bot* [*simp*]: *bot* ⊢*res res* √
by(*rule WT-resource.intros*)*auto*

lemma *WT-resource-full*: *ℐ-full* ⊢*res res* √
by(*coinduction arbitrary*: *res*)(*auto*)

lemma (**in** *callee-invariant-on*) *WT-resource-of-oracle*:
I s ⟹ *ℐ* ⊢*res resource-of-oracle callee s* √
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 *ℐ1* ⊢*res res1* √
and *ℐ2* ⊢*res res2* √
shows *ℐ1* $\oplus_{\mathcal{I}}$ *ℐ2* ⊢*res parallel-resource res1 res2* √
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*. *ℐ* ⊢*res res* √) *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 ℐ res* ∧ *ℐ* ⊢*res res* √) *ℐ*
by(*unfold-locales*)(*auto dest*: *WT-resourceD lossless-resourceD intro*: *WT-calleeI*)

interpretation *run-lossless-resource*:
callee-invariant-on run-resource λres. lossless-resource $\mathcal{I}$ *res* ∧ $\mathcal{I}$ ⊢*res res* √ $\mathcal{I}$
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* (λ*x*. *B* (*f x*)) (λ*x*. *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* (λ*x y*. *B x* (*f y*)) (λ*x 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* ⟧
$\Longrightarrow$ *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* ⟧
$\Longrightarrow$ *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*
(‹-,/ - $\vdash_C$/ - √› [*100*, *0*, *0*] *99*)
for $\mathcal{I}$ $\mathcal{I}'$ **where**
WT-converterI: $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *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* ∧ $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *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-ℐ ℐ ⟧
  ⟹ ℐ' ⊢g run-converter conv q √ ∧
      ((r, conv') ∈ results-gpv ℐ' (run-converter conv q) ⟶ r ∈ responses-ℐ ℐ q ∧
(X conv' ∨ ℐ, ℐ' ⊢C conv' √))
  shows ℐ, ℐ' ⊢C conv √
  using assms(1) by(rule WT-converter.coinduct)(blast dest: assms(2))

lemma WT-converterD:
  assumes ℐ, ℐ' ⊢C conv √ q ∈ outs-ℐ ℐ
  shows WT-converterD-WT: ℐ' ⊢g run-converter conv q √
    and WT-converterD-results: (r, conv') ∈ results-gpv ℐ' (run-converter conv q)
⟹ r ∈ responses-ℐ ℐ q ∧ ℐ, ℐ' ⊢C conv' √
  using assms by(auto elim: WT-converter.cases)

lemma WT-converterD':
  assumes ℐ, ℐ' ⊢C conv √ q ∈ outs-ℐ ℐ
  shows ℐ' ⊢g run-converter conv q √ ∧ (∀ (r, conv') ∈ results-gpv ℐ' (run-converter
conv q). r ∈ responses-ℐ ℐ q ∧ ℐ, ℐ' ⊢C conv' √)
  using assms by(auto elim: WT-converter.cases)

lemma WT-converter-bot1 [simp]: bot, ℐ ⊢C conv √
  by(rule WT-converter.intros) auto

lemma WT-converter-mono:
  ⟦ ℐ1,ℐ2 ⊢C conv √; ℐ1' ≤ ℐ1; ℐ2 ≤ ℐ2' ⟧ ⟹ ℐ1',ℐ2' ⊢C conv √
  apply(coinduction arbitrary: conv)
  apply(auto)
    apply(drule WT-converterD-WT)
     apply(erule (1) outs-ℐ-mono[THEN subsetD])
    apply(erule WT-gpv-mono)
     apply(erule outs-ℐ-mono)
    apply(erule (1) responses-ℐ-mono)
   apply(frule WT-converterD-results)
     apply(erule (1) outs-ℐ-mono[THEN subsetD])
    apply(erule results-gpv-mono[THEN subsetD])
     apply(erule WT-converterD-WT)
     apply(erule (1) outs-ℐ-mono[THEN subsetD])
    apply simp
   apply clarify
   apply(erule (2) responses-ℐ-mono[THEN subsetD])
  apply(frule WT-converterD-results)
    apply(erule (1) outs-ℐ-mono[THEN subsetD])
   apply(erule results-gpv-mono[THEN subsetD])
    apply(erule WT-converterD-WT)
    apply(erule (1) outs-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'*) $\in$ *results-gpv* $\mathcal{I}'$ (*run-converter* (*map-converter f g f' g' conv*) *q*)
then obtain *r' conv''*
where *results*: (*r'*, *conv''*) $\in$ *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'* $\in$ *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* $\in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *q* **by**(*auto simp add*: *surj-f-inv-f*[*OF g*])

show $\exists$ *conv*. *conv'* = *map-converter f g f' g' conv* $\wedge$
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* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$*; (*r*, *s'*) ∈ *results-gpv $\mathcal{I}'$* (*callee s q*) ⟧ ⟹ *r* ∈ *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* $\surd$; $\mathcal{I}2, \mathcal{I} \vdash_C$ *conv2* $\surd$ ⟧
⟹ *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-ru*
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 $\mathcal{I}1$,$\mathcal{I}2$ $\vdash_C$ *conv1* $\surd$
and $\mathcal{I}1'$,$\mathcal{I}2'$ $\vdash_C$ *conv2* $\surd$
shows $\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}1'$,$\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}2'$ $\vdash_C$ *parallel-converter2 conv1 conv2* $\surd$
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*) = (*λx. 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* $\surd$
shows $\mathcal{I}, \mathcal{I}' \vdash_C$ *converter-of-resource res* $\surd$
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 $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}1 \Longrightarrow$ *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*) = (*λa. if a* ∈ *A then*
map-gpv (*map-prod id* (*λcnv'. 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* $\surd$
shows $\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *restrict-converter A* $\mathcal{I}'$ *cnv* $\surd$
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* $\surd$ $\Longrightarrow$ *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* $\Rightarrow$ (*'out*, *'in*) *resource* $\Rightarrow$ (*'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*]: $\llbracket \mathcal{I}, \mathcal{I}' \vdash_C conv \surd; \mathcal{I}' \vdash res\ res \surd \rrbracket \Longrightarrow \mathcal{I} \vdash res\ attach\ conv\ res \surd$
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*)
$\bigwedge b\ conv'.\ (b, conv') \in results\text{-}gpv\ \mathcal{I}'\ (run\text{-}converter\ conv\ a) \Longrightarrow plossless\text{-}converter\ \mathcal{I}\ \mathcal{I}'\ conv'$ **by** *auto*
from *WT-converterD'*[*OF lossless-resource*(*3*,*5*)] **have** *WT*: $\mathcal{I}' \vdash g$ *run-converter conv a* $\surd$
$\bigwedge b\ conv'.\ (b, conv') \in results\text{-}gpv\ \mathcal{I}'\ (run\text{-}converter\ conv\ a) \Longrightarrow b \in responses\text{-}\mathcal{I}\ \mathcal{I}\ a \wedge \mathcal{I}, \mathcal{I}' \vdash_C conv' \surd$ **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** $\forall (b, res') \in ?set.\ ?P\ b\ res' \vee$ -)
proof(*safe*)
fix *b res''*
assume $(b, res'') \in ?set$
then obtain *conv' res'* **where** $*$: $((b, conv'), res') \in set\text{-}spmf$ (*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'*: $(b, conv') \in results\text{-}gpv\ \mathcal{I}'$ (*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*
:: $('s \Rightarrow 'a \Rightarrow ('b \times 's, 'out, 'in)\ gpv)$
$\Rightarrow ('s' \Rightarrow 'out \Rightarrow ('in \times 's')\ spmf)$
$\Rightarrow ('s \times 's' \Rightarrow 'a \Rightarrow ('b \times 's \times 's')\ spmf)$ **where**
attach-callee callee oracle = $(\lambda(s, s')\ q.$ *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* = *λres1res2 res1*. *res1res2* = *parallel-resource res1 res2* **and**
A=(=) **and** *CALL*=*λl r*. *l* = *Inl r* **and** *R*=*λl 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*]:
$[\![\mathcal{I}, \mathcal{I}' \vdash_C conv \surd; \mathcal{I}', \mathcal{I}'' \vdash_C conv' \surd]\!] \Longrightarrow \mathcal{I}, \mathcal{I}'' \vdash_C$ *comp-converter conv conv'*

$\surd$
  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* $\surd$ $\mathcal{I}', \mathcal{I}'' \vdash_C$ *conv'* $\surd$
  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** ∀ (*b*, *conv'*)∈*?res*. *?P b conv'* ∨ -)
  proof(*clarify*)
    fix *b conv''*
    assume (*b*, *conv''*) ∈ *?res*
    then obtain *conv1 conv2* **where** [*simp*]: *conv''* = *comp-converter conv1 conv2*

     and *inline*: ((*b*, *conv1*), *conv2*) ∈ *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*) ∈ *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 ∧ 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=λc *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=λl r. l = *Inl r* **and** R=λl 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*=*λc1 c2. c1 = parallel-converter2 conv2l c2* **and** *C*=*λl r. l = Inr r* **and** *R*=*λl 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-converter*
)

qed

lemma *interaction-any-bounded-converter-of-callee* [*interaction-bound*]:
assumes $\bigwedge$*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.cases*)

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**
eq-resource-onI: *A* $\vdash_R$ *res* ∼ *res'* **if**
⋀*a*. *a* ∈ *A* ⟹ *rel-spmf* (*rel-prod* (=) (*eq-resource-on A*)) (*run-resource res a*) (*run-resource res' a*)

lemma *eq-resource-on-coinduct* [*consumes 1*, *case-names eq-resource-on*, *coinduct pred*: *eq-resource-on*]:
assumes *X res res'*
and ⋀*res res' a*. ⟦ *X res res'*; *a* ∈ *A* ⟧
⟹ *rel-spmf* (*rel-prod* (=) (λ*res res'*. *X res res'* ∨ *A* $\vdash_R$ *res* ∼ *res'*)) (*run-resource res a*) (*run-resource res' a*)
shows *A* $\vdash_R$ *res* ∼ *res'*
using *assms*(*1*) **by**(*rule eq-resource-on.coinduct*)(*auto dest*: *assms*(*2*))

lemma *eq-resource-onD*:
assumes *A* $\vdash_R$ *res* ∼ *res'* *a* ∈ *A*
shows *rel-spmf* (*rel-prod* (=) (*eq-resource-on A*)) (*run-resource res a*) (*run-resource res' a*)
using *assms* **by**(*auto elim*: *eq-resource-on.cases*)

lemma *eq-resource-on-refl* [*simp*]: *A* $\vdash_R$ *res* ∼ *res*
by(*coinduction arbitrary*: *res*)(*auto intro*: *rel-spmf-reflI*)

lemma *eq-resource-on-reflI*: *res* = *res'* ⟹ *A* $\vdash_R$ *res* ∼ *res'*
by(*simp add*: *eq-resource-on-refl*)

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

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

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

lemma *eq-resource-on-UNIV-iff*: *UNIV* $\vdash_R$ *res* ∼ *res'* ⟷ *res* = *res'*
by(*auto dest*: *eq-resource-on-UNIV-D*)

lemma *eq-resource-on-mono*: ⟦ *A'* $\vdash_R$ *res* ∼ *res'*; *A* ⊆ *A'* ⟧ ⟹ *A* $\vdash_R$ *res* ∼ *res'*
by(*coinduction arbitrary*: *res res'*)(*auto dest*: *eq-resource-onD elim*!: *rel-spmf-mono*)

lemma *eq-resource-on-empty* [*simp*]: {} $\vdash_R$ *res* ∼ *res'*
by(*rule eq-resource-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* ∼ *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-ℐ ℐ* $\vdash_R$ *res* ∼ *res'*
and *ℐ* ⊢*g gpv* √
and *ℐ* ⊢*res res* √
shows *rel-spmf* (*rel-prod* (=) (*eq-resource-on* (*outs-ℐ 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-ℐ ℐ* $\vdash_R$ *r1* ∼ *r2*⟧ ⟹ *rel-spmf* (*rel-prod* (=) (*eq-resource-on* (*outs-ℐ 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-ℐ-generat* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'out*, *'in*) *ℐ* ⇒ (*'c* ⇒ *'d* ⇒ *bool*)

$\Rightarrow$ ('a, 'out, 'in $\Rightarrow$ 'c) *generat* $\Rightarrow$ ('b, 'out, 'in $\Rightarrow$ 'd) *generat* $\Rightarrow$ *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* $\in$ *outs-$\mathcal{I}$ $\mathcal{I}$* $\bigwedge$*input. input* $\in$ *responses-$\mathcal{I}$ $\mathcal{I}$ out* $\Longrightarrow$ *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′ ⟷
eq-$\mathcal{I}$-generat (λx. A (f x)) $\mathcal{I}$ (λx. D (g x)) generat generat′
by(*cases generat*; *cases generat′*) *auto*

lemma *eq-$\mathcal{I}$-generat-map2*:
eq-$\mathcal{I}$-generat A $\mathcal{I}$ D generat (map-generat f id ((∘) g) generat′) ⟷
eq-$\mathcal{I}$-generat (λx y. A x (f y)) $\mathcal{I}$ (λx y. D x (g y)) generat generat′
by(*cases generat*; *cases generat′*) *auto*

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

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

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

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

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

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

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

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

lemma *eq-$\mathcal{I}$-gpv-mono*: *eq-$\mathcal{I}$-gpv A $\mathcal{I}$* ≤ *eq-$\mathcal{I}$-gpv A′ $\mathcal{I}$′* **if** *A*: *A* ≤ *A′ $\mathcal{I}$* ≤ *$\mathcal{I}$′*
proof
show *eq-$\mathcal{I}$-gpv A′ $\mathcal{I}$′ gpv gpv′* **if** *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv′* **for** *gpv gpv′* **using** *that*
by(*coinduction arbitrary*: *gpv gpv′*)
(*drule eq-$\mathcal{I}$-gpvD*, *auto dest*: *eq-$\mathcal{I}$-gpvD elim*: *rel-spmf-mono eq-$\mathcal{I}$-generat-mono*[*OF A*(*1*) - *A*(*2*), *THEN predicate2D*, *rotated −1*])
qed

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

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

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

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

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

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

lemma *eq-$\mathcal{I}$-gpv-relcompp*: *eq-$\mathcal{I}$-gpv* (*A OO A′*) *$\mathcal{I}$* = *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ OO eq-$\mathcal{I}$-gpv A′*

$\mathcal{I}$ (**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-$\mathcal{I}$-generat* (*A OO A'*) $\mathcal{I}$ (*eq-$\mathcal{I}$-gpv* (*A OO A'*) $\mathcal{I}$)) (*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-$\mathcal{I}$-generat* (*A OO A'*) $\mathcal{I}$ (*eq-$\mathcal{I}$-gpv* (*A OO A'*) $\mathcal{I}$)) (*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-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv* (*middle* (*gpv*, *gpv''*)) **using** ∗
   by(*coinduction arbitrary*: *gpv gpv''*)
     (*drule eq-$\mathcal{I}$-gpvD*, *simp add*: *spmf-rel-map*, *erule rel-witness-spmf1*[*THEN rel-spmf-mono*]
     , *auto 4 3 del*: *relcomppE elim*!: *relcompp-witness eq-$\mathcal{I}$-generat.cases*)

  show *eq-$\mathcal{I}$-gpv A' $\mathcal{I}$* (*middle* (*gpv*, *gpv''*)) *gpv''* **using** ∗
   by(*coinduction arbitrary*: *gpv gpv''*)
     (*drule eq-$\mathcal{I}$-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-$\mathcal{I}$-generat.cases*)
next
  show *?lhs gpv gpv''* **if** *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv'* **and** *eq-$\mathcal{I}$-gpv A' $\mathcal{I}$ gpv' gpv''* **for** *gpv gpv' gpv''* **using** *that*
   by(*coinduction arbitrary*: *gpv gpv' gpv''*)
    ((*drule eq-$\mathcal{I}$-gpvD*)+, *simp*, *drule* (*1*) *rel-spmf-OO-trans*, *erule rel-spmf-mono*
     , *auto simp add*: *eq-$\mathcal{I}$-generat-relcompp elim*: *eq-$\mathcal{I}$-generat-mono'*)
qed

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

lemma *eq-𝓘-gpv-map-gpv2*: *eq-𝓘-gpv A 𝓘 gpv* (*map-gpv f id gpv′*) = *eq-𝓘-gpv* (*λx 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* (*λgpv 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* (*λx 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 $\mathcal{I}$ :: (*'out*, *'in*) $\mathcal{I}$
and $\mathcal{I}'$ :: (*'out'*, *'in'*) $\mathcal{I}$
assumes *callee*: ⋀*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' $\Longrightarrow$ *eq-$\mathcal{I}$-gpv A* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*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' $\Longrightarrow$ *eq-$\mathcal{I}$-gpv A* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*right-gpv gpv*) (*right-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-$\mathcal{I}$-gpvD-WT1*: $[\![$ *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv'*; $\mathcal{I} \vdash g$ *gpv* $\surd$ $]\!]$ $\Longrightarrow$ $\mathcal{I} \vdash g$ *gpv'* $\surd$
by(*coinduction arbitrary*: *gpv gpv'*)(*fastforce simp add*: *eq-$\mathcal{I}$-generat-iff2 dest*: *WT-gpv-ContD eq-$\mathcal{I}$-gpvD dest*!: *rel-setD2 set-spmf-parametric*[*THEN rel-funD*])

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

coinductive *eq-$\mathcal{I}$-converter* :: (*'a*, *'b*) $\mathcal{I}$ $\Rightarrow$ (*'out*, *'in*) $\mathcal{I}$ $\Rightarrow$ (*'a*, *'b*, *'out*, *'in*) *converter* $\Rightarrow$ (*'a*, *'b*, *'out*, *'in*) *converter* $\Rightarrow$ *bool*
(‹-,- $\vdash_C$/ - $\sim$/ -› [*100*, *0*, *99*, *99*] *99*)
for $\mathcal{I}$ $\mathcal{I}'$ **where**
eq-$\mathcal{I}$-converterI: $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* $\sim$ *conv'* **if**
$\bigwedge q$. $q \in$ *outs-$\mathcal{I}$ $\mathcal{I}$* $\Longrightarrow$ *eq-$\mathcal{I}$-gpv* (*rel-prod* (*eq-onp* (λr. $r \in$ *responses-$\mathcal{I}$ $\mathcal{I}$ q*)) (*eq-$\mathcal{I}$-converter $\mathcal{I}$ $\mathcal{I}'$*)) $\mathcal{I}'$ (*run-converter conv q*) (*run-converter conv' q*)

lemma *eq-$\mathcal{I}$-converter-coinduct* [*consumes 1*, *case-names eq-$\mathcal{I}$-converter*, *coinduct pred*: *eq-$\mathcal{I}$-converter*]:
assumes *X conv conv'*
and $\bigwedge$*conv conv' q*. $[\![$ *X conv conv'*; $q \in$ *outs-$\mathcal{I}$ $\mathcal{I}$* $]\!]$
$\Longrightarrow$ *eq-$\mathcal{I}$-gpv* (*rel-prod* (*eq-onp* (λr. $r \in$ *responses-$\mathcal{I}$ $\mathcal{I}$ q*)) (λ*conv conv'*. *X conv conv'* $\vee$ $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* $\sim$ *conv'*)) $\mathcal{I}'$
(*run-converter conv q*) (*run-converter conv' q*)
shows $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* $\sim$ *conv'*
using *assms*(*1*) **by**(*rule eq-$\mathcal{I}$-converter.coinduct*)(*blast dest*: *assms*(*2*))

lemma *eq-$\mathcal{I}$-converterD*:
eq-$\mathcal{I}$-gpv (*rel-prod* (*eq-onp* (λr. $r \in$ *responses-$\mathcal{I}$ $\mathcal{I}$ q*)) (*eq-$\mathcal{I}$-converter $\mathcal{I}$ $\mathcal{I}'$*)) $\mathcal{I}'$ (*run-converter conv q*) (*run-converter conv' q*)
if $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* $\sim$ *conv'* $q \in$ *outs-$\mathcal{I}$ $\mathcal{I}$*
using *that* **by**(*blast elim*: *eq-$\mathcal{I}$-converter.cases*)

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

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

lemma *eq-$\mathcal{I}$-converter-trans* [*trans*]:
⟦ $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* ∼ *conv'*; $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv'* ∼ *conv''* ⟧ ⟹ $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* ∼ *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* ∼ *conv'*
and *le*: $\mathcal{I}1' \le \mathcal{I}1$ $\mathcal{I}2 \le \mathcal{I}2'$
shows $\mathcal{I}1'$, $\mathcal{I}2'$ $\vdash_C$ *conv* ∼ *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 $\mathcal{I}'$ ⊢*res res* √ $\mathcal{I}$-*uniform A UNIV*, $\mathcal{I}'$ $\vdash_C$ *conv* ∼ *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 $\mathcal{I}'$ ⊢*res res* √ $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* ∼ *conv'* *A* ⊆ *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*:

$[\![\mathcal{I}' \vdash res\ res\ \surd;\ \mathcal{I}\text{-}full,\ \mathcal{I}' \vdash_C\ conv \sim conv']\!] \Longrightarrow attach\ conv\ res = attach\ conv'$ *res*
by(*rule eq-resource-on-UNIV-D*)(*simp add*: *eq-$\mathcal{I}$-attach-on*)

lemma *eq-$\mathcal{I}$-comp-cong*:
$[\![\mathcal{I}1, \mathcal{I}2 \vdash_C\ conv1 \sim conv1';\ \mathcal{I}2, \mathcal{I}3 \vdash_C\ conv2 \sim conv2']\!]$
$\Longrightarrow \mathcal{I}1, \mathcal{I}3 \vdash_C$ *comp-converter conv1 conv2* $\sim$ *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-$\mathcal{I}$-converterD*)+)

lemma *comp-converter-cong*: *comp-converter conv1 conv2* = *comp-converter conv1' conv2'*
if $\mathcal{I}$-*full*, $\mathcal{I} \vdash_C$ *conv1* $\sim$ *conv1'* $\mathcal{I}$, $\mathcal{I}$-*full* $\vdash_C$ *conv2* $\sim$ *conv2'*
by(*rule eq-$\mathcal{I}$-converter-eq*)(*rule eq-$\mathcal{I}$-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-$\mathcal{I}$-gpv-Pause simp add*: *eq-onp-same-args*)

lemma *parallel-converter2-eq-$\mathcal{I}$-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-$\mathcal{I}$-gpv-left-gpv-cong eq-$\mathcal{I}$-gpv-right-gpv-cong dest*: *eq-$\mathcal{I}$-converterD elim*!: *eq-$\mathcal{I}$-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-$\mathcal{I}$-converter-mono*[*OF - order-refl that*])(*rule eq-$\mathcal{I}$-converter-reflI*[*OF WT-converter-id*])

lemma *eq-$\mathcal{I}$-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-$\mathcal{I}$-converterD*, *auto 4 3 dest*: *eq-$\mathcal{I}$-converterD eq-$\mathcal{I}$-gpvD-WT1 WT-converterD-WT WT-converterD-results eq-$\mathcal{I}$-gpvD-results-gpv2 simp add*: *eq-onp-def*)

lemma *eq-$\mathcal{I}$-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-$\mathcal{I}$-converterD-WT1*)
next
case *2* **then show** *?case* **using** *assms*[*symmetric*] **by** (*auto intro*: *eq-$\mathcal{I}$-converterD-WT1*)
qed

lemma *eq-$\mathcal{I}$-gpv-Fail* [*simp*]: *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ Fail Fail*
  by(*rule eq-$\mathcal{I}$-gpv.intros*) *simp*

lemma *eq-$\mathcal{I}$-restrict-gpv*:
  assumes *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv'*
  shows *eq-$\mathcal{I}$-gpv A $\mathcal{I}$* (*restrict-gpv $\mathcal{I}$ gpv*) *gpv'*
  using *assms*
  by(*coinduction arbitrary*: *gpv gpv'*)
    (*fastforce dest*: *eq-$\mathcal{I}$-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* $\Rightarrow$ *'s spmf* $\Rightarrow$ (*'a* $\times$ *'b*) *list* $\Rightarrow$ *'a* $\Rightarrow$ *'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* (*λs. 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*
(‹- ⊢$_C$/ (-'((-)')) ≈/ (-'((-)'))› [*90*, *0*, *0*, *0*, *0*] *91*)
where *trace-callee-eq' A callee1 s1 callee2 s2* ≡ *trace-callee-eq callee1 callee2 A* (*return-spmf s1*) (*return-spmf s2*)

lemma *trace-callee-eqI*:
assumes ⋀*xs x*. ⟦ *set xs* ⊆ *A* × *UNIV*; *x* ∈ *A* ⟧
⟹ *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* ⊆ *A* × *UNIV x* ∈ *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*: ⋀*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 *sim*: ⋀*p q a res b s'*. ⟦ *S p q*; *a* ∈ *A*; *res* ∈ *set-spmf q*; (*b*, *s'*) ∈ *set-spmf* (*callee2 res a*) ⟧
⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (*λs. callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q* (*λs. callee2 s a*)) *b*)
shows *trace-callee-eq callee1 callee2 A p q*
proof(*rule trace-callee-eqI*)
fix *xs* :: (*'a* × *'b*) *list* **and** *z*
assume *xs*: *set xs* ⊆ *A* × *UNIV* **and** *z*: *z* ∈ *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* ∈ *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* × *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* ∈ *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 ⋀*p q a*. ⟦ *S p q*; *a* ∈ *A* ⟧ ⟹
      bind-spmf p (λ*s*. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ*s*. *map-spmf fst* (*callee2 s a*))
    and ⋀*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* (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 ℐ*
and *inv1*: *callee-invariant-on callee2 I2 ℐ*
and *WT*: *ℐ* ⊢*g gpv* √
and *out*: *outs-gpv 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 (λ*s*. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ*s*. *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* (λ*s*. *callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s a*)) *b*)
by(*rule trace-callee-eq-complete*) *blast*

interpret *I1*: *callee-invariant-on callee1 I1 ℐ* **by** *fact*
interpret *I2*: *callee-invariant-on callee2 I2 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 ≫= (λ*xa*. *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 ≫= (λ*xa*. *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* ≫= (λ*xa*. *callee1 xa out*)) = *map-spmf fst* (*q* ≫= (λ*xa*. *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* ≫= (λ*xa*. *callee1 xa out*)) *a* ≫= (λ*x*. *map-spmf fst* (*exec-gpv′* (*rpv a*) *x*)) =
cond-spmf-fst (*q* ≫= (λ*xa*. *callee2 xa out*)) *a* ≫= (λ*x*. *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*: $A \vdash_C$ *callee1*(*s1*) ≈ *callee2*(*s2*)
and *inv1*: *callee-invariant-on callee1 I1* $\mathcal{I}$
and *inv1*: *callee-invariant-on callee2 I2* $\mathcal{I}$
and *WT*: $\mathcal{I} \vdash g$ *gpv* √
and *out*: *outs-gpv* $\mathcal{I}$ *gpv* ⊆ *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*
(‹(-) $\vdash_R$/ (-)/ ≈ (-)› [*90*, *90*, *90*] *91*) **where**
$A \vdash_R$ *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*]:
$A \vdash_R$ *resource-of-oracle callee1 s1* ≈ *resource-of-oracle callee2 s2* =
$A \vdash_C$ *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* = λ*res* (*conv'*, *res'*). *res* = *attach conv' res'*
have∗: *rel-spmf* (*rel-prod* (=) *?R*) (*exec-gpv run-resource d* (*attach conv res*))
(*exec-gpv* (λ*p y*. *map-spmf* (λ*p*. (*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*=λ*res* (*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* = λ*conv* (*conv'*, *conv''*). *conv* = *comp-converter conv' conv''*
have∗: *rel-gpv* (*rel-prod* (=) *?R*) (=) (*inline run-converter d* (*comp-converter conv conv'*))
(*inline* (λ*p c2*. *map-gpv* (λ*p*. (*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*: $A \vdash_R$ *res1* ≈ *res2*
and *WT*: $\mathcal{I}$ ⊢*g d* √
and *out*: *outs-gpv* $\mathcal{I}$ *d* ⊆ *A*
and *WT1*: $\mathcal{I}$ ⊢*res res1* √
and *WT2*: $\mathcal{I}$ ⊢*res res2* √
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*: ⋀*distinguisher*. $\mathcal{I}$ ⊢*g distinguisher* √ ⟹ *connect distinguisher res* = *connect distinguisher res'*
and *WT1*: $\mathcal{I}$ ⊢*res res1* √
and *WT2*: $\mathcal{I}$ ⊢*res res2* √
shows *outs-*$\mathcal{I}$ $\mathcal{I}$ $\vdash_R$ *res* ≈ *res'*
proof(*rule ccontr*)
let *?P* = λ*xs*. ∃ *x*. *set xs* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$ × *UNIV* ∧ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ∧ *trace-callee run-resource* (*return-spmf res*) *xs x* ≠ *trace-callee run-resource* (*return-spmf res'*) *xs x*
assume ¬ *?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* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$ × *UNIV*
and *x*: *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$
and *neq*: *trace-callee run-resource* (*return-spmf res*) *xs x* ≠ *trace-callee run-resource* (*return-spmf res'*) *xs x*
and *shortest*: ⋀*xs' x*. ⟦ *strict-prefix xs' xs*; *set xs'* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$ × *UNIV*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ *trace-callee run-resource* (*return-spmf res*) *xs' x* = *trace-callee run-resource* (*return-spmf res'*) *xs' x*
by(*auto*)
have *shortest*: ⋀*xs' x*. ⟦ *strict-prefix xs' xs*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ *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* ≡ *return-spmf res*
define *q* **where** *q* ≡ *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* ≠ *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* ∈ *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* × *'b*) *list* ⇒ -
  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*: $\mathcal{I}$ ⊢*g d xs* √ **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* ⪢= *connect* (*d xs*)) *True* =
LINT a|*measure-spmf* (*q* ⪢= (λ*s*. *map-spmf fst* (*run-resource s x*′))). *indicator* {*y*′} *a* ∗ *spmf* (*?q* ⪢= *connect* (*d xs*)) *True*
by(*rule box-equals*; *use nothing* **in** ‹*rule Bochner-Integration.integral-cong*›) *auto*
hence *spmf* (*p* ⪢= (λ*s*. *map-spmf fst* (*run-resource s x*′))) *y*′ ∗ *spmf* (*?p* ⪢= *connect* (*d xs*)) *True* =
spmf (*q* ⪢= (λ*s*. *map-spmf fst* (*run-resource s x*′))) *y*′ ∗ *spmf* (*?q* ⪢= *connect* (*d xs*)) *True*
by(*simp add*: *spmf-conv-measure-spmf*)
moreover from *Cons.prems*(*3*)[*of* [] *x*′] *Cons.prems*(*4*)
have *spmf* (*p* ⪢= (λ*s*. *map-spmf fst* (*run-resource s x*′))) *y*′ = *spmf* (*q* ⪢= (λ*s*. *map-spmf fst* (*run-resource s x*′))) *y*′
by(*simp*)
ultimately have *spmf* (*?p* ⪢= *connect* (*d xs*)) *True* = *spmf* (*?q* ⪢= *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* **for** *ys x y*
using *Cons.prems*(*3*)[*of xy* # *ys x y*] *ys x* **by** *simp*
moreover have *set xs* ⊆ *outs-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* $\vdash_R$ *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*]:
  ⟦ $\mathcal{I}' \vdash g$ *gpv* $\surd$; $\mathcal{I}'$, $\mathcal{I} \vdash_C$ *conv* $\surd$ ⟧ $\Longrightarrow$ $\mathcal{I} \vdash g$ *absorb gpv conv* $\surd$
  by(*simp add*: *absorb-def run-converter.WT-gpv-inline-invar*)

lemma *plossless-gpv-absorb* [*plossless-intro*]:
  assumes *gpv*: *plossless-gpv* $\mathcal{I}'$ *gpv*
    and *conv*: *plossless-converter* $\mathcal{I}'$ $\mathcal{I}$ *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* $\equiv lassocr_C \odot (1_C |_= swap_C) \odot rassocl_C$
definition *swap-lassocr* **where** *swap-lassocr* $\equiv rassocl_C \odot (swap_C |_= 1_C) \odot lassocr_C$

definition *parallel-wiring* :: $(('a + 'b) + ('e + 'f), ('c + 'd) + ('g + 'h), ('a + 'e) + ('b + 'f), ('c + 'g) + ('d + 'h))$ *converter* **where**
  parallel-wiring = $lassocr_C \odot (1_C |_=$ *swap-lassocr*$) \odot rassocl_C$

lemma *WT-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-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-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-lassocr* $\surd$
unfolding *swap-lassocr-def*
by(*rule WT-converter-comp WT-lassocr*$_C$ *WT-rassocl*$_C$ *WT-converter-parallel-converter2 WT-converter-id WT-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*:
$((C1 \mid_= C2) \mid_= (C3 \mid_= C4)) \odot$ *parallel-wiring* $=$ *parallel-wiring* $\odot ((C1 \mid_= C3) \mid_= (C2 \mid_= C4))$
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 $\mid_\propto 1_C \rhd$ *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*:
$1_C \mid_\propto$ *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* ∘ *map-sum swap-sum id* ∘ *lsumr*

lemma *run-converter-swap-lassocr* [*simp*]:
run-converter swap-lassocr x = *Pause* (*lassocr-swap-sum x*) (
case lsumr x of Inl - ⇒ (λ*y*. *case lsumr y of Inl* - ⇒ *Done* (*lassocr-swap-sum y*, *swap-lassocr*) | - ⇒ *Fail*)
| *Inr* - ⇒ (λ*y*. *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* = ($'a \Rightarrow 'c$) × ($'d \Rightarrow 'b$)

inductive *wiring* :: ($'a$, $'b$) $\mathcal{I}$ ⇒ ($'c$, $'d$) $\mathcal{I}$ ⇒ ($'a$, $'b$, $'c$, $'d$) *converter* ⇒ ($'a$, $'b$, $'c$, $'d$) *wiring* ⇒ *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* ∼ *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* ∼ *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* ⇒ ($'s$, $'c$, $'d$) *oracle'* ⇒ ($'s$, $'a$, $'b$) *oracle'*
where *apply-wiring* = (λ(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*: $\mathcal{I}2$ ⊢res *RES res s* √
and *outs*: *outs-*$\mathcal{I}$ $\mathcal{I}1$ = *UNIV*
shows *conv* ▷ *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* ∼ *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* ▷ *RES res s* = *map-converter id id f g* 1_C ▷ *RES res s*
by(*rule eq-*$\mathcal{I}$*-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* $\mathcal{I}$ $\mathcal{I}$ 1_C (*id*, *id*)
using *wiring.intros*[*of* $\mathcal{I}$ $\mathcal{I}$ 1_C *id id*]
by(*simp add*: *eq-*$\mathcal{I}$*-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* $\mathcal{I}1$ $\mathcal{I}2$ *conv w*
and *WT*: $\mathcal{I}2$, $\mathcal{I}3$ $\vdash_C$ *CNV callee s* √
shows $\mathcal{I}1$, $\mathcal{I}3$ $\vdash_C$ *conv* ⊙ *CNV callee s* ∼ *CNV* (*attach-wiring w callee*) *s*
using *wiring*
proof *cases*
case (*wiring f g*)
from *wiring*(*2*) **have** $\mathcal{I}1$,$\mathcal{I}3$ $\vdash_C$ *conv* ⊙ *CNV callee s* ∼ *map-converter id id f g* 1_C ⊙ *CNV callee s*
by(*rule eq-*$\mathcal{I}$*-comp-cong*)(*rule eq-*$\mathcal{I}$*-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* $\Rightarrow$ $('c, 'd, 'e, 'f)$ *wiring* $\Rightarrow$ $('a, 'b, 'e, 'f)$ *wiring* (**infixl** ‹$\circ_w$› *55*)
  where *comp-wiring* $= (\lambda(f, g)\ (f', g').\ (f' \circ f, g \circ g'))$

lemma *comp-wiring-simps*: *comp-wiring* $(f, g)\ (f', g') = (f' \circ f, g \circ g')$
  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 $= (\lambda(f, g)\ (f', g').$ (*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* $\mathcal{I}1$ $\mathcal{I}1'$ *conv1 fg*
    and *wiring* $\mathcal{I}2$ $\mathcal{I}2'$ *conv2 fg'*
  shows *wiring* $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2)$ $(\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2')$ (*conv1* $|_=$ *conv2*) (*fg* $|_w$ *fg'*)
proof −
  from *assms*(*1*) **obtain** *f1 g1*

where *conv1*: $\mathcal{I}1$,$\mathcal{I}1'$ $\vdash_C$ *conv1* $\sim$ *map-converter id id f1 g1* 1_C
and *WT1*: $\mathcal{I}1$, $\mathcal{I}1'$ $\vdash_C$ *conv1* $\surd$
and [*simp*]: *fg* = (*f1*, *g1*)
by *cases*
from *assms*(*2*) **obtain** *f2 g2*
where *conv2*: $\mathcal{I}2$,$\mathcal{I}2'$ $\vdash_C$ *conv2* $\sim$ *map-converter id id f2 g2* 1_C
and *WT2*: $\mathcal{I}2$, $\mathcal{I}2'$ $\vdash_C$ *conv2* $\surd$
and [*simp*]: *fg'* = (*f2*, *g2*)
by *cases*
from *eq-$\mathcal{I}$-converterD-WT1*[*OF conv1 WT1*] **have** $\mathcal{I}1$: $\mathcal{I}1 \leq$ *map-$\mathcal{I}$ f1 g1* $\mathcal{I}1'$
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 $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$, $\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2'$ $\vdash_C$ ... $\sim$ 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 (($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) $\oplus_{\mathcal{I}}$ $\mathcal{I}3$) ($\mathcal{I}1 \oplus_{\mathcal{I}}$ ($\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3$)) $lassocr_C$ $lassocr_w$
unfolding $lassocr_C$*-def* $lassocr_w$*-def*
by(*subst map-converter-id-move-right*)(*auto intro*!: *wiringI eq-*$\mathcal{I}$*-converter-reflI WT-converter-map-converter*)

lemma *wiring-rassocl* [*simp*, *wiring-intro*]:
wiring ($\mathcal{I}1 \oplus_{\mathcal{I}}$ ($\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3$)) (($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) $\oplus_{\mathcal{I}}$ $\mathcal{I}3$) $rassocl_C$ $rassocl_w$
unfolding $rassocl_C$*-def* $rassocl_w$*-def*
by(*subst map-converter-id-move-right*)(*auto intro*!: *wiringI eq-*$\mathcal{I}$*-converter-reflI WT-converter-map-converter*)

lemma *wiring-swap* [*simp*, *wiring-intro*]: *wiring* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) ($\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}1$) $swap_C$ $swap_w$
unfolding $swap_C$*-def* $swap_w$*-def*
by(*subst map-converter-id-move-right*)(*auto intro*!: *wiringI eq-*$\mathcal{I}$*-converter-reflI WT-converter-map-converter*)

lemma *apply-*$lassocr_w$ [*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-*$rassocl_w$ [*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-*$swap_w$ [*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*]: $\bigwedge\eta$. $\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η ⊢*res real-resource* η $\surd$
and *WT-ideal* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η ⊢*res ideal-resource* η $\surd$
and *WT-sim* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$-*real* η, $\mathcal{I}$-*ideal* η $\vdash_C$ *sim* η $\surd$
and *adv*: $\bigwedge\mathcal{A}$ :: *security* ⇒ ('*a* + '*e*, '*b* + '*f*) *distinguisher*.
⟦ $\bigwedge\eta$. $\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η ⊢*g* $\mathcal{A}$ η $\surd$;
$\bigwedge\eta$. *interaction-bounded-by* (λ-. *True*) ($\mathcal{A}$ η) (*bound* η);
$\bigwedge\eta$. *lossless* ⟹ *plossless-gpv* ($\mathcal{I}$-*real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*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 $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η ⊢*g* $\mathcal{D}$ η $\surd$)
⟶ (∀ η. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound* η))
⟶ (∀ η. *lossless* ⟶ *plossless-gpv* ($\mathcal{I}$-*ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*common* η) ($\mathcal{D}$ η))
⟶ (∀ η. *wiring* ($\mathcal{I}$-*ideal* η) ($\mathcal{I}$-*real* η) (*cnv* η) (*w* η)) ∧

negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ *1*$_C$ $\rhd$ *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* $\Rightarrow$ ($'a$ + $'e$, $'b$ + $'f$) *resource*
and *ideal-resource* :: *security* $\Rightarrow$ ($'c$ + $'e$, $'d$ + $'f$) *resource*
and *sim* :: *security* $\Rightarrow$ ($'a$, $'b$, $'c$, $'d$) *converter*
and $\mathcal{I}$*-real* :: *security* $\Rightarrow$ ($'a$, $'b$) $\mathcal{I}$
and $\mathcal{I}$*-ideal* :: *security* $\Rightarrow$ ($'c$, $'d$) $\mathcal{I}$
and $\mathcal{I}$*-common* :: *security* $\Rightarrow$ ($'e$, $'f$) $\mathcal{I}$
and *bound* :: *security* $\Rightarrow$ *enat*
and *lossless* :: *bool*
and *w* :: *security* $\Rightarrow$ ($'c$, $'d$, $'a$, $'b$) *wiring*
+
assumes *sim*: $\exists$ *cnv*. $\forall$ η. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η) $\wedge$ *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-ideal* η) (*cnv* η $\odot$ *sim* η) (*id*, *id*)
begin

lemma *constructive-security*:
constructive-security real-resource ideal-resource sim $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common bound lossless w*
proof
from *sim* **obtain** *cnv*
where *w*: $\bigwedge\eta$. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η)
and *inverse*: $\bigwedge\eta$. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-ideal* η) (*cnv* η $\odot$ *sim* η) (*id*, *id*)
by *blast*
show $\exists$ *cnv*. $\forall$ $\mathcal{D}$. ($\forall$ η. $\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η $\vdash$*g* $\mathcal{D}$ η $\surd$)
$\longrightarrow$ ($\forall$ η. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound* η))
$\longrightarrow$ ($\forall$ η. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η))
$\longrightarrow$ ($\forall$ η. *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 $|_=$ *1*$_C$ $\rhd$ *real-resource* η))
proof(*intro strip exI conjI*)
fix $\mathcal{D}$ :: *security* $\Rightarrow$ ($'c$ + $'e$, $'d$ + $'f$) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall$ η. $\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η $\vdash$*g* $\mathcal{D}$ η $\surd$
and *bound* [*rule-format*, *interaction-bound*]: $\forall$ η. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound* η)
and *lossless* [*rule-format*]: $\forall$ η. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η)

show *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η) **for** η **by** *fact*

let *?A* = $\lambda\eta$. *outs-*$\mathcal{I}$ ($\mathcal{I}$*-ideal* η)
let *?cnv* = $\lambda\eta$. *restrict-converter* (*?A* η) ($\mathcal{I}$*-real* η) (*cnv* η)
let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{D}$ η) (*?cnv* η $|_=$ *1*$_C$)

have *eq*: *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ *1*$_C$ $\rhd$ *real-resource* η) = *advantage* (*?*$\mathcal{A}$ η) (*sim* η $|_=$ *1*$_C$ $\rhd$ *ideal-resource* η) (*real-resource* η) **for** η
proof −
from *w*[*of* η] **have** [*WT-intro*]: $\mathcal{I}$*-ideal* η, $\mathcal{I}$*-real* η $\vdash_C$ *cnv* η $\surd$ **by** *cases*
have $\mathcal{I}$*-ideal* η, $\mathcal{I}$*-ideal* η $\vdash_C$ *?cnv* η $\odot$ *sim* η $\sim$ *cnv* η $\odot$ *sim* η
by(*rule eq-*$\mathcal{I}$*-comp-cong eq-*$\mathcal{I}$*-restrict-converter WT-intro order-refl eq-*$\mathcal{I}$*-converter-reflI*)+
also from *inverse*[*of* η] **have** $\mathcal{I}$*-ideal* η, $\mathcal{I}$*-ideal* η $\vdash_C$ *cnv* η $\odot$ *sim* η $\sim$ *1*$_C$ **by** *cases simp*
finally have *inverse'*: $\mathcal{I}$*-ideal* η, $\mathcal{I}$*-ideal* η $\vdash_C$ *?cnv* η $\odot$ *sim* η $\sim$ *1*$_C$ **.**
hence $\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η, $\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η $\vdash_C$ *?cnv* η $\odot$ *sim* η $|_=$ *1*$_C$ $\sim$ *1*$_C$ $|_=$ *1*$_C$
by(*rule parallel-converter2-eq-*$\mathcal{I}$*-cong*)(*intro eq-*$\mathcal{I}$*-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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* $\eta \vdash_C$ *cnv* η $\surd$
and *eq*: $\mathcal{I}$*-ideal* η, $\mathcal{I}$*-real* $\eta \vdash_C$ *cnv* $\eta \sim$ *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* $\eta \leq$ *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* $\eta \odot$ *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 η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-common η $\vdash$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 η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-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 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-inner* η) **for** η
by *interaction-bound-converter*(*rule bound-le*)
moreover have *plossless-gpv* (*$\mathcal{I}$-middle η $\oplus_{\mathcal{I}}$ $\mathcal{I}$-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* η $\odot$ *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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ $\mathcal{D}$ 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* $\eta \oplus_{\mathcal{I}}$
$\mathcal{I}$*-common* η) ($\mathcal{D}$ η)) $\longrightarrow$
($\forall\eta$. *wiring* ($\mathcal{I}$*-inner* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w1* $\eta \circ_w$ *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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$
$\mathcal{D}$ 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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η)

let *?cnv* = $\lambda\eta$. *cnv-inner* $\eta \odot$ *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* $\eta \vdash_C$
cnv-inner η $\surd$

and *fg* [*simp*]: $\bigwedge\eta.$ *w1* $\eta = (f\ \eta,\ g\ \eta)$
and *eq1*: $\bigwedge\eta.$ $\mathcal{I}$*-inner* η, $\mathcal{I}$*-middle* $\eta \vdash_C$ *cnv-inner* $\eta \sim$ *map-converter id id* $(f\ \eta)\ (g\ \eta)\ 1_C$
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* $\eta \vdash_C$ *cnv-inner* $\eta\ \surd$ **for** η **by** *cases*

let *?*$\mathcal{D}$ = $\lambda\eta.$ *absorb* ($\mathcal{D}$ η) (*map-converter id id* $(f\ \eta)\ (g\ \eta)\ 1_C \mid_= 1_C$)
have $\mathcal{I}$: $\mathcal{I}$*-inner* $\eta \leq$ *map-*$\mathcal{I}$ $(f\ \eta)\ (g\ \eta)$ ($\mathcal{I}$*-middle* η) **for** η
using *eq-*$\mathcal{I}$*-converterD-WT1*[*OF eq1 WT-inner*, *of* η] **by**(*rule WT-map-converter-idD*)
with *WT-converter-id* **have** [*WT-intro*]: $\mathcal{I}$*-inner* η, *map-*$\mathcal{I}$ $(f\ \eta)\ (g\ \eta)$ ($\mathcal{I}$*-middle* η) $\vdash_C 1_C\ \surd$
for η **by**(*rule WT-converter-mono*) *simp*

have *WT-D′*: $\mathcal{I}$*-middle* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ *?*$\mathcal{D}$ $\eta\ \surd$ **for** η **by**(*rule WT-intro* | *simp*)+
have *bound′*: *interaction-any-bounded-by* (*?*$\mathcal{D}$ η) (*bound-outer* η) **for** η
by(*subst map-converter-id-move-left*)(*interaction-bound*; *simp*)
have [*simp*]: *plossless-converter* ($\mathcal{I}$*-inner* η) (*map-*$\mathcal{I}$ $(f\ \eta)\ (g\ \eta)$ ($\mathcal{I}$*-middle* η)) 1_C **for** η
using *plossless-id-converter* - $\mathcal{I}$[*of* η] **by**(*rule plossless-converter-mono*) *auto*
from *lossless*
have *plossless-gpv* ($\mathcal{I}$*-middle* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) (*?*$\mathcal{D}$ 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*: $\bigwedge\eta.$ *wiring* ($\mathcal{I}$*-middle* η) ($\mathcal{I}$*-real* η) (*cnv-outer* η) (*w2* η)
and *adv2*: *negligible* ($\lambda\eta.$ *advantage* (*?*$\mathcal{D}$ η) (*middle* η) (*cnv-outer* $\eta \mid_= 1_C \rhd$ *real* η))
by *auto*
from *w2* **have** [*WT-intro*]: $\mathcal{I}$*-middle* η, $\mathcal{I}$*-real* $\eta \vdash_C$ *cnv-outer* $\eta\ \surd$ **for** η **by** *cases*

show *wiring* ($\mathcal{I}$*-inner* η) ($\mathcal{I}$*-real* η) (*?cnv* η) (*w1* $\eta \circ_w$ *w2* η) **for** η
using *w1 w2* **by**(*rule wiring-comp-converterI*)

have *eq1′*: *connect* ($\mathcal{D}$ η) (*cnv-inner* $\eta \mid_= 1_C \rhd$ *middle* η) = *connect* (*?*$\mathcal{D}$ η) (*middle* η) **for** η
unfolding *distinguish-attach*[*symmetric*]
by(*rule connect-eq-resource-cong WT-intro eq-*$\mathcal{I}$*-attach-on′ parallel-converter2-eq-*$\mathcal{I}$*-cong eq1 eq-*$\mathcal{I}$*-converter-reflI order-refl*)+
have *eq2′*: *connect* (*?*$\mathcal{D}$ η) (*cnv-outer* $\eta \mid_= 1_C \rhd$ *real* η) = *connect* ($\mathcal{D}$ η)

(*?cnv* η $|_=$ $1_C \odot 1_C \rhd$ *real* η) **for** η
unfolding *distinguish-attach*[*symmetric*] *attach-compose comp-converter-parallel2*[*symmetric*]
by(*rule connect-eq-resource-cong WT-intro eq-*$\mathcal{I}$*-attach-on′ parallel-converter2-eq-*$\mathcal{I}$*-cong eq1*[*symmetric*] *eq-*$\mathcal{I}$*-converter-reflI order-refl*|*simp*)+

show *negligible* ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal* η) (*?cnv* η $|_=$ $1_C \rhd$ *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 $|_=$ *conv* η $\rhd$ *real-resource* η) ($\lambda\eta$. 1_C $|_=$ *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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common′* η) ($\mathcal{A}$ η) **for** η

let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{A}$ η) (1_C $|_=$ *conv* η)

have *ideal*: *connect* ($\mathcal{A}$ η) (*sim* η $|_=$ $1_C \rhd 1_C$ $|_=$ *conv* η $\rhd$ *ideal-resource* η) = *connect* (*?*$\mathcal{A}$ η) (*sim* η $|_=$ $1_C \rhd$ *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 $|_=$ *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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ *?*$\mathcal{A}$ η $\surd$ **for** η **by**(*rule WT-intro*)+
moreover have *interaction-any-bounded-by* (*?*$\mathcal{A}$ η) (*bound* η) **for** η
by *interaction-bound-converter*(*use bound-le*[*of* η] **in** ‹*simp add*: *max.commute*›)
moreover have *plossless-gpv* ($\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) (*absorb* ($\mathcal{A}$ η) (1_C $|_=$ *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 $|_=$ $1_C \rhd 1_C$ $|_=$ *conv* η $\rhd$ *ideal-resource* η) (1_C $|_=$ *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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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 $|_=$ 1_C $\rhd$ *real-resource* η))
by *blast*
show $\exists$ *cnv*. $\forall\mathcal{D}$. ($\forall\eta$. $\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common'* $\eta \vdash g$ $\mathcal{D}$ η $\surd$) $\longrightarrow$
($\forall\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound'* η)) $\longrightarrow$
($\forall\eta$. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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) (1_C $|_=$ *conv* η $\rhd$ *ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ 1_C $|_=$ *conv* η $\rhd$ *real-resource* η))
proof(*intro exI conjI strip*)
fix $\mathcal{D}$ :: *security* $\Rightarrow$ (*'c* + *'g*, *'d* + *'h*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall\eta$. $\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common'* $\eta \vdash g$ $\mathcal{D}$ η $\surd$
and *bound* [*rule-format*, *interaction-bound*]: $\forall\eta$. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound'* η)
and *lossless* [*rule-format*]: $\forall\eta$. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common'* η) ($\mathcal{D}$ η)

let *?*$\mathcal{D}$ = $\lambda\eta$. *absorb* ($\mathcal{D}$ η) (1_C $|_=$ *conv* η)
have *WT-D'* [*WT-intro*]: $\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash g$ *?*$\mathcal{D}$ η $\surd$ **for** η **by**(*rule WT-intro*)+
have *bound'*: *interaction-any-bounded-by* (*?*$\mathcal{D}$ η) (*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* ($\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) (*?*$\mathcal{D}$ η) **if** *lossless* **for** η
by(*rule lossless that WT-intro plossless-intro*)+
from *correct'*[*OF WT-D' bound' this*]
have *w1*: *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η)
and *adv'*: *negligible* ($\lambda\eta$. *advantage* (*?*$\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η)) **for** η
by *auto*
show *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η) **for** η **by**(*rule w1*)
have *cnv* η $|_=$ 1_C $\rhd$ 1_C $|_=$ *conv* η $\rhd$ *real-resource* η = 1_C $|_=$ *conv* η $\rhd$ *cnv* η $|_=$ 1_C $\rhd$ *real-resource* η **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* ($\mathcal{D}$ η) (1_C $|_=$ *conv* η $\rhd$ *ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ 1_C $|_=$ *conv* η $\rhd$ *real-resource* η))
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* ⇒ ('*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* ($\mathcal{A}$ η) (1_C $|_=$ 1_C ⊳ *res* η) = *connect* ($\mathcal{A}$ η) (1_C ⊳ *res* η) **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* ($\mathcal{A}$ η) (1_C $|_=$ 1_C ⊳ *res* η) (*res* η))
unfolding *advantage-def* **by** *simp*
next
show ∃ *cnv*. ∀ $\mathcal{D}$. (∀ η. $\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta \vdash g\ \mathcal{D}\ \eta\ \surd$) ⟶
(∀ η. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound* η)) ⟶
(∀ η. *lossless* ⟶ *plossless-gpv* ($\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta$) ($\mathcal{D}$ η)) ⟶
(∀ η. *wiring* ($\mathcal{I}$ η) ($\mathcal{I}$ η) (*cnv* η) (*id*, *id*)) ∧
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*res* η) (*cnv* η $|_=$ 1_C ⊳ *res* η))
proof(*intro exI strip conjI*)
fix $\mathcal{D}$ :: *security* ⇒ ('*a* + '*b*, '*c* + '*d*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: ∀ η. $\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta \vdash g\ \mathcal{D}\ \eta\ \surd$
and *bound* [*rule-format*, *interaction-bound*]: ∀ η. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound* η)
and *lossless* [*rule-format*]: ∀ η. *lossless* ⟶ *plossless-gpv* ($\mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common$ η) ($\mathcal{D}$ η)
show *wiring* ($\mathcal{I}$ η) ($\mathcal{I}$ η) 1_C (*id*, *id*) **for** η **by** *simp*
have *connect* ($\mathcal{D}$ η) (1_C $|_=$ 1_C ⊳ *res* η) = *connect* ($\mathcal{D}$ η) (1_C ⊳ *res* η) **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* ($\mathcal{D}$ η) (*res* η) (1_C $|_=$ 1_C ⊳ *res* η))
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* ⟹ *lossless-resource* ($\mathcal{I}$*-real1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common1* η) (*real1* η)
and *lossless-sim2* [*plossless-intro*]: $\bigwedge\eta$. *lossless1* ⟹ *plossless-converter* ($\mathcal{I}$*-real2* η) ($\mathcal{I}$*-inner2* η) (*sim2* η)

and *lossless-ideal2* [*plossless-intro*]: $\bigwedge\eta$. *lossless1* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$*-inner2* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) (*ideal2* η)
shows *constructive-security* ($\lambda\eta$. *parallel-wiring* $\rhd$ *real1* $\eta \parallel$ *real2* η) ($\lambda\eta$. *parallel-wiring* $\rhd$ *ideal1* $\eta \parallel$ *ideal2* η) ($\lambda\eta$. *sim1* η $|_=$ *sim2* η)
($\lambda\eta$. $\mathcal{I}$*-real1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) ($\lambda\eta$. $\mathcal{I}$*-inner1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) ($\lambda\eta$. $\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η)
($\lambda\eta$. *min* (*bound1* η) (*bound2* η)) (*lossless1* $\vee$ *lossless2*) ($\lambda\eta$. *w1* η $|_w$ *w2* η)
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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) $\vdash$*res parallel-wiring* $\rhd$ *real1* $\eta \parallel$ *real2* η $\surd$
and ($\mathcal{I}$*-inner1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) $\vdash$*res parallel-wiring* $\rhd$ *ideal1* $\eta \parallel$ *ideal2* η $\surd$
and $\mathcal{I}$*-real1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η, $\mathcal{I}$*-inner1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* $\eta \vdash_C$ *sim1* η $|_=$ *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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η)))
UNIV,($\mathcal{I}$*-real1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common1* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-inner2* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* **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* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η)) $\vdash_R$
((1_C $|_=$ *sim2* η) $|_=$ 1_C $|_=$ 1_C) $\odot$ *parallel-wiring* $\rhd$ *real1* $\eta \parallel$ *ideal2* η $\sim$
parallel-wiring $\odot$ (*converter-of-resource* (*real1* η) $|_\propto$ 1_C) $\rhd$ *sim2* η $|_=$ 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 $|_=$ *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-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 $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\|$ *ideal2* η) =
connect (*?*$\mathcal{A}$ η) (*sim2* η $|_=$ 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-$\mathcal{I}$-attach-on*[**where** *conv′*=((1_C $|_=$ *sim2* η) $|_=$ 1_C $|_=$ 1_C) $\odot$ *parallel-wiring*])
((*rule WT-intro*)+ | *intro* $*$ | *intro* $**$)+
qed (*rule WT-intro*)+
have *real2*: *connect* ($\mathcal{A}$ η) (*parallel-wiring* $\rhd$ *real1* η $\|$ *real2* η) = *connect* (*?*$\mathcal{A}$ η) (*real2* η) **for** η
unfolding *distinguish-attach*[*symmetric*]
by(*simp add*: *attach-compose attach-converter-of-resource-conv-parallel-resource*)
have $\mathcal{I}$*-real2* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common2* η $\vdash g$ *?*$\mathcal{A}$ η $\surd$ **for** η **by**(*rule WT-intro*)+
moreover have *interaction-any-bounded-by* (*?*$\mathcal{A}$ η) (*bound2* η) **for** η
by *interaction-bound-converter simp*
moreover have *plossless-gpv* ($\mathcal{I}$*-real2* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common2* η) (*?*$\mathcal{A}$ η) **if** *lossless2* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
ultimately
have *negl2*: *negligible* ($\lambda\eta$. *advantage* ($\mathcal{A}$ η)
((1_C $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\|$ *ideal2* η)
(*parallel-wiring* $\rhd$ *real1* η $\|$ *real2* η))
unfolding *advantage-def ideal2 real2* **by**(*rule sec2.adv*[*unfolded advantage-def*])

let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*parallel-wiring* $\odot$ *parallel-converter* 1_C (*converter-of-resource* (*sim2* η $|_=$ 1_C $\rhd$ *ideal2* η)))
have *ideal1*:
connect ($\mathcal{A}$ η) ((*sim1* η $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *ideal1* η $\|$ *ideal2* η) =
connect (*?*$\mathcal{A}$ η) (*sim1* η $|_=$ 1_C $\rhd$ *ideal1* η) **for** η
proof −
have $*$: $\mathcal{I}$*-uniform* ((*outs-*$\mathcal{I}$ ($\mathcal{I}$*-real1* η) <+> *outs-*$\mathcal{I}$ ($\mathcal{I}$*-real2* η)) <+> *outs-*$\mathcal{I}$ ($\mathcal{I}$*-common1* η) <+>
outs-$\mathcal{I}$ ($\mathcal{I}$*-common2* η)) *UNIV*,($\mathcal{I}$*-inner1* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common1* η) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-inner2* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common2* η) $\vdash_C$
((*sim1* η $|_=$ *sim2* η) $|_=$ 1_C) $\odot$ *parallel-wiring* $\sim$ ((*sim1* η $|_=$ *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 comp-parallel-wiring′ attach-compose attach-parallel2 attach-converter-of-resource-conv-parallel-resource2*

intro: *WT-intro parallel-converter2-id-id*[*symmetric*] *eq-$\mathcal{I}$-converter-reflI parallel-converter2-eq-$\mathcal{I}$-cong eq-$\mathcal{I}$-converter-mono*)

have ∗∗:((*outs-$\mathcal{I}$* (*$\mathcal{I}$-real1* η) <+> *outs-$\mathcal{I}$* (*$\mathcal{I}$-real2* η)) <+> *outs-$\mathcal{I}$* (*$\mathcal{I}$-common1* η) <+> *outs-$\mathcal{I}$* (*$\mathcal{I}$-common2* η)) $\vdash_R$
(*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-$\mathcal{I}$-attach-on*[**where** *conv'*=((*sim1* η $|_=$ *sim2* η) $|_=$ *1*$_C$ $|_=$ *1*$_C$) ⊙ *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*$_C$ ▷ *parallel-wiring* ▷ *real1* η ∥ *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*$_C$) ⊙ *parallel-wiring* ∼ ((*1*$_C$ $|_=$ *sim2* η) $|_=$ *1*$_C$ $|_=$ *1*$_C$) ⊙ *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 ⊙ ((*1*$_C$ $|_=$ *1*$_C$) $|_=$ *sim2* η $|_=$ *1*$_C$) ▷ *real1* η ∥ *ideal2* η ∼
parallel-wiring ⊙ (*1*$_C$ $|_\propto$ *converter-of-resource* (*sim2* η $|_=$ *1*$_C$ ▷ *ideal2* η)) ▷ *real1* η
by(*rule eq-resource-on-trans*, *rule eq-$\mathcal{I}$-attach-on*[**where** *conv'*=*parallel-wiring* ⊙ (*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 $\|$ *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 $\|$ *ideal2* η)
((1_C $|_=$ *sim2* η) $|_=$ 1_C $\rhd$ *parallel-wiring* $\rhd$ *real1* η $\|$ *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 $\|$ *ideal2* η)
(*parallel-wiring* $\rhd$ *real1* η $\|$ *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* η $\vdash g$ $\mathcal{D}$ 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* η $\vdash g$ $\mathcal{D}$ 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) $\oplus_{\mathcal{I}}$

$(\mathcal{I}\text{-}common1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common2\ \eta))\ (\mathcal{D}\ \eta)) \longrightarrow$
$(\forall \eta.\ wiring\ (\mathcal{I}\text{-}inner1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}inner2\ \eta)\ (\mathcal{I}\text{-}real1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}real2\ \eta)\ (cnv\ \eta)\ (w1\ \eta \mid_w w2\ \eta)) \wedge$
$negligible\ (\lambda\eta.\ advantage\ (\mathcal{D}\ \eta)\ (parallel\text{-}wiring \rhd ideal1\ \eta \parallel ideal2\ \eta)\ (cnv\ \eta \mid_= 1_C \rhd parallel\text{-}wiring \rhd real1\ \eta \parallel real2\ \eta))$
proof(*intro exI strip conjI*)
fix $\mathcal{D}$ :: *security* ⇒ (('*e* + '*k*) + '*b* + '*h*, ('*f* + '*l*) + '*d* + '*j*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall \eta.\ (\mathcal{I}\text{-}inner1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}inner2\ \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-}common1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common2\ \eta) \vdash g\ \mathcal{D}\ \eta\ \surd$
and *bound* [*rule-format*, *interaction-bound*]: $\forall \eta.$ *interaction-any-bounded-by* $(\mathcal{D}\ \eta)$ (*min* (*bound1* η) (*bound2* η))
and *lossless* [*rule-format*, *plossless-intro*]: $\forall \eta.$ *lossless1* ∨ *lossless2* ⟶ *plossless-gpv* $((\mathcal{I}\text{-}inner1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}inner2\ \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-}common1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common2\ \eta))\ (\mathcal{D}\ \eta)$

let $?cnv = \lambda\eta.\ cnv1\ \eta \mid_= cnv2\ \eta$

let $?\mathcal{D}1 = \lambda\eta.\ absorb\ (\mathcal{D}\ \eta)\ (parallel\text{-}wiring \odot parallel\text{-}converter\ 1_C\ (converter\text{-}of\text{-}resource\ (ideal2\ \eta)))$
have *WT1*: $\mathcal{I}\text{-}inner1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common1\ \eta \vdash g\ ?\mathcal{D}1\ \eta\ \surd$ **for** η **by**(*rule WT-intro*)+
have *bound1*: *interaction-any-bounded-by* $(?\mathcal{D}1\ \eta)$ (*bound1* η) **for** η **by** *interaction-bound simp*
have *plossless-gpv* $(\mathcal{I}\text{-}inner1\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common1\ \eta)\ (?\mathcal{D}1\ \eta)$ **if** *lossless1* **for** η
by(*rule plossless-intro WT-intro* | *simp add*: *that*)+
from *correct1*[*OF WT1 bound1 this*]
have *w1*: *wiring* $(\mathcal{I}\text{-}inner1\ \eta)\ (\mathcal{I}\text{-}real1\ \eta)\ (cnv1\ \eta)\ (w1\ \eta)$
and *adv1*: $negligible\ (\lambda\eta.\ advantage\ (?\mathcal{D}1\ \eta)\ (ideal1\ \eta)\ (cnv1\ \eta \mid_= 1_C \rhd real1\ \eta))$ **for** η
by *auto*

from *w1* **obtain** *f g* **where** *fg*: $\bigwedge\eta.\ w1\ \eta = (f\ \eta,\ g\ \eta)$
and [*WT-intro*]: $\bigwedge\eta.\ \mathcal{I}\text{-}inner1\ \eta,\ \mathcal{I}\text{-}real1\ \eta \vdash_C cnv1\ \eta\ \surd$
and *eq1*: $\bigwedge\eta.\ \mathcal{I}\text{-}inner1\ \eta,\ \mathcal{I}\text{-}real1\ \eta \vdash_C cnv1\ \eta \sim map\text{-}converter\ id\ id\ (f\ \eta)\ (g\ \eta)\ 1_C$
apply *atomize-elim*
apply(*fold all-conj-distrib*)
apply(*subst choice-iff*[*symmetric*])+
apply(*fastforce elim*!: *wiring.cases*)
done
have $\mathcal{I}1$: $\mathcal{I}\text{-}inner1\ \eta \le map\text{-}\mathcal{I}\ (f\ \eta)\ (g\ \eta)\ (\mathcal{I}\text{-}real1\ \eta)$ **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*]: $\mathcal{I}\text{-}inner1\ \eta,\ map\text{-}\mathcal{I}\ (f\ \eta)\ (g\ \eta)\ (\mathcal{I}\text{-}real1\ \eta) \vdash_C 1_C\ \surd$ **for** η
by(*rule WT-converter-mono*)
have *lossless1* [*plossless-intro*]: *plossless-converter* $(\mathcal{I}\text{-}inner1\ \eta)\ (\mathcal{I}\text{-}real1\ \eta)\ (map\text{-}converter\ id\ id\ (f\ \eta)\ (g\ \eta)\ 1_C)$ **for** η
by(*rule plossless-map-converter*)(*rule plossless-intro order-refl* $\mathcal{I}1$ *WT-intro plossless-converter-mono* | *simp*)+

let *?𝒟2* = $\lambda\eta$. *absorb* ($\mathcal{D}$ η) (*parallel-wiring* ⊙ *parallel-converter* (*converter-of-resource* (*map-converter id id* (*f* η) (*g* η) *1*$_C$ $|_=$ *1*$_C$ ▷ *real1* η)) *1*$_C$)
have *WT2*: *ℐ-inner2* η $\oplus_{\mathcal{I}}$ *ℐ-common2* η ⊢*g* *?𝒟2* η √ **for** η **by**(*rule WT-intro* | *simp*)+
have *bound2*: *interaction-any-bounded-by* (*?𝒟2* η) (*bound2* η) **for** η **by** *interaction-bound simp*
have *plossless-gpv* (*ℐ-inner2* η $\oplus_{\mathcal{I}}$ *ℐ-common2* η) (*?𝒟2* 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*$_C$ ▷ *real2* η)) **for** η
by *auto*

from *w2* **have** [*WT-intro*]: *ℐ-inner2* η, *ℐ-real2* η $\vdash_C$ *cnv2* η √ **for** η **by** *cases*

have ∗: *connect* ($\mathcal{D}$ η) (*?cnv* η $|_=$ *1*$_C$ ▷ *parallel-wiring* ▷ *real1* η ∥ *real2* η) =
connect (*?𝒟2* η) (*cnv2* η $|_=$ *1*$_C$ ▷ *real2* η) **for** η
proof −
have *outs-ℐ* ((*ℐ-inner1* η $\oplus_{\mathcal{I}}$ *ℐ-inner2* η) $\oplus_{\mathcal{I}}$ (*ℐ-common1* η $\oplus_{\mathcal{I}}$ *ℐ-common2* η)) $\vdash_R$
?cnv η $|_=$ *1*$_C$ ▷ *parallel-wiring* ▷ *real1* η ∥ *real2* η ∼
(*map-converter id id* (*f* η) (*g* η) *1*$_C$ $|_=$ *cnv2* η) $|_=$ (*1*$_C$ $|_=$ *1*$_C$) ▷ *parallel-wiring* ▷ *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*$_C$) ▷ *parallel-wiring* ▷ *real1* η ∥ *real2* η =
parallel-wiring ▷ (*map-converter id id* (*f* η) (*g* η) *1*$_C$ $|_=$ *1*$_C$ ▷ *real1* η) ∥ (*cnv2* η $|_=$ *1*$_C$ ▷ *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*$_C$ ▷ *real1* η) **for** η
proof −
have *connect* (*?𝒟2* η) (*ideal2* η) =
connect ($\mathcal{D}$ η) (*parallel-wiring* ▷ ((*map-converter id id* (*f* η) (*g* η) *1*$_C$ $|_=$ *1*$_C$) $|_=$ *1*$_C$) ▷ (*real1* η ∥ *ideal2* η))
by(*simp add*: *distinguish-attach*[*symmetric*] *attach-converter-of-resource-conv-parallel-resource attach-compose attach-parallel2*)
also have . . . = *connect* ($\mathcal{D}$ η) (*parallel-wiring* ▷ ((*cnv1* η $|_=$ *1*$_C$) $|_=$ *1*$_C$) ▷ (*real1* η ∥ *ideal2* η))
unfolding *attach-compose*[*symmetric*] **using** *WT-D*
by(*rule connect-eq-resource-cong*[*symmetric*])

(*rule eq-$\mathcal{I}$-attach-on′ WT-intro eq-$\mathcal{I}$-comp-cong eq-$\mathcal{I}$-converter-reflI parallel-converter2-eq-$\mathcal{I}$-cong eq1* | *simp*)+
also have ... = *connect* (*?$\mathcal{D}$1 η*) (*cnv1 η* $|_=$ 1_C ▷ *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* (*?$\mathcal{D}$1 η*) (*ideal1 η*) = *connect* ($\mathcal{D}$ *η*) (*parallel-wiring* ▷ *ideal1 η* ∥ *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 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*) **for** *η*
using *w1 w2* **by**(*rule wiring-parallel-converter2*)
from *negligible-plus*[*OF adv1 adv2*]
show *negligible* (*λη. advantage* ($\mathcal{D}$ *η*) (*parallel-wiring* ▷ *ideal1 η* ∥ *ideal2 η*) (*?cnv η* $|_=$ 1_C ▷ *parallel-wiring* ▷ *real1 η* ∥ *real2 η*))
by(*rule negligible-le*)(*simp add*: *advantage-def* ∗ ∗∗ ∗∗∗)
qed
qed

theorem (**in** *constructive-security*) *parallel-realisation1*:
assumes *WT-res*: ⋀*η*. $\mathcal{I}$*-res η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common′ η* ⊢*res res η* √
and *lossless-res*: ⋀*η*. *lossless* ⟹ *lossless-resource* ($\mathcal{I}$*-res η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common′ η*) (*res η*)
shows *constructive-security* (*λη. parallel-wiring* ▷ *res η* ∥ *real-resource η*)
(*λη. parallel-wiring* ▷ (*res η* ∥ *ideal-resource η*)) (*λη. parallel-converter2 id-converter* (*sim η*))
(*λη.* $\mathcal{I}$*-res η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-real η*) (*λη.* $\mathcal{I}$*-res η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-ideal η*) (*λη.* $\mathcal{I}$*-common′ η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common η*) *bound lossless* (*λη.* (*id*, *id*) $|_w$ *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*: ⋀*η*. $\mathcal{I}$*-res η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common′ η* ⊢*res res η* √
and *lossless-res*: ⋀*η*. *lossless* ⟹ *lossless-resource* ($\mathcal{I}$*-res η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common′ η*) (*res η*)
shows *constructive-security* (*λη. parallel-wiring* ▷ *real-resource η* ∥ *res η*)
(*λη. parallel-wiring* ▷ (*ideal-resource η* ∥ *res η*)) (*λη. parallel-converter2* (*sim η*) *id-converter*)
(*λη.* $\mathcal{I}$*-real η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-res η*) (*λη.* $\mathcal{I}$*-ideal η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-res η*) (*λη.* $\mathcal{I}$*-common η* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common′ η*) *bound lossless* (*λη. w η* $|_w$ (*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* η $\vdash$*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* $\rhd$ *res* η $\parallel$ *real-resource* η)
($\lambda\eta$. *parallel-resource1-wiring* $\rhd$ *res* η $\parallel$ *ideal-resource* η) *sim*
$\mathcal{I}$-real *$\mathcal{I}$-ideal* ($\lambda\eta$. *$\mathcal{I}$-res* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-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* ▷ *res* η ‖ *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* ($\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) (*?*$\mathcal{A}$ 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 $|_=$ 1_C ▷
parallel-resource1-wiring ▷ *res* η ‖ *ideal-resource* η)
(*parallel-resource1-wiring* ▷ *res* η ‖ *real-resource* η))
unfolding *advantage-def ideal real* **by**(*rule adv*[*unfolded advantage-def*])
next
from *correct* **obtain** *cnv*
where *correct'*: $\bigwedge\mathcal{D}$. ⟦ $\bigwedge\eta$. $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* ($\mathcal{I}$*-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 $|_=$ 1_C ▷ *real-resource* η))
by *blast*
show $\exists$ *cnv*. $\forall\mathcal{D}$. ($\forall\eta$. $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* 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* η)) $\longrightarrow$
($\forall\eta$. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* 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) (*parallel-resource1-wiring* ▷ *res* η ‖ *ideal-resource* η)
(*cnv* η $|_=$ 1_C ▷ *parallel-resource1-wiring* ▷ *res* η ‖ *real-resource* η))
proof(*intro exI conjI strip*)
fix $\mathcal{D}$ :: *security* ⇒ (*'c* + *'g* + *'e*, *'d* + *'h* + *'f*) *distinguisher*
assume *WT-D* [*rule-format*, *WT-intro*]: $\forall\eta$. $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* ($\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η)) ($\mathcal{D}$ η)

let *?*$\mathcal{D}$ = $\lambda\eta$. *absorb* ($\mathcal{D}$ η) (*swap-lassocr* ⊙ *parallel-converter* (*converter-of-resource* (*res* η)) 1_C)
have *WT'*: $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*g* *?*$\mathcal{D}$ η $\surd$ **for** η **by**(*rule WT-intro*)+
have *bound'*: *interaction-any-bounded-by* (*?*$\mathcal{D}$ η) (*bound* η) **for** η **by** *interaction-bound simp*
have *lossless'*: *plossless-gpv* ($\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) (*?*$\mathcal{D}$ η) **if** *lossless* **for** η
by(*rule plossless-intro WT-intro that*)+

from *correct'*[*OF WT' bound' lossless'*]
have *w*: *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η)
and *adv*: *negligible* ($\lambda\eta$. *advantage* (*?*$\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ *1*$_C$ $\rhd$ *real-resource* η))
for η **by** *auto*
show *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η) **for** η **by**(*rule w*)
from *w* **have** [*WT-intro*]: $\mathcal{I}$*-ideal* η, $\mathcal{I}$*-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* $\oplus_O$ *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* ≡ *ForwardOrEdit 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* $\oplus_O$ *send-oracle* $\oplus_O$ *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* ≡ (1_C |= $lassocr_C$) ⊙ *swap-lassocr* ⊙ (1_C |= (1_C |= *swap-lassocr*) ⊙ *swap-lassocr*) ⊙ $rassocl_C$

definition *res* = (1_C |= *enc* |= *dec*) ▷ (1_C |= *parallel-wiring*) ▷ *parallel-resource1-wiring* ▷ (*KEY.res* ∥ *AUTH.res*)

lemma *res-alt-def*: *res* = ((1_C |= *enc* |= *dec*) ⊙ (1_C |= *parallel-wiring*)) ▷ *parallel-resource1-wiring* ▷ (*KEY.res* ∥ *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* ⇒ ′*m* ⇒ ′*a spmf*
begin

definition *enm* :: (′*m*, *unit*, ′*m* + (′*a* × ′*m*), ′*r* + *unit*) *converter* **where**
enm ≡ *CNV* (λ*bs m*. *if bs*
then Done ((), *True*)
else do {
r ← *Pause* (*Inl m*) *Done*;
a ← *lift-spmf* (*mac-alg* (*projl r*) *m*);
(- :: ′*r* + *unit*) ← *Pause* (*Inr* (*a*, *m*)) *Done*;
Done ((), *True*)
}) *False*

definition *dem* :: (*unit*, ′*m option*, ′*m* + *unit*, ′*r* + (′*a* × ′*m*) *option*) *converter*
where
dem ≡ *CNV* (*stateless-callee* (λ-. *Pause* (*Inr* ()) (λ*am*′.
case am′ *of Inr* (*Some* (*a*, *m*)) ⇒ (*do* {
r ← *Pause* (*Inl m*) *Done*;
a′ ← *lift-spmf* (*mac-alg* (*projl r*) *m*);
Done (*if a*′ = *a then Some m else None*) })
| - ⇒ *Done None*)

)) ()

definition πE :: $(('a \times 'm)$ *insec-query*, $('a \times 'm)$ *option*, $('a \times 'm)$ *insec-query*, $('a \times 'm)$ *option*) *converter* $(\langle \pi^E \rangle)$ **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-lassocr*$) \odot$ *swap-lassocr*$) \odot rassocl_C$

definition *res* $= (1_C \mid_=$ *enm* $\mid_=$ *dem*$) \rhd (1_C \mid_=$ *parallel-wiring*$) \rhd$ *parallel-resource1-wiring* $\rhd$ (*RO.res* $\parallel$ *INSEC.res*)

end

lemma *interface-wiring*:
(*cnv-advr* $\mid_=$ *cnv-send* $\mid_=$ *cnv-recv*) $\rhd (1_C \mid_=$ *parallel-wiring*$) \rhd$ *parallel-resource1-wiring* $\rhd$
(*RES* (*res2-send* $\oplus_O$ *res2-recv*) *res2-s* $\parallel$ *RES* (*res1-advr* $\oplus_O$ *res1-send* $\oplus_O$ *res1-recv*) *res1-s*)
=
cnv-advr $\mid_=$ *cnv-send* $\mid_=$ *cnv-recv* $\rhd$
RES ($\dagger$*res1-advr* $\oplus_O$ (*res2-send*$\dagger$ $\oplus_O$ $\dagger$*res1-send*) $\oplus_O$ *res2-recv*$\dagger$ $\oplus_O$ $\dagger$*res1-recv*) (*res2-s*, *res1-s*)
(**is** - $\rhd$ *?L1* $\rhd$ *?L2* $\rhd$ *?L3* = - $\rhd$ *?R*)
proof −
let *?wiring* = (*id*, *id*) $\mid_w$ ($lassocr_w \circ_w$ ((*id*, *id*) $\mid_w$ ($rassocl_w \circ_w$ ($swap_w \mid_w$ (*id*, *id*) $\circ_w lassocr_w$))
$\circ_w rassocl_w$)) $\circ_w$ ($rassocl_w \circ_w$ ($swap_w \mid_w$ (*id*, *id*) $\circ_w lassocr_w$))

have *?L1* $\rhd$ *?L2* $\rhd$ *?L3* = *?L1* $\odot$ *?L2* $\rhd$
RES ((*res2-send*$\dagger$ $\oplus_O$ *res2-recv*$\dagger$) $\oplus_O$ $\dagger$*res1-advr* $\oplus_O$ $\dagger$*res1-send* $\oplus_O$ $\dagger$*res1-recv*) (*res2-s*, *res1-s*) (**is** - = - $\rhd$ *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* (λ*key*. (*Some key*, *Store* (*key* [⊕] *plain*))) (*spmf-of-set* (*nlists UNIV* η)))
if *length plain* = *id'* η
| *S* (*return-spmf* (*None*, (), *Collect plain*))
(*map-spmf* (λ*key*. (*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* (λ*x*. (*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*) $\vdash_R$
RES (*callee-sec-channel sim* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*) (*None* :: *bool list option*, (), *Void*)
≈
RES (†*auth-channel.auth-oracle* $\oplus_O$ *key-channel-send* $\oplus_O$ *key-channel-recv*) (*None* :: *bool list option*, *Void*)
(**is** *?A* $\vdash_R$ *RES* (*?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*) *?SL* ≈ *RES* (*?R1* $\oplus_O$ *?R2* $\oplus_O$ *?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* $\vdash_C$ *?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*(*?SL*) ≈ *?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
    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* (λ*x*. (*Inl* (*Some x*), *Some x*, (), *Store plain*))
(*spmf-of-set* (*nlists UNIV* (*id′ η*)))) (*Inl* (*Some* (*key* [⊕] *plain*))))
(*cond-spmf-fst* (*map-spmf* (λ*x*. (*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* $\oplus_O$ *key-channel-send* $\oplus_O$ *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.π*E-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* $_C$ $\rhd$ *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* $\rhd$ - = *?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* $\vdash c$ (*sec-channel.sec-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* $\rhd$ *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-*$\mathcal{I}$*-attach*[*OF WT-resource-of-oracle*, **where** *?*$\mathcal{I}'$ = *?I* **and** *?conv*=*?L1* **and** *?conv'*=*?L1*]) *simp-all*
qed

end

lemma *eq-$\mathcal{I}$-gpv-Done1*:
eq-$\mathcal{I}$-gpv A $\mathcal{I}$ (Done x) gpv $\longleftrightarrow$ *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 $\mathcal{I}$) (Pure x) a*)
by(*auto intro*: *eq-$\mathcal{I}$-gpv.intros simp add*: *rel-spmf-return-spmf1 elim*: *eq-$\mathcal{I}$-gpv.cases*)

lemma *eq-$\mathcal{I}$-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* $|_=$ *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*: *?*$\mathcal{I}$*-dec* η $\vdash$*c S-CHAN.recv-oracle s* $\surd$ **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*: *?*$\mathcal{I}$*-enc* η $\vdash$*c S-CHAN.send-oracle s* $\surd$ **for** η *s*
by(*rule WT-calleeI*)(*auto*)
have *: *callee-invariant-on* (*CIPHER.AUTH.auth-oracle* $\oplus_O$ *S-CHAN.send-oracle* $\oplus_O$ *S-CHAN.recv-oracle*) (*?I* η)
($\mathcal{I}$*-real* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-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*: $\mathcal{I}$*-common-def in-nlists-UNIV*)
subgoal by(*auto simp add*: $\mathcal{I}$*-common-def WT-auth WT-recv intro*: *WT-calleeI*)
done
have *i4* [*unfolded* $\mathcal{I}$*-common-def*, *WT-intro*]: $\mathcal{I}$*-real* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η $\vdash$*res CIPHER.AUTH.res* $\surd$ **for** η
unfolding *CIPHER.AUTH.res-def* **by**(*rule callee-invariant-on.WT-resource-of-oracle*[*OF* *]) *simp*
show *cipher*: $\mathcal{I}$*-real* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η $\vdash$*res CIPHER.res* η $\surd$ **for** η
unfolding *CIPHER.res-def* **by**(*rule WT-intro i3*)+

show *secure*: $\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η $\vdash$*res S-CHAN.res* $\surd$ **for** η
proof −
have[*simp*]:$\mathcal{I}$*-ideal* η $\vdash$*c S-CHAN.sec-oracle s* $\surd$ **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*: $\mathcal{I}$*-ideal-def*)
qed
have[*simp*]: $\mathcal{I}$*-common* η $\vdash$*c* (*S-CHAN.send-oracle* $\oplus_O$ *S-CHAN.recv-oracle*) *s* $\surd$
if *pred-cstate* (λx. *length x* = η) *s* **for** *s*
unfolding $\mathcal{I}$*-common-def* **by**(*rule WT-plus-oracleI WT-send WT-recv that*)+

have *: *callee-invariant-on* (*S-CHAN.sec-oracle* $\oplus_O$ *S-CHAN.send-oracle* $\oplus_O$ *S-CHAN.recv-oracle*) (*?I* η) ($\mathcal{I}$*-ideal* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-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*: $\mathcal{I}$*-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*]: *$\mathcal{I}$-real* η, *$\mathcal{I}$-ideal* η $\vdash_C$ *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* √ **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*: ⋀η. *$\mathcal{I}$-real* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common* η ⊢*g* $\mathcal{A}$ η √
and *bound*: ⋀η. *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 ⊢*res CIPHER.res* η √ **by**(*rule cipher*)
show *$\mathcal{I}$-real* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common* η ⊢*res CNV sim None* $|_=$ 1_C ⊳ *S-CHAN.res* √
by(*rule WT-intro secure* | *simp*)+
qed
then show *negligible* (λη. *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 √
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* √ **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 η* $\vdash_C$ *?cnv* ⊙ *CNV sim None* √ **for** *η* **by**(*rule WT WT-intro*)+ *simp*
with - **have** *wiring* (*$\mathcal{I}$-ideal η*) (*$\mathcal{I}$-ideal η*) (*?cnv* ⊙ *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 ∃ *cnv*. ∀ *η*. *wiring* (*$\mathcal{I}$-ideal η*) (*$\mathcal{I}$-real η*) (*cnv η*) *auth-sec-wiring* ∧
wiring (*$\mathcal{I}$-ideal η*) (*$\mathcal{I}$-ideal η*) (*cnv η* ⊙ *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* ⇒ *bool list set* **where**
rnd η ≡ *nlists UNIV η*

definition *mac* :: *security* ⇒ *bool list* ⇒ *bool list* ⇒ *bool list spmf* **where**
mac η r m ≡ *return-spmf r*

definition *vld* :: *security* ⇒ *bool list set* **where**
vld η ≡ *nlists UNIV η*

fun *valid-mac-query* :: *security* ⇒ (*bool list* × *bool list*) *insec-query* ⇒ *bool* **where**
valid-mac-query η (*ForwardOrEdit* (*Some* (*a*, *m*))) ⟷ *a* ∈ *vld η* ∧ *m* ∈ *vld η*
| *valid-mac-query η* - = *True*

fun *sim* :: (*'b list* × *'b list*) *option* + *unit* ⇒ (*'b list* × *'b list*) *insec-query*
⇒ ((*'b list* × *'b list*) *option* × ((*'b list* × *'b list*) *option* + *unit*), *auth-query* , *'b list option*) *gpv* **where**
sim (*Inr* ()) - = *Done* (*None*, *Inr*())
| *sim* (*Inl None*) (*Edit* (*a'*, *m'*)) = *do* { - ← *Pause Drop Done*; *Done* (*None*, *Inr* ())}
| *sim* (*Inl* (*Some* (*a*, *m*))) (*Edit* (*a'*, *m'*)) = (*if a* = *a'* ∧ *m* = *m'*

```
    then do { - ← Pause Forward Done; Done (None, Inl (Some (a, m)))}
    else do { - ← 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* (λ*a*. ((*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* (λ*a*. ((*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* (λ*a*. ((*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*) $\vdash_R$
RES (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*)
≈
RES (††*insec-channel.insec-oracle* $\oplus_O$ *rorc-channel-send* $\oplus_O$ *rorc-channel-recv*) ((*False*, ()), *Map.empty*, *Void*)
(**is** *?A* $\vdash_R$ *RES* (*?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*) *?SL* ≈ *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

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 $\times$ *nlists* (*UNIV*::*bool set*) η $\cap$ $\{x.\ snd\ x \neq a'\} \neq \{\}$ (**is** *?aux1*)
and *aux2*: *nlists* (*UNIV*::*bool set*) η $\cap$ $\{x.\ x \neq a'\} \neq \{\}$ (**is** *?aux2*) **for** a'
proof −
have $\exists\, a.\ a \in$ *nlists* (*UNIV*::*bool set*) $\eta \wedge a \neq a'$ **for** a'
proof (*cases* $a' \in$ *nlists* (*UNIV*::*bool set*) η)
case *True*
show *?thesis*
proof (*rule ccontr*)
have $2\ \hat{}\ \eta$ = *card* (*nlists* (*UNIV* :: *bool set*) η) **by** (*simp add*: *card-nlists*)
also assume $\nexists\, a.\ a \in$ *nlists UNIV* $\eta \wedge a \neq a'$
then have *nlists UNIV* $\eta = \{a'\}$ **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*:$a \in$ *nlists* (*UNIV*::*bool set*) η **using** *assms* **by** *auto*
then have $a \neq a'$ **using** *False* **by** *blast*
then show *?thesis* **using** *obt* **by** *auto*
qed
then obtain *a* **where** *o1*: $a \in$ *nlists* (*UNIV*::*bool set*) η **and** *o2*: $a \neq a'$ **by** *blast*

obtain *m* **where** $m \in$ *nlists* (*UNIV*::*bool set*) η **by** *blast*
with *o1 o2* **have** $(m,\ a) \in \{x.\ snd\ x \neq a'\}$ **and** $(m,\ a) \in$ *nlists UNIV* η $\times$ *nlists UNIV* η **by** *auto*
then show *?aux1* **by** *blast*

from *o1 o2* **have** $a \in \{x.\ x \neq a'\}$ **and** $a \in$ *nlists UNIV* η **by** *auto*
then show *?aux2* **by** *blast*
qed

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** (*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*]: $a \in$ *nlists UNIV* $\eta \Longrightarrow$
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*]: $a' \in$ *nlists UNIV* $\eta \Longrightarrow$ *nlists* (*UNIV* :: *bool set*) $\eta \times$ *nlists UNIV* $\eta \cap \{x.\ snd\ x = a'\}$ = *nlists UNIV* $\eta \times \{a'\}$
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*: $\mathcal{I}$ ⊢*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* (=) (λ*x 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* (=) (λ*x 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* (=) (λ*x 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* (=) (*λx y. x* = *snd y*))
((*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv-f*) (*ams*, *es*, *as*) *query*)
((*lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *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* (=) (*λx 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* (=) (*λx y. x* = *snd y*)]); *auto*)+
qed

have[*simp*]: *rel-spmf* (*λx y. fst x* = *fst y*)
(*exec-gpv ?orc-lft* $\mathcal{A}$ (*Void*, *None*, *Map.empty*)) (*exec-gpv ?orc-rgt* $\mathcal{A}$ (*False*, *Void*, *None*, *Map.empty*))
by(*rule rel-spmf-mono*[**where** *A*=*rel-prod* (=) (*λx 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* $\mathcal{A}$ (*RES* (*lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv-f′*) (*False*, *Void*, *None*, *Map.empty*))) *True* −
spmf (*connect* $\mathcal{A}$ (*RES* (*lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv′*) (*False*, *Void*, *None*, *Map.empty*))) *True*|
≤ *Sigma-Algebra.measure* (*measure-spmf* (*game* $\mathcal{A}$ *lazy-channel-recv-f′*)) {*x*.

snd x} (**is** |*spmf ?L* - − *spmf ?R* - | ≤ -)
proof −
let *?orc-lft* = *lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv-f′*

let *?orc-rgt* = *lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *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* $\oplus_{\mathcal{I}}$ (*ℐ-uniform* (*vld η*) *UNIV* $\oplus_{\mathcal{I}}$ *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** $\mathcal{I}$ = ?$\mathcal{I}$, THEN rel-spmf-mono])
(auto simp add: lazy-channel-insec′-def intro: WT-calleeI)
also let ?I = (λ(-, s1, s2, s3). pred-cstate (λx. 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′
supply [[simproc del: defined-all]]
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′
supply [[simproc del: defined-all]]
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: op-

tion.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′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv′*) (*False*, *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*
proof −
let *?orc-lft* = *lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv′*
let *?orc-rgt* = *lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *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* (=) (λ*x 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* (=) (λ*x 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* (=) (λ*x y*. *y* = *snd x*))
((*lazy-channel-insec′* $\oplus_O$ *lazy-channel-send′* $\oplus_O$ *lazy-channel-recv′*) (*flg*, *ams*, *es*, *as*) *query*)
((*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *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* (=) (λ*x 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* (=) (λ*x y*. *y* = *snd x*)]; *auto*)+
qed

have[*simp*]: *rel-spmf* (λ*x y*. *fst x* = *fst y*)
(*exec-gpv ?orc-lft* $\mathcal{A}$ (*False*, *Void*, *None*, *Map.empty*)) (*exec-gpv ?orc-rgt* $\mathcal{A}$ (*Void*, *None*, *Map.empty*))
by(*rule rel-spmf-mono*[**where** *A*=*rel-prod* (=) (λ*x 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'* $\oplus_O$ *lazy-channel-send'* $\oplus_O$ *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* $\leq$ *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-spmf* (*Inl None*, (), *Void*))
(*return-spmf* (*Void*, *None*, *Map.empty*))
| S' (*return-spmf* (*Inl None*, (), *Store m*))
(*return-spmf* (*Store m*, *None*, *Map.empty*))
if *length m* = *id*$'$ η
| S' (*return-spmf* (*Inr* (), (), *Collect m*))
(*return-spmf* (*Collect m*, *None*, *Map.empty*))
if *length m* = *id*$'$ η
| S' (*return-spmf* (*Inl* (*Some* (*a*, *m*)), (), *Store m*))
(*return-spmf* (*Store m*, *None*, [*m* $\mapsto$ *a*]))
if *length m* = *id*$'$ η
| S' (*return-spmf* (*Inr* (), (), *Collect m*))
(*return-spmf* (*Collect m*, *None*, [*m* $\mapsto$ *a*]))
if *length m* = *id*$'$ η
| S' (*return-spmf* (*Inr* (), (), *Fail*))
(*return-spmf* (*Fail*, *None*, *Map.empty*))
| S' (*return-spmf* (*Inr* (), (), *Fail*))
(*return-spmf* (*Fail*, *None*, [*m* $\mapsto$ *x*]))
if *length m* = *id*$'$ η
| S' (*return-spmf* (*Inr* (), (), *Void*))
(*return-spmf* (*Collect m*$'$, *Some* (*a*$'$, *m*$'$), *Map.empty*))
if *length m*$'$ = *id*$'$ η **and** *length a*$'$ = *id*$'$ η
| S' (*return-spmf* (*Inr* (), (), *Fail*))
(*return-spmf* (*Collect m*$'$, *Some* (*a*$'$, *m*$'$), *Map.empty*))
if *length m*$'$ = *id*$'$ η **and** *length a*$'$ = *id*$'$ η
| S' (*return-spmf* (*Inr* (), (), *Store m*))
(*return-spmf* (*Collect m*$'$, *Some* (*a*$'$, *m*$'$), *Map.empty*))
if *length m* = *id*$'$ η **and** *length m*$'$ = *id*$'$ η **and** *length a*$'$ = *id*$'$ η
| S' (*return-spmf* (*Inl* (*Some* (*a*$'$, *m*$'$)), (), *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* (*Inl None*, (), *cstate.Collect m*))
  (*return-spmf* (*cstate.Collect m*, *None*, *Map.empty*))
if *length m* $= id'\ \eta$
| S' (*return-spmf* (*Inl None*, (), *cstate.Fail*))
  (*return-spmf* (*cstate.Fail*, *None*, *Map.empty*))

| S' (*return-spmf* (*Inr* (), (), *Fail*))
  (*return-spmf* (*Collect* m', *Some* (a', m'), $[m \mapsto a]$))
if *length* $m = id'\ \eta$ **and** *length* $m' = id'\ \eta$ **and** *length* $a' = id'\ \eta$ **and** $m \neq m'$
| S' (*return-spmf* (*Inr* (), (), *Fail*))
  (*return-spmf* (*Collect* m', *Some* (a', m'), $[m \mapsto a]$))
if *length* $m = id'\ \eta$ **and** *length* $m' = id'\ \eta$ **and** *length* $a' = id'\ \eta$ **and** $a \neq a'$
| S' (*return-spmf* (*Inl None*, (), *Collect* m'))
  (*return-spmf* (*Collect* m', *Some* (a', m'), $[m' \mapsto a']$))
if *length* $m' = id'\ \eta$ **and** *length* $a' = id'\ \eta$
| S' (*return-spmf* (*Inr* (), (), *Collect* m'))
  (*return-spmf* (*Collect* m', *Some* (a', m'), $[m' \mapsto a']$))
if *length* $m' = id'\ \eta$ **and** *length* $a' = id'\ \eta$
| S' (*return-spmf* (*Inr* (), (), *Void*))
  (*map-spmf* ($\lambda a'$. (*Fail*, *None*, $[m' \mapsto a']$)) (*spmf-of-set* (*nlists UNIV* η)))
if *length* $m' = id'\ \eta$
| S' (*return-spmf* (*Inr* (), (), *Fail*))
  (*map-spmf* ($\lambda a'$. (*Fail*, *None*, $[m' \mapsto a']$)) (*spmf-of-set* (*nlists UNIV* η)))
if *length* $m' = id'\ \eta$
| S' (*return-spmf* (*Inr* (), (), *Store m*))
  (*map-spmf* ($\lambda a'$. (*Fail*, *None*, $[m' \mapsto a']$)) (*spmf-of-set* (*nlists UNIV* η)))
if *length* $m = id'\ \eta$ **and** *length* $m' = id'\ \eta$
| 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* $m = id'\ \eta$ **and** *length* $m' = id'\ \eta$ **and** $m \neq m'$
| S' (*return-spmf* (*Inl* (*Some* (a', m')), (), *Fail*))
  (*return-spmf* (*Fail*, *None*, $[m' \mapsto a']$))
if *length* $m' = id'\ \eta$ **and** *length* $a' = id'\ \eta$
| S' (*return-spmf* (*Inl None*, (), *Fail*))
  (*return-spmf* (*Fail*, *None*, $[m' \mapsto a']$))
if *length* $m' = id'\ \eta$ **and** *length* $a' = id'\ \eta$

private lemma *trace-eq-sim*:
  shows (*valid-insecQ* <+> *nlists UNIV* $(id'\ \eta)$ <+> *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 ⊢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 (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* $\oplus_O$ *rorc-channel-send* $\oplus_O$ *rorc-channel-recv*) ((*False*, ()), *Map.empty*, *Void*)
(**is** *?L* = *?R*) **including** *lifting-syntax*
proof −
have ∗: (λ*x y*. *map-spmf* (*map-prod id lprodr*) ((*A* $\oplus_O$ *B*) (*rprodl x*) *y*))
= (λ*x yl*. *map-spmf* (λ*p*. ((), *lprodr* (*snd p*))) (*A* (*rprodl x*) *yl*)) $\oplus_O$
(λ*x yr*. *map-spmf* (λ*p*. (*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* $\oplus_O$ (*rprodl* −−−> *id* −−−> *map-spmf* (*map-prod id lprodr*))
(*lift-state-oracle extend-state-oracle* (*attach-callee*
(λ*s 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))† $\oplus_O$ †*channel.send-oracle*))
$\oplus_O$
††(λ*s 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*=(), *symmetric*]
o-def bind-map-spmf exec-gpv-bind split-def option.case-distrib[**where** *h*=λ*gpv*. *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*$_C$ ▷ *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* ▷ - = *?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* (*auth-channel.auth-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* (*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-ℐ-attach*[*OF WT-resource-of-oracle*, **where** *ℐ'* = *?I* **and** *conv*=*?L1* **and** *conv'*=*?L1*]) *simp-all*
qed

lemma *all-together*:
defines *ℐ* ≡ *ℐ-uniform* (*Set.Collect* (*valid-mac-query η*)) (*insert None* (*Some* ‘ (*nlists UNIV η* × *nlists UNIV η*))) $\oplus_{\mathcal{I}}$
(*ℐ-uniform* (*vld η*) *UNIV* $\oplus_{\mathcal{I}}$ *ℐ-uniform UNIV* (*insert None* (*Some* ‘ *vld η*)))
assumes *η* > *0*
and *interaction-bounded-by'* (*λ-. True*) (*𝒜 η*) *q*
and *lossless*: *plossless-gpv ℐ* (*𝒜 η*)
and *WT*: *ℐ* ⊢*g* *𝒜 η* √
shows
|*spmf* (*connect* (*𝒜 η*) (*CNV sim* (*Inl None*) $|_=$ *1*$_C$ ▷ *channel.res auth-channel.auth-oracle*)) *True* −
spmf (*connect* (*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* 𝓘 (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* 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*) 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* 𝓘*-def*)[*1*]
apply(*fastforce simp*: *channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* 𝓘*-def*)[*1*]
apply(*fastforce simp*: *channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* 𝓘*-def*)[*1*]
apply(*fastforce simp*: *channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* 𝓘*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* 𝓘*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* 𝓘*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* 𝓘*-def*)[*1*]
apply(*fastforce simp*: *auth-channel.auth-oracle.simps channel.send-oracle.simps channel.recv-oracle.simps vld-def in-nlists-UNIV* 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* 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* 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* 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* 𝓘*-def*)
done
done
then have *WT1*: 𝓘 ⊢*res RES* (*callee-auth-channel sim* $\oplus_O$ ††*channel.send-oracle*

$\oplus_O$ ††*channel.recv-oracle*) (*Inl None*, (), *Void*) $\surd$
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-f*)
(λ(*s1*, *s2*, *s3*). *pred-cstate* (λx. *length* $x = \eta$) *s1* $\wedge$ *pred-option* (λ(*x*, *y*). *length* $x = \eta \wedge$ *length* $y = \eta$) *s2* $\wedge$ *ran s3* $\subseteq$ *nlists UNIV* η)
$\mathcal{I}$
apply *unfold-locales*
subgoal for *s x y s'*
using [[*simproc del*: *defined-all*]] **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* $x = \eta$) *s1* $\wedge$ *pred-option* (λ(*x*, *y*). *length* $x = \eta \wedge$ *length* $y = \eta$) *s2* $\wedge$ *ran s3* $\subseteq$ *nlists UNIV* η)
$\mathcal{I}$
apply *unfold-locales*
subgoal for *s x y s'*
using [[*simproc del*: *defined-all*]] **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*: $\mathcal{I}$ ⊢*res RES* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*) √
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* ⊆ *nlists UNIV* η ∧ *pred-cstate* (λ(*x*, *y*). *length x* = η ∧ *length y* = η) *s*) $\mathcal{I}$
apply(*unfold-locales*)
subgoal for *s x y s'*
using [[*simproc del*: *defined-all*]] **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*: $\mathcal{I}$ ⊢*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*:
 $X \neq \{\} \Longrightarrow$ $\mathcal{I}$*-uniform* (*vld* η) *UNIV*, $\mathcal{I}$*-uniform* (*vld* η) $X \oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* ($X \times$ *vld* η) *UNIV* $\vdash_C$ *MAC.enm* η $\surd$
 unfolding *MAC.enm-def*
 by(*rule WT-converter-of-callee*) (*auto simp add*: *mac-def*)

lemma *WT-dem*: $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *vld* η)), $\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ (*nlists UNIV* η $\times$ *nlists UNIV* η))) $\vdash_C$ *MAC.dem* η $\surd$
 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 $\mathcal{I}$*-real* $\equiv \lambda\eta$. $\mathcal{I}$*-uniform* $\{x.$ *valid-mac-query* η $x\}$ (*insert None* (*Some* ‘ (*nlists UNIV* η $\times$ *nlists UNIV* η)))
  and $\mathcal{I}$*-ideal* $\equiv \lambda\eta$. $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η))
  and $\mathcal{I}$*-common* $\equiv \lambda\eta$. $\mathcal{I}$*-uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*insert None* (*Some* ‘ *vld* η))
 shows
  constructive-security MAC.res (λ-. *A-CHAN.res*) (λ-. *CNV sim* (*Inl None*))
  $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* (λ-. *enat q*) *True* (λ-. *insec-auth-wiring*)
proof
 show *WT-res* [*WT-intro*]: $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res MAC.res* η $\surd$ **for** η
 proof −
  let *?I* = *pred-cstate* ($\lambda(x, y)$. *length* $x = \eta \wedge$ *length* $y = \eta$)

have ∗: *callee-invariant-on* (*MAC.RO.rnd-oracle* $\eta \oplus_O$ *MAC.RO.rnd-oracle* η) ($\lambda m.$ *ran m* $\subseteq$ *vld* η) ($\mathcal{I}$*-uniform* (*vld* η) (*vld* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-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*]: $\mathcal{I}$*-uniform* (*vld* η) (*vld* η) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full* $\vdash$*res MAC.RO.res* η $\surd$ **for** η
unfolding *MAC.RO.res-def* **by**(*rule callee-invariant-on.WT-resource-of-oracle*[*OF* ∗]) *simp*
have [*simp*]: $\mathcal{I}$*-real* η $\vdash$*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* η × *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* = η)
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*: *I-uniform UNIV* (*insert None* (*Some ' vld η*)) ⊢*c A-CHAN.recv-oracle s* √
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*) (*I-ideal η* $\oplus_{\mathcal{I}}$ *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*: *I-common-def vld-def in-nlists-UNIV*)
subgoal for *s* **using** *WT-auth WT-recv* **by**(*auto simp add*: *I-common-def I-ideal-def intro*: *WT-calleeI*)
done
show *WT-auth*: *I-ideal η* $\oplus_{\mathcal{I}}$ *I-common η* ⊢*res A-CHAN.res* √ **for** *η*
unfolding *A-CHAN.res-def* **by**(*rule callee-invariant-on.WT-resource-of-oracle*[*OF* ∗]) *simp*

let *?I-sim* = *λη* (*s* :: (*bool list* × *bool list*) *option* + *unit*). ∀ *x y*. *s* = *Inl* (*Some* (*x*, *y*)) ⟶ *length x* = *η* ∧ *length y* = *η*

have *I-real η*, *I-ideal η* $\vdash_C$ *CNV sim s* √ **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*: *I-real-def I-ideal-def vld-def in-nlists-UNIV*)
done
then show [*WT-intro*]: *I-real η*, *I-ideal η* $\vdash_C$ *CNV sim* (*Inl None*) √ **for** *η* **by** *simp*

{ **fix** *A* :: *security* ⇒ ((*bool list* × *bool list*) *insec-query* + *bool list* + *unit*, (*bool list* × *bool list*) *option* + *unit* + *bool list option*) *distinguisher*
assume *WT*: *I-real η* $\oplus_{\mathcal{I}}$ *I-common η* ⊢*g A η* √ **for** *η*
assume *bounded*[*simplified*]: *interaction-bounded-by'* (λ-. *True*) (*A η*) *q* **for** *η*
assume *lossless*[*simplified*]: *True* ⟹ *plossless-gpv* (*I-real η* $\oplus_{\mathcal{I}}$ *I-common η*) (*A η*) **for** *η*
show *negligible* (*λη*. *advantage* (*A η*) (*CNV sim* (*Inl None*) $|_=$ *1* $_C$ ⊳ *A-CHAN.res*) (*MAC.res η*))
proof −
have *f1*: *negligible* (*λη*. *q* ∗ (*1* / *real* (*2* ^ *η*))) (**is** *negligible ?ov*)
by(*rule negligible-poly-times*[**where** *n=0*]) (*simp add*: *negligible-inverse-powerI*)+

have *f2*: (*λη*. *spmf* (*connect* (*A η*) (*CNV sim* (*Inl None*) $|_=$ *1* $_C$ ⊳ *A-CHAN.res*)) *True* −
spmf (*connect* (*A η*) (*MAC.res η*)) *True*) ∈ *O*(*?ov*) (**is** *?L* ∈ -)
by (*auto simp add*: *bigo-def intro*!: *all-together*[*simplified*] *bounded eventually-at-top-linorderI*[**where** *c=1*]
exI[**where** *x=1*] *lossless*[*unfolded I-real-def I-common-def*] *WT*[*unfolded I-real-def 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 ∃ *cnv*. ∀ *𝒟*. (∀ *η*. *ℐ-ideal η* $\oplus_{\mathcal{I}}$ *ℐ-common η* ⊢*g* *𝒟 η* √) ⟶
(∀ *η*. *interaction-bounded-by′* (λ-. *True*) (*𝒟 η*) *q*) ⟶
(∀ *η*. *True* ⟶ *plossless-gpv* (*ℐ-ideal η* $\oplus_{\mathcal{I}}$ *ℐ-common η*) (*𝒟 η*)) ⟶
(∀ *η*. *wiring* (*ℐ-ideal η*) (*ℐ-real η*) (*cnv η*) (*insec-query-of*, *map-option snd*)) ∧
negligible (λ*η*. *advantage* (*𝒟 η*) *A-CHAN.res* (*cnv η* $|_=$ 1_C ⊳ *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* $\oplus_{\mathcal{I}}$ *ℐ-common η* ⊢*g* *𝒟 η* √

let *?A* = λ*η*. *UNIV* <+> *nlists UNIV η* <+> *UNIV*
have *WT1*: *ℐ-ideal η*, *map-ℐ insec-query-of* (*map-option snd*) (*ℐ-real η*) $\vdash_C$ 1_C √ **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* $\oplus_{\mathcal{I}}$ *ℐ-common η* ⊢*res map-converter id id insec-query-of* (*map-option snd*) 1_C $|_=$ 1_C ⊳ *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*) $\vdash_C$ 1_C √ **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* $\vdash_C$ *?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* $\oplus_O$
((*MAC.RO.rnd-oracle η*)† $\oplus_O$ †*A-CHAN.send-oracle*) $\oplus_O$ (*MAC.RO.rnd-oracle η*)† $\oplus_O$ †*A-CHAN.recv-oracle*
for *η*

have *push*: *map-resource* (*map-sum f id*) (*map-sum g id*) ((1_C $|_=$ *conv*) ⊳ *res*) =
(1_C $|_=$ *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*) ((1_C $|_=$ *conv*) ⊳ *res*) = *map-converter f g id id* 1_C $|_=$ *conv* ⊳ *res*
by(*simp add*: *attach-map-converter parallel-converter2-map1-out sum.map-id0*)
also have ... = (1_C $|_=$ *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*) $\oplus_O$
†(*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)† $\oplus_O$ †*A-CHAN.send-oracle*)) $\oplus_O$
††(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*=*λx. exec-gpv - x -*]
option.case-distrib[**where** *h*=*λx. exec-gpv - x -*] *prod.case-distrib*[**where** *h*=*λx. 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* = *η* ∧ *m* = [*z* ↦ *y*] ∧ *x* = *z*
| (*Collect x*, ((-, *True*, -), *m*, *Collect* (*y*, *z*))) ⇒ *length x* = *η* ∧ *length y* = *η* ∧ *length z* = *η* ∧ *m* = [*z* ↦ *y*] ∧ *x* = *z*
| (*Fail*, ((-, *True*, -), *m*, *Fail*)) ⇒ *True*
| - ⇒ *False*) **for** *η l r*

note [*simp*] = *spmf-rel-map bind-map-spmf exec-gpv-bind*
have *connect* (*𝒟 η*) (*?cnv* $|_=$ *1*$_C$ ▷ *MAC.res η*) = *connect* (*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 ℐ-common-def vld-def in-nlists-UNIV S-def*)+
subgoal by(*simp add*: *S-def*)
done
then show *adv*: *negligible* (*λη. advantage* (*𝒟 η*) *A-CHAN.res* (*?cnv* $|_=$ *1*$_C$ ▷ *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* $\equiv$ λ-. *$\mathcal{I}$-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 $|_=$ (*CIPHER.enc* η $|_=$ *CIPHER.dec* η) $\odot$ *parallel-wiring* $\triangleright$
 parallel-resource1-wiring $\triangleright$
 CIPHER.KEY.res η $\|$
 (1_C $|_=$ *MAC.enm* η $|_=$ *MAC.dem* η $\triangleright$
 1_C $|_=$ *parallel-wiring* $\triangleright$
 parallel-resource1-wiring $\triangleright$ *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 $\times$ *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* (λx. *length x* = η) *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* η $\oplus_O$ *CIPHER.KEY.key-oracle* η) (*pred-option* (λx. *length x* = η))
(*$\mathcal{I}$-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-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.