

# Constructive Cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar

April 24, 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

<b>1</b>	<b>Resources</b>	<b>3</b>
1.1	Type definition . . . . .	3
1.2	Functor . . . . .	3
1.3	Relator . . . . .	4
1.4	Losslessness . . . . .	10
1.5	Operations . . . . .	10
1.6	Well-typing . . . . .	12
<b>2</b>	<b>Converters</b>	<b>14</b>
2.1	Type definition . . . . .	14
2.2	Functor . . . . .	14
2.3	Set functions with interfaces . . . . .	15
2.4	Relator . . . . .	16
2.5	Well-typing . . . . .	25
2.6	Losslessness . . . . .	28
2.7	Operations . . . . .	29
2.8	Attaching converters to resources . . . . .	36
2.9	Composing converters . . . . .	39
2.10	Interaction bound . . . . .	44

<b>3</b>	<b>Equivalence of converters restricted by interfaces</b>	<b>49</b>
<b>4</b>	<b>Trace equivalence for resources</b>	<b>60</b>
<b>5</b>	<b>Distinguisher</b>	<b>66</b>
<b>6</b>	<b>Wiring</b>	<b>71</b>
6.1	Notation . . . . .	71
6.2	Wiring primitives . . . . .	72
6.3	Characterization of wirings . . . . .	78
<b>7</b>	<b>Security</b>	<b>83</b>
7.1	Composition theorems . . . . .	86
<b>8</b>	<b>Examples</b>	<b>102</b>
8.1	Random oracle resource . . . . .	102
8.2	Key resource . . . . .	103
8.3	Channel resource . . . . .	103
8.3.1	Generic channel . . . . .	103
8.3.2	Insecure channel . . . . .	104
8.3.3	Authenticated channel . . . . .	104
8.3.4	Secure channel . . . . .	105
8.4	Cipher converter . . . . .	105
8.5	Message authentication converter . . . . .	106
<b>9</b>	<b>Security of one-time-pad encryption</b>	<b>108</b>
<b>10</b>	<b>Security of message authentication</b>	<b>116</b>
<b>11</b>	<b>Secure composition: Encrypt then MAC</b>	<b>148</b>

```

theory Resource imports
  CryptHOL.CryptHOL
begin

```

## 1 Resources

### 1.1 Type definition

```

codatatype ('a, 'b) resource
  = Resource (run-resource: 'a  $\Rightarrow$  ('b  $\times$  ('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-iff)

```

### 1.2 Functor

```

context
  fixes a :: 'a  $\Rightarrow$  'a'
  and b :: 'b  $\Rightarrow$  'b'
begin

```

```

primcorec map-resource :: ('a', 'b) resource  $\Rightarrow$  ('a, 'b) resource where
  run-resource (map-resource res) = map-spmf (map-prod b map-resource)  $\circ$  (run-resource
  res)  $\circ$  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)  $\circ$ 
  f  $\circ$  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-refl)

```

```

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]:

```

$map-resource\ a\ b\ (map-resource\ a'\ b'\ res) = map-resource\ (a' \circ a)\ (b \circ b')\ res$   
**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* :: ( $'a \Rightarrow 'b \Rightarrow bool$ )  $\Rightarrow$  ( $'c \Rightarrow 'd \Rightarrow bool$ )  $\Rightarrow$  ( $'a, 'c$ )  
*resource*  $\Rightarrow$  ( $'b, 'd$ ) *resource*  $\Rightarrow$  *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*)*

$\Rightarrow$  *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**  $\bigwedge res1\ res2. X\ res1\ res2 \Rightarrow$

*rel-fun A (rel-*spmf* (rel-*prod B* ( $\lambda res1\ res2. X\ res1\ res2 \vee rel-resource\ A\ B\ res1\ res2$ )))*

(*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)  $\longleftrightarrow$  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  $\Rightarrow$  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*  $A B$ )<sup>-1-1</sup>  
**proof**(*intro ext iffI; simp*)  
**show** *rel-resource*  $A B res1 res2$  **if** *rel-resource*  $A^{-1-1} B^{-1-1} res2 res1$   
**for**  $A :: 'a1 \Rightarrow 'a2 \Rightarrow bool$  **and**  $B :: 'c1 \Rightarrow 'c2 \Rightarrow bool$  **and**  $res1 res2$   
**using** *that by*(*coinduct*)  
(*drule rel-resourceD, rewrite in  $\sqsupset$  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*  $A$  ( $\lambda x. B (f x)$ )  $res1$   
 $res2$   
(*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*  $A$  ( $\lambda x y. B x (f y)$ )

```

res1 res2
  using rel-resource-map-resource'1 [of conversep A conversep B f res2 res1]
  by (rewrite in  $\sqsupset = - \text{conversep-iff}$  [symmetric]
      , rewrite in  $- = \sqsupset \text{conversep-iff}$  [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'  $\leq$  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:
   $\llbracket \text{left-total } A; \text{left-unique } B \rrbracket \implies \text{left-unique } (\text{rel-resource } A \ B)$ 
  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:
   $\llbracket \text{right-total } A; \text{right-unique } B \rrbracket \implies \text{right-unique } (\text{rel-resource } A \ B)$ 
  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]:
   $\llbracket \text{bi-total } A; \text{bi-unique } B \rrbracket \implies \text{bi-unique } (\text{rel-resource } A \ B)$ 
  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: rel-
comppE elim!: relcompp-witness)
show rel-resource A' B' ?res' res'' using rel-witness-resource2[OF * A A'] un-
folding resource-rel-map'
  by(rule rel-resource-mono[THEN predicate2D, rotated -1]; clarify del: rel-
comppE 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*]:  
 $((S \text{====>} A \text{====>} \text{rel-spmf } (\text{rel-prod } B (\text{rel-sum } (\text{rel-resource } A B) S))) \text{====>} S \text{====>} \text{rel-resource } A B)$   
*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*]:  
 $((A' \text{====>} A) \text{====>} (B \text{====>} B') \text{====>} \text{rel-resource } A B \text{====>} \text{rel-resource } A' B')$  *map-resource map-resource*  
**unfolding** *map-resource-def* **by**(*transfer-prover*)

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

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

**end**

**lemma** *rel-resource-Grp*:  
 $\text{rel-resource } (\text{conversep } (\text{BNF-Def.Grp UNIV } f)) (\text{BNF-Def.Grp UNIV } g) = \text{BNF-Def.Grp UNIV } (\text{map-resource } f g)$   
**proof**((*rule ext iffI*)+, *goal-cases*)  
**case** (1 *res res'*)  
**have** \*:  $\text{rel-resource } (\lambda a b. b = f a)^{-1-1} (\lambda a b. b = g a) \text{res } \text{res}' \implies \text{res}' = \text{map-resource } f g \text{res}$   
**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)  $\mathcal{I} \Rightarrow$  ('a, 'b) *resource*  $\Rightarrow$  *bool*  
**for**  $\mathcal{I}$  **where**

*lossless-resourceI: lossless-resource*  $\mathcal{I}$  *res* **if**  
 $\bigwedge a. a \in \text{outs-}\mathcal{I} \ \mathcal{I} \Longrightarrow \text{lossless-spmf} \ (\text{run-resource} \ \text{res} \ a)$   
 $\bigwedge a \ b \ \text{res}'. \llbracket a \in \text{outs-}\mathcal{I} \ \mathcal{I}; (b, \text{res}') \in \text{set-spmf} \ (\text{run-resource} \ \text{res} \ a) \rrbracket \Longrightarrow$   
*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**  $X \ \text{res}$   
**and**  $\bigwedge \text{res} \ a. \llbracket X \ \text{res}; a \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \Longrightarrow \text{lossless-spmf} \ (\text{run-resource} \ \text{res} \ a) \wedge$   
 $(\forall (b, \text{res}') \in \text{set-spmf} \ (\text{run-resource} \ \text{res} \ a). X \ \text{res}' \vee \text{lossless-resource} \ \mathcal{I} \ \text{res}')$   
**shows** *lossless-resource*  $\mathcal{I}$  *res*  
**using** *assms(1)* **by**(*rule lossless-resource.coinduct*)(*auto dest: assms(2)*)

**lemma** *lossless-resourceD*:

$\llbracket \text{lossless-resource} \ \mathcal{I} \ \text{res}; a \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$   
 $\Longrightarrow \text{lossless-spmf} \ (\text{run-resource} \ \text{res} \ a) \wedge (\forall (x, \text{res}') \in \text{set-spmf} \ (\text{run-resource} \ \text{res} \ a). \text{lossless-resource} \ \mathcal{I} \ \text{res}')$   
**by**(*auto elim: lossless-resource.cases*)

**lemma** *lossless-resource-mono*:

**assumes** *lossless-resource*  $\mathcal{I}' \ \text{res}$   
**and**  $le: \text{outs-}\mathcal{I} \ \mathcal{I} \subseteq \text{outs-}\mathcal{I} \ \mathcal{I}'$   
**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'*:

$\llbracket \text{lossless-resource} \ \mathcal{I}' \ \text{res}; \mathcal{I} \leq \mathcal{I}' \rrbracket \Longrightarrow \text{lossless-resource} \ \mathcal{I} \ \text{res}$   
**by**(*erule lossless-resource-mono*)(*simp add: le- $\mathcal{I}$ -def*)

## 1.5 Operations

**context** *fixes oracle* :: 's  $\Rightarrow$  'a  $\Rightarrow$  ('b  $\times$  's) *spmf* **begin**

**primcorec** *resource-of-oracle* :: 's  $\Rightarrow$  ('a, 'b) *resource* **where**

*run-resource* (*resource-of-oracle* *s*) = ( $\lambda a. \text{map-spmf} \ (\text{map-prod} \ \text{id} \ \text{resource-of-oracle})$   
(*oracle* *s* *a*))

**end**

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

**lemma** *map-resource-resource-of-oracle*:  
 $\text{map-resource } f \ g \ (\text{resource-of-oracle } \text{oracle } s) = \text{resource-of-oracle } (\text{map-fun } \text{id}$   
 $(\text{map-fun } f \ (\text{map-spmf } (\text{map-prod } g \ \text{id}))) \ \text{oracle}) \ s$   
**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*  
 $(\text{subst } (\text{asm}) \ (1 \ 2) \ \text{eq-alt}[\text{symmetric}]$   
 $, \ \text{subst } (\text{asm}) \ (1 \ 2) \ \text{conversep-eq}[\text{symmetric}]$   
 $, \ \text{subst } (\text{asm}) \ (1 \ 2) \ \text{eq-alt}$   
 $, \ \text{unfold } \text{rel-fun-Grp}, \ \text{simp } \text{add: } \text{rel-fun-Grp } \text{rel-fun-def } \text{Grp-iff})$

**lemma** (**in** *callee-invariant-on*) *lossless-resource-of-oracle*:  
**assumes**  $*$ :  $\bigwedge s \ x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; \ I \ s \rrbracket \implies \text{lossless-spmf } (\text{callee } s \ x)$   
**and**  $I \ s$   
**shows** *lossless-resource*  $\mathcal{I}$  (*resource-of-oracle callee s*)  
**using**  $\langle I \ s \rangle$  **by**(*coinduction arbitrary: s*)(*auto intro: \* dest: callee-invariant*)

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

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

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

**end**

**lemma** *exec-gpv-resource-of-oracle*:  
 $\text{exec-gpv } \text{run-resource } \text{gpv} \ (\text{resource-of-oracle } \text{oracle } s) = \text{map-spmf } (\text{map-prod } \text{id}$   
 $(\text{resource-of-oracle } \text{oracle})) \ (\text{exec-gpv } \text{oracle } \text{gpv } s)$   
**by**( $\text{subst } \text{spmfm.map-id}[\text{symmetric}], \ \text{fold } \text{pmf.rel-eq}$   
 $(\text{rule } \text{pmfm.map-transfer}[\text{THEN } \text{rel-funD}, \ \text{THEN } \text{rel-funD}, \ \text{rotated}]$   
 $, \ \text{rule } \text{exec-gpv-parametric}[\text{where } S = \lambda \text{res } s. \ \text{res} = \text{resource-of-oracle } \text{oracle } s$   
**and**  $\text{CALL} = (=) \ \text{and } A = (=), \ \text{THEN } \text{rel-funD}, \ \text{THEN } \text{rel-funD}, \ \text{THEN } \text{rel-funD}]$

, *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  $\Rightarrow$  ('c, 'd) resource  $\Rightarrow$  ('a + 'c, 'b + 'd) resource **where**  
*run-resource (parallel-resource res1 res2) =*  
*( $\lambda$ ac. case ac of Inl a  $\Rightarrow$  map-spmf (map-prod Inl ( $\lambda$ res1'. parallel-resource res1' res2)) (run-resource res1 a)*  
*| Inr c  $\Rightarrow$  map-spmf (map-prod Inr ( $\lambda$ res2'. parallel-resource res1 res2')) (run-resource res2 c))*

**lemma** *parallel-resource-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**  
*(rel-resource A B  $\implies$  rel-resource C D  $\implies$  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 I res1 lossless-resource I' res2*  
**shows** *lossless-resource (I  $\oplus_{\mathcal{I}}$  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)  $\mathcal{I} \Rightarrow$  ('a, 'b) resource  $\Rightarrow$  bool (*«- / $\vdash$ res -  $\surd$ »*  
[100, 0] 99)  
**for**  $\mathcal{I}$  **where**  
*WT-resourceI:  $\mathcal{I} \vdash$ res res  $\surd$*   
**if**  $\bigwedge q r res'. \llbracket q \in \text{outs-}\mathcal{I} \ \mathcal{I}; (r, res') \in \text{set-spmf (run-resource res } q) \rrbracket \implies r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q \wedge \mathcal{I} \vdash$ res  $res' \surd$

**lemma** *WT-resource-coinduct* [*consumes 1, case-names WT-resource, case-conclusion WT-resource response WT-resource, coinduct pred: WT-resource*]:

**assumes**  $X \text{ res}$   
**and**  $\bigwedge \text{res } q \text{ } r \text{ } \text{res}' . \llbracket X \text{ res}; q \in \text{outs-}\mathcal{I} \ \mathcal{I}; (r, \text{res}') \in \text{set-spmf } (\text{run-resource } \text{res } q) \rrbracket$   
 $\implies r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q \wedge (X \text{ res}' \vee \mathcal{I} \vdash_{\text{res}} \text{res}' \ \checkmark)$   
**shows**  $\mathcal{I} \vdash_{\text{res}} \text{res} \ \checkmark$   
**using** *assms(1)* **by**(*rule WT-resource.coinduct*)(*blast dest: assms(2)*)

**lemma** *WT-resourceD*:

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

**lemma** *WT-resource-of-oracle* [*simp*]:

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

**lemma** *WT-resource-bot* [*simp*]:  $\text{bot} \vdash_{\text{res}} \text{res} \ \checkmark$

**by**(*rule WT-resource.intros*)*auto*

**lemma** *WT-resource-full*:  $\mathcal{I}\text{-full} \vdash_{\text{res}} \text{res} \ \checkmark$

**by**(*coinduction arbitrary: res*)(*auto*)

**lemma** (**in** *callee-invariant-on*) *WT-resource-of-oracle*:

$I \ s \implies \mathcal{I} \vdash_{\text{res}} \text{resource-of-oracle } \text{callee } s \ \checkmark$   
**by**(*coinduction arbitrary: s*)(*auto dest: callee-invariant'*)

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

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

**lemma** *WT-parallel-resource* [*WT-intro*]:

**assumes**  $\mathcal{I}1 \vdash_{\text{res}} \text{res}1 \ \checkmark$   
**and**  $\mathcal{I}2 \vdash_{\text{res}} \text{res}2 \ \checkmark$   
**shows**  $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_{\text{res}} \text{parallel-resource } \text{res}1 \ \text{res}2 \ \checkmark$   
**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* ( $\lambda \text{res} . \mathcal{I} \vdash_{\text{res}} \text{res} \ \checkmark$ )  $\mathcal{I}$

**by**(*unfold-locales*)(*auto dest: WT-resourceD intro: WT-calleeI*)

**lemma** *callee-invariant-run-lossless-resource*:

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

```

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

end
theory Converter imports
  Resource
begin

```

## 2 Converters

### 2.1 Type definition

```

codatatype ('a, results'-converter: 'b, outs'-converter: 'out, 'in) converter
  = Converter (run-converter: 'a  $\Rightarrow$  ('b  $\times$  ('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  $\Rightarrow$  'a'
  and b :: 'b  $\Rightarrow$  'b'
  and out :: 'out  $\Rightarrow$  'out'
  and inn :: 'in  $\Rightarrow$  'in'
begin

primcorec map-converter :: ('a', 'b, 'out, 'in') converter  $\Rightarrow$  ('a, 'b', 'out', 'in)
converter where
  run-converter (map-converter conv) =
    map-gpv (map-prod b map-converter) out  $\circ$  map-gpv' id id inn  $\circ$  run-converter
conv  $\circ$  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' o a)*  
*(b o b') (f o f') (g' o 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 I :: ('a, 'b) I and I' :: ('out, 'in) 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 I' (run-converter conv a) a ∈*  
*outs-I I*

| *Cont: outsp-converter out conv*

**if** *(b, conv') ∈ results-gpv I' (run-converter conv a) outsp-converter out conv' a ∈*  
*outs-I 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 I' (run-converter conv a) a ∈ outs-I I*

| *Cont: resultsp-converter b conv*

**if** *(b', conv') ∈ results-gpv I' (run-converter conv a) resultsp-converter b conv' a ∈*  
*outs-I 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*  
 $\mathcal{I} \mathcal{I}' x = (\lambda conv. x \in \text{outs-converter } \mathcal{I} \mathcal{I}' conv)$   
**by**(*simp add: outs-converter-def*)

**context begin**

**local-setup**  $\langle \text{Local-Theory.map-background-naming } (\text{Name-Space.mandatory-path } \text{outs-converter}) \rangle$

**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*  $\mathcal{I} \mathcal{I}' x = (\lambda conv. x \in \text{results-converter } \mathcal{I} \mathcal{I}' conv)$   
**by**(*simp add: results-converter-def*)

**context begin**

**local-setup**  $\langle \text{Local-Theory.map-background-naming } (\text{Name-Space.mandatory-path } \text{results-converter}) \rangle$

**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*

$:: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('c \Rightarrow 'd \Rightarrow \text{bool}) \Rightarrow ('out \Rightarrow 'out' \Rightarrow \text{bool}) \Rightarrow ('in \Rightarrow 'in'$   
 $\Rightarrow \text{bool})$



$\Rightarrow ('a, 'c, 'out, 'in) \text{ converter} \Rightarrow ('b, 'd, 'out', 'in') \text{ converter} \Rightarrow \text{bool}$   
**for**  $A B C R$  **where**  
*rel-converterI*:  
 $\text{rel-fun } A (\text{rel-gpv}'' (\text{rel-prod } B (\text{rel-converter } A B C R)) C R) (\text{run-converter conv1}) (\text{run-converter conv2})$   
 $\Rightarrow \text{rel-converter } A B C R \text{ conv1 conv2}$

**lemma** *rel-converter-coinduct* [*consumes 1, case-names rel-converter, coinduct pred: rel-converter*]:

**assumes**  $X \text{ conv1 conv2}$   
**and**  $\bigwedge \text{conv1 conv2}. X \text{ conv1 conv2} \Rightarrow$   
 $\text{rel-fun } A (\text{rel-gpv}'' (\text{rel-prod } B (\lambda \text{conv1 conv2}. X \text{ conv1 conv2} \vee \text{rel-converter } A B C R \text{ conv1 conv2})) C R)$   
 $(\text{run-converter conv1}) (\text{run-converter conv2})$   
**shows**  $\text{rel-converter } A B C R \text{ conv1 conv2}$   
**using** *assms(1)* **by**(*rule rel-converter.coinduct*)(*simp add: assms(2)*)

**lemma** *rel-converter-simps* [*simp, code*]:

$\text{rel-converter } A B C R (\text{Converter } f) (\text{Converter } g) \longleftrightarrow$   
 $\text{rel-fun } A (\text{rel-gpv}'' (\text{rel-prod } B (\text{rel-converter } A B C R)) C R) f g$   
**by**(*subst rel-converter.simps*) *simp*

**lemma** *rel-converterD*:

$\text{rel-converter } A B C R \text{ conv1 conv2}$   
 $\Rightarrow \text{rel-fun } A (\text{rel-gpv}'' (\text{rel-prod } B (\text{rel-converter } A B C R)) C R) (\text{run-converter conv1}) (\text{run-converter conv2})$   
**by**(*blast elim: rel-converter.cases*)

**lemma** *rel-converter-eq14*:  $\text{rel-converter } (=) B C (=) = \text{rel-converter}' B C$  (**is**  $?lhs = ?rhs$ )

**proof**(*intro ext iffI*)

**show**  $?rhs \text{ conv1 conv2}$  **if**  $?lhs \text{ conv1 conv2}$  **for**  $\text{conv1 conv2}$  **using** *that*  
**by**(*coinduction arbitrary: conv1 conv2*)(*auto elim: rel-converter.cases simp add: rel-gpv-conv-rel-gpv''*)

**show**  $?lhs \text{ conv1 conv2}$  **if**  $?rhs \text{ conv1 conv2}$  **for**  $\text{conv1 conv2}$  **using** *that*  
**by**(*coinduction arbitrary: conv1 conv2*)  
 $(\text{auto } 4 \ 4 \ \text{elim: converter.rel-cases intro: gpv.rel-mono-strong simp add: rel-fun-def rel-gpv-conv-rel-gpv}''[\text{symmetric}])$   
**qed**

**lemma** *rel-converter-eq* [*relator-eq*]:  $\text{rel-converter } (=) (=) (=) (=) (=) (=)$

**by**(*simp add: rel-converter-eq14 converter.rel-eq*)

**lemma** *rel-converter-mono* [*relator-mono*]:

**assumes**  $A' \leq A B \leq B' C \leq C' R' \leq R$

**shows**  $\text{rel-converter } A B C R \leq \text{rel-converter } A' B' C' R'$

**proof**

**show**  $\text{rel-converter } A' B' C' R' \text{ conv1 conv2}$  **if**  $\text{rel-converter } A B C R \text{ conv1 conv2}$  **for**  $\text{conv1 conv2}$  **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<sup>-1-1</sup> B<sup>-1-1</sup> C<sup>-1-1</sup> R<sup>-1-1</sup> 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<sup>-1-1</sup> B<sup>-1-1</sup> C<sup>-1-1</sup> R<sup>-1-1</sup>]*

**show** *rel-converter A<sup>-1-1</sup> B<sup>-1-1</sup> C<sup>-1-1</sup> R<sup>-1-1</sup> conv2 conv1 if rel-converter A B C R conv1 conv2 for conv1 conv2*

**using** *that by simp*

**qed**

**lemma** *rel-converter-map-converter'1*:

*rel-converter A B C R (map-converter' f g conv1) conv2 = rel-converter A (λx. B (f x)) (λx. C (g x)) R conv1 conv2*

(*is ?lhs = ?rhs*)

**proof**

**show** *?rhs if ?lhs using that*

**by**(*coinduction arbitrary: conv1 conv2*)

(*drule rel-converterD, auto intro: prod.rel-mono elim!: rel-fun-mono rel-gpv''-mono[THEN predicate2D, rotated -1]*)

*simp add: map-gpv'-id rel-gpv''-map-gpv map-converter.sel map-converter-id14[symmetric] rel-fun-comp spmf-rel-map prod.rel-map[abs-def]*)

**show** *?lhs if ?rhs using that*

**by**(*coinduction arbitrary: conv1 conv2*)

(*drule rel-converterD, auto intro: prod.rel-mono elim!: rel-fun-mono rel-gpv''-mono[THEN predicate2D, rotated -1]*)

*simp add: map-gpv'-id rel-gpv''-map-gpv map-converter.sel map-converter-id14[symmetric] rel-fun-comp spmf-rel-map prod.rel-map[abs-def]*)

**qed**

**lemma** *rel-converter-map-converter'2*:

*rel-converter A B C R conv1 (map-converter' f g conv2) = rel-converter A (λx y. B x (f y)) (λx y. C x (g y)) R conv1 conv2*

```

using rel-converter-map-converter'1[of conversep A conversep B conversep C
conversep R f g conv2 conv1]
apply(rewrite in  $\sqsupset = -$  conversep-iff[symmetric])
apply(rewrite in  $- = \sqsupset$  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]:

$rel-converter A B C R OO rel-converter A' B' C' R' \leq rel-converter (A OO A')$   
 $(B OO B') (C OO C') (R OO R')$

**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:

$\llbracket left-total A; left-unique B; left-unique C; left-total R \rrbracket \implies left-unique (rel-converter A B C R)$

**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:

$\llbracket right-total A; right-unique B; right-unique C; right-total R \rrbracket \implies right-unique (rel-converter A B C R)$

**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]:

$\llbracket bi-total A; bi-unique B; bi-unique C; bi-total R \rrbracket \implies bi-unique (rel-converter A B C R)$

**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 (λb (b', c). b = b' ∧ B b' c) (λc (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  $\text{rel-converter } (A \text{ OO } A') (B \text{ OO } B') (C \text{ OO } C') (R \text{ OO } R') \leq \text{rel-converter } A B C R \text{ OO } \text{rel-converter } A' B' C' R'$   
**proof**(rule predicate2I relcomppI)+  
 fix *conv conv''*  
 assume \*:  $\text{rel-converter } (A \text{ OO } A') (B \text{ OO } B') (C \text{ OO } C') (R \text{ OO } R') \text{ conv } \text{conv''}$   
 let  $?conv' = \text{map-converter}' (\text{relcompp-witness } B B') (\text{relcompp-witness } C C') (\text{rel-witness-converter } A A' (B \text{ OO } B') (C \text{ OO } C') R R' (\text{conv}, \text{conv''}))$   
 show  $\text{rel-converter } A B C R \text{ conv } ?conv' \text{ using } \text{rel-witness-converter1}[OF * A A' R R']$  **unfolding** *converter-rel-map'*  
 by(rule rel-converter-mono[THEN predicate2D, rotated -1]; clarify del: rel-comppE elim!: relcompp-witness)  
 show  $\text{rel-converter } A' B' C' R' ?conv' \text{conv''}$  **using** *rel-witness-converter2*[OF \* A A' R R'] **unfolding** *converter-rel-map'*  
 by(rule rel-converter-mono[THEN predicate2D, rotated -1]; clarify del: rel-comppE elim!: relcompp-witness)  
**qed**

**lemma** *left-total-rel-converter*:  
 $\llbracket \text{left-unique } A; \text{right-total } A; \text{left-total } B; \text{left-total } C; \text{left-unique } R; \text{right-total } R \rrbracket$   
 $\implies \text{left-total } (\text{rel-converter } A B C R)$   
**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*:  
 $\llbracket \text{right-unique } A; \text{left-total } A; \text{right-total } B; \text{right-total } C; \text{right-unique } R; \text{left-total } R \rrbracket$   
 $\implies \text{right-total } (\text{rel-converter } A B C R)$   
**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  $\subseteq$  B
let ?abr2= $\lambda$ conv. outs-converter (I-uniform (range f) UNIV) (I-uniform UNIV
(range k)) conv  $\subseteq$  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  $\implies$  ?abr2 conv  $\implies$  z  $\in$  run-converter conv ' range f  $\implies$ 
out  $\in$  outs-gpv (I-uniform UNIV (range k)) z  $\implies$  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  $\implies$  rel-gpv'' (rel-prod B (rel-converter A B C R)) C R)  $\implies$  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  $\implies$  A  $\implies$  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*]:  
 $((S \text{====>} A \text{====>} \text{rel-gpv''} (\text{rel-prod } B (\text{rel-sum} (\text{rel-converter } A B C R) S))$   
 $C R) \text{====>} S \text{====>} \text{rel-converter } A B C R)$   
*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*]:  
 $((A' \text{====>} A) \text{====>} (B \text{====>} B') \text{====>} (C \text{====>} C') \text{====>} (R' \text{====>} R))$   
 $\text{====>} \text{rel-converter } A B C R \text{====>} \text{rel-converter } A' B' C' R')$   
*map-converter map-converter*  
**unfolding** *map-converter-def* **by**(*transfer-prover*)

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

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

**end**

## 2.5 Well-typing

**coinductive** *WT-converter* :: ('a, 'b)  $\mathcal{I} \Rightarrow ('out, 'in) \mathcal{I} \Rightarrow ('a, 'b, 'out, 'in) \text{converter} \Rightarrow \text{bool}$   
( $\langle -, / - \vdash_C / - \sqrt{\ } \rangle$  [100, 0, 0] 99)  
**for**  $\mathcal{I} \mathcal{I}'$  **where**  
*WT-converterI*:  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sqrt{\ }$  **if**  
 $\bigwedge q. q \in \text{outs-}\mathcal{I} \mathcal{I} \Longrightarrow \mathcal{I}' \vdash_g \text{run-converter conv } q \sqrt{\ }$   
 $\bigwedge q r \text{conv}'. \llbracket q \in \text{outs-}\mathcal{I} \mathcal{I}; (r, \text{conv}') \in \text{results-gpv } \mathcal{I}' (\text{run-converter conv } q) \rrbracket$   
 $\Longrightarrow r \in \text{responses-}\mathcal{I} \mathcal{I} q \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \sqrt{\ }$

**lemma** *WT-converter-coinduct*[*consumes 1, case-names WT-converter, case-conclusion WT-converter WT-gpv results-gpv, coinduct pred: WT-converter*]:

**assumes**  $X \text{ conv}$   
**and**  $\bigwedge \text{conv } q \text{ } r \text{ conv}' . \llbracket X \text{ conv}; q \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$   
 $\implies \mathcal{I}' \vdash_g \text{run-converter } \text{conv } q \ \checkmark \wedge$   
 $((r, \text{conv}') \in \text{results-gpv } \mathcal{I}' (\text{run-converter } \text{conv } q) \longrightarrow r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q \wedge$   
 $(X \text{ conv}' \vee \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \ \checkmark))$   
**shows**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \checkmark$   
**using** *assms(1)* **by**(rule *WT-converter.coinduct*)(blast dest: *assms(2)*)

**lemma** *WT-converterD*:

**assumes**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \checkmark \ q \in \text{outs-}\mathcal{I} \ \mathcal{I}$   
**shows** *WT-converterD-WT*:  $\mathcal{I}' \vdash_g \text{run-converter } \text{conv } q \ \checkmark$   
**and** *WT-converterD-results*:  $(r, \text{conv}') \in \text{results-gpv } \mathcal{I}' (\text{run-converter } \text{conv } q)$   
 $\implies r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \ \checkmark$   
**using** *assms* **by**(auto elim: *WT-converter.cases*)

**lemma** *WT-converterD'*:

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

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

**lemma** *WT-converter-mono*:

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

**lemma** *callee-invariant-on-run-resource* [simp]: *callee-invariant-on run-resource* (WT-resource  $\mathcal{I}$ )  $\mathcal{I}$

**by**(*unfold-locales*)(*auto dest: WT-resourceD intro: WT-calleeI*)

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

**by** *simp*

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

**by**(*unfold-locales*)(*auto dest: WT-converterD*)

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

**by**(*rule raw-converter-invariant-run-converter*)

**lemma** *WT-converter- $\mathcal{I}$ -full:  $\mathcal{I}$ -full,  $\mathcal{I}$ -full  $\vdash_C$  conv  $\checkmark$*

**by**(*coinduction arbitrary: conv*)(*auto*)

**lemma** *WT-converter-map-converter* [WT-intro]:

$\mathcal{I}, \mathcal{I}' \vdash_C$  *map-converter*  $f g f' g' conv \checkmark$  **if**

$*$ : *map- $\mathcal{I}$  (inv-into UNIV f) (inv-into UNIV g)  $\mathcal{I}$ , map- $\mathcal{I}$   $f' g' \mathcal{I}' \vdash_C$  conv  $\checkmark$*

**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''  $\checkmark$*

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

## 2.6 Losslessness

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

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

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

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

**lemma** *plossless-converter-mono*:  
**assumes** \*:  $\text{plossless-converter } \mathcal{I}1 \mathcal{I}2 \text{ conv}$   
**and** *le*:  $\text{outs-}\mathcal{I} \mathcal{I}1' \subseteq \text{outs-}\mathcal{I} \mathcal{I}1 \mathcal{I}2 \leq \mathcal{I}2'$   
**and** *WT*:  $\mathcal{I}1, \mathcal{I}2 \vdash_C \text{conv} \checkmark$   
**shows**  $\text{plossless-converter } \mathcal{I}1' \mathcal{I}2' \text{ conv}$   
**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* ( $\lambda conv. plossless-converter \mathcal{I} \mathcal{I}' conv \wedge \mathcal{I}, \mathcal{I}' \vdash_C conv \checkmark$ )  
**by**(*unfold-locales*)(*auto dest: WT-converterD plossless-converterD*)

**interpretation** *run-plossless-converter: raw-converter-invariant*  
 $\mathcal{I} \mathcal{I}'$  *run-converter*  $\lambda conv. plossless-converter \mathcal{I} \mathcal{I}' conv \wedge \mathcal{I}, \mathcal{I}' \vdash_C conv \checkmark$  **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*  $s$ ) = ( $\lambda a. map-gpv (map-prod id \text{converter-of-callee}) id (callee s a)$ )

**end**

**lemma** *converter-of-callee-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**  
 $((S \text{====>} A \text{====>} rel-gpv'' (rel-prod B S) C R) \text{====>} S \text{====>} rel-converter A B C R)$   
*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*  $f g h k (converter-of-callee \text{callee } s) =$   
*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*:  $\bigwedge s q. q \in outs-\mathcal{I} \mathcal{I} \Longrightarrow \mathcal{I}' \vdash g \text{callee } s q \checkmark$

**and**  $res: \bigwedge s q r s'. \llbracket q \in \text{outs-}\mathcal{I} \ \mathcal{I}; (r, s') \in \text{results-gpv} \ \mathcal{I}' \ (\text{callee} \ s \ q) \rrbracket \implies r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q$   
**shows**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{converter-of-callee} \ \text{callee} \ s \ \checkmark$   
**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*

$:: ('a, 'b, 'out, 'in) \text{converter} \Rightarrow ('c, 'd, 'out, 'in) \text{converter} \Rightarrow ('a + 'c, 'b + 'd, 'out, 'in) \text{converter}$

**where**

*run-converter* (*parallel-converter* *conv1* *conv2*) = ( $\lambda ac.$  *case* *ac* *of*  
 $\text{Inl } a \Rightarrow \text{map-gpv} (\text{map-prod } \text{Inl} (\lambda \text{conv1}'. \text{parallel-converter } \text{conv1}' \ \text{conv2})) \ \text{id}$   
(*run-converter* *conv1* *a*)  
 $| \text{Inr } b \Rightarrow \text{map-gpv} (\text{map-prod } \text{Inr} (\lambda \text{conv2}'. \text{parallel-converter } \text{conv1} \ \text{conv2}')) \ \text{id}$   
(*run-converter* *conv2* *b*))

**lemma** *parallel-callee-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**

(*rel-converter* *A* *B* *C* *R*  $\implies$  *rel-converter* *A'* *B'* *C* *R*  $\implies$  *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*  $\&$  *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*]:

$\llbracket \text{plossless-converter } \mathcal{I}1 \ \mathcal{I} \ \text{conv1}; \text{plossless-converter } \mathcal{I}2 \ \mathcal{I} \ \text{conv2}; \mathcal{I}1, \mathcal{I} \vdash_C \text{conv1} \checkmark; \mathcal{I}2, \mathcal{I} \vdash_C \text{conv2} \checkmark \rrbracket$

$\implies \text{plossless-converter} (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \ \mathcal{I} \ (\text{parallel-converter } \text{conv1} \ \text{conv2})$

**by**(*coinduction arbitrary: conv1 conv2*)

(*clarsimp*; *erule* *PlusE*; *drule* (1) *plossless-converterD*; *drule* (1) *WT-converterD'*; *fastforce*)

**primcorec** *id-converter*  $:: ('a, 'b, 'a, 'b) \text{converter}$  **where**

*run-converter* *id-converter* = ( $\lambda a.$

*map-gpv* (*map-prod* *id* ( $\lambda.$  *id-converter*)) *id* (*Pause* *a* ( $\lambda b.$  *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*  $\checkmark$   
**by**(*coinduction*) *auto*

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

**definition** *fail-converter* :: (*'a, 'b, 'out, 'in*) *converter* **where**  
*fail-converter = Converter* ( $\lambda$ -. *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*  $\longleftrightarrow$   
 $\mathcal{I} = \text{bot}$  (**is** *?lhs*  $\longleftrightarrow$  *?rhs*)  
**proof**(*rule iffI*)  
**show** *?rhs* **if** *?lhs* **using** *that* **by**(*cases*)(*auto intro!: 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*  $\circ$  *f*) (*g*  $\circ$  *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*  $\circ$  *f*) (*g*  $\circ$  *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*  $\Rightarrow ('c, 'd, 'out', 'in')$  *converter*  $\Rightarrow ('a + 'c, 'b + 'd,$   
 $'out + 'out', 'in + 'in')$  *converter*

**where**

*run-converter* (*parallel-converter2* *conv1* *conv2*) = ( $\lambda ac.$  *case* *ac* *of*  
  *Inl* *a*  $\Rightarrow$  *map-gpv* (*map-prod* *Inl* ( $\lambda conv1'.$  *parallel-converter2* *conv1'* *conv2*))  
*id* (*left-gpv* (*run-converter* *conv1* *a*))  
  *Inr* *b*  $\Rightarrow$  *map-gpv* (*map-prod* *Inr* ( $\lambda conv2'.$  *parallel-converter2* *conv1* *conv2'*))  
*id* (*right-gpv* (*run-converter* *conv2* *b*)))

**lemma** *parallel-converter2-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**  
(*rel-converter* *A* *B* *C* *R*  $====>$  *rel-converter* *A'* *B'* *C'* *R'*  
 $====>$  *rel-converter* (*rel-sum* *A* *A'*) (*rel-sum* *B* *B'*) (*rel-sum* *C* *C'*) (*rel-sum* *R*  
*R'*))

*parallel-converter2* *parallel-converter2*

**unfolding** *parallel-converter2-def*

**supply** *left-gpv-parametric*'[*transfer-rule*] *right-gpv-parametric*'[*transfer-rule*] *map-gpv-parametric*'[*transfer-r*

**by** *transfer-prover*

**lemma** *map-converter-parallel-converter2*:

*map-converter* (*map-sum* *f* *f'*) (*map-sum* *g* *g'*) (*map-sum* *h* *h'*) (*map-sum* *k* *k'*)  
(*parallel-converter2* *conv1* *conv2*) =  
*parallel-converter2* (*map-converter* *f* *g* *h* *k* *conv1*) (*map-converter* *f'* *g'* *h'* *k'*  
*conv2*)

**using** *parallel-converter2-parametric*[*of*

*conversep* (*BNF-Def.Grp* *UNIV* *f*) *BNF-Def.Grp* *UNIV* *g* *BNF-Def.Grp*  
*UNIV* *h* *conversep* (*BNF-Def.Grp* *UNIV* *k*)

*conversep* (*BNF-Def.Grp* *UNIV* *f'*) *BNF-Def.Grp* *UNIV* *g'* *BNF-Def.Grp*  
*UNIV* *h'* *conversep* (*BNF-Def.Grp* *UNIV* *k'*)]



**unfolding** *sum.rel-conversep sum.rel-Grp*  
**by**(*simp add: rel-converter-Grp rel-fun-def Grp-iff*)

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

**lemma** *parallel-converter2-map1-out*:  
 $\text{parallel-converter}2 (\text{map-converter } f g h k \text{conv}1) \text{conv}2 =$   
 $\text{map-converter } (\text{map-sum } f \text{id}) (\text{map-sum } g \text{id}) (\text{map-sum } h \text{id}) (\text{map-sum } k \text{id})$   
 $(\text{parallel-converter}2 \text{conv}1 \text{conv}2)$   
**by**(*simp add: map-converter-parallel-converter2*)

**lemma** *parallel-converter2-map2-out*:  
 $\text{parallel-converter}2 \text{conv}1 (\text{map-converter } f g h k \text{conv}2) =$   
 $\text{map-converter } (\text{map-sum } \text{id } f) (\text{map-sum } \text{id } g) (\text{map-sum } \text{id } h) (\text{map-sum } \text{id } k)$   
 $(\text{parallel-converter}2 \text{conv}1 \text{conv}2)$   
**by**(*simp add: map-converter-parallel-converter2*)

**primcorec** *left-interface* ::  $('a, 'b, 'out, 'in) \text{converter} \Rightarrow ('a, 'b, 'out + 'out', 'in + 'in) \text{converter}$  **where**  
 $\text{run-converter } (\text{left-interface } \text{conv}) = (\lambda a. \text{map-gpv } (\text{map-prod } \text{id } \text{left-interface}) \text{id } (\text{left-gpv } (\text{run-converter } \text{conv } a)))$

**lemma** *left-interface-parametric [transfer-rule]*: **includes** *lifting-syntax* **shows**  
 $(\text{rel-converter } A B C R \implies \text{rel-converter } A B (\text{rel-sum } C C') (\text{rel-sum } R R'))$   
*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*  $\Rightarrow$  ('a, 'b, 'out' + 'out, 'in' + 'in) *converter* **where**  
*run-converter* (*right-interface conv*) = ( $\lambda a$ . *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 gpu.rel-refl*)  
**by** *force+*

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

**primcorec** *converter-of-resource* :: (*'a, 'b*) *resource*  $\Rightarrow$  (*'a, 'b, 'c, 'd*) *converter*  
**where**  
*run-converter (converter-of-resource res) = ( $\lambda x.$  map-gpv (map-prod id con-*  
*verter-of-resource) id (lift-spmf (run-resource res x)))*

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

**lemma** *plossless-converter-of-resource* [*plossless-intro*]:  
**assumes** *lossless-resource*  $\mathcal{I} \text{ res}$   
**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**  $\bigwedge s x. x \in \text{outs-}\mathcal{I} \mathcal{I}1 \implies \text{plossless-gpv } \mathcal{I}2$  (*callee s x*)  $\wedge$  ( $\forall (y, s') \in \text{results-gpv}$   
 $\mathcal{I}2$  (*callee s x*).  $y \in \text{responses-}\mathcal{I} \mathcal{I}1 x$ )  
**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**  $A :: 'a \text{ set}$   
**and**  $\mathcal{I} :: ('c, 'd) \mathcal{I}$   
**begin**

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

**end**

**lemma** *WT-restrict-converter* [*WT-intro*]:

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

**lemma** *pgen-lossless-restrict-gpv* [*simp*]:  
 $\mathcal{I} \vdash g \text{gpv} \checkmark \implies \text{pgen-lossless-gpv } b \mathcal{I} (\text{restrict-gpv } \mathcal{I} \text{gpv}) = \text{pgen-lossless-gpv } b \mathcal{I} \text{gpv}$   
**unfolding** *pgen-lossless-gpv-def* **by**(*simp add: expectation-gpv-restrict-gpv*)

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

**lemma** *plossless-map-converter*:  
*plossless-converter*  $\mathcal{I} \mathcal{I}' (\text{map-converter } f g h k \text{conv})$   
**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*  $\implies$  (*'out, 'in*) *resource*  $\implies$  (*'a, 'b*) *resource* **where**  
 $\text{run-resource } (\text{attach } \text{conv } \text{res}) = (\lambda a. \text{map-spmf } (\lambda (b, \text{conv}') . (b, \text{attach } \text{conv}' \text{res}')) (\text{exec-gpv } \text{run-resource } (\text{run-converter } \text{conv } a) \text{res}))$

**lemma** *attach-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**  
(*rel-converter*  $A B C R \implies \implies \text{rel-resource } C R \implies \implies \text{rel-resource } A B$ ) *attach attach*  
**unfolding** *attach-def*  
**supply** *exec-gpv-parametric[transfer-rule]* **by** *transfer-prover*

**lemma** *attach-map-converter*:  
 $\text{attach } (\text{map-converter } f g h k \text{conv}) \text{res} = \text{map-resource } f g (\text{attach } \text{conv } (\text{map-resource } h k \text{res}))$   
**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  $\sqsupset$  -)* **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*]:  $\llbracket \mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \surd; \mathcal{I}' \vdash_{\text{res}} \text{res } \surd \rrbracket \implies \mathcal{I} \vdash_{\text{res}} \text{attach } \text{conv } \text{res } \surd$   
**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*  $\mathcal{I} \mathcal{I}' \text{conv}$   
**and** *lossless-resource*  $\mathcal{I}' \text{res}$   
**and**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \surd \mathcal{I}' \vdash_{\text{res}} \text{res } \surd$   
**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*)  
 $\bigwedge b \text{conv}' . (b, \text{conv}') \in \text{results-gpv } \mathcal{I}'$  (*run-converter conv a*)  $\implies$  *plossless-converter*  $\mathcal{I} \mathcal{I}' \text{conv}'$  **by** *auto*  
**from** *WT-converterD'*[*OF lossless-resource(3,5)*] **have** *WT: \mathcal{I}' \vdash\_g run-converter conv a \surd*  
 $\bigwedge b \text{conv}' . (b, \text{conv}') \in \text{results-gpv } \mathcal{I}'$  (*run-converter conv a*)  $\implies b \in \text{responses-}\mathcal{I}$   $\mathcal{I} a \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \surd$  **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**  $(b, \text{res}'') \in ?\text{set}$   
**then obtain** *conv' res'* **where**  $*$ :  $((b, \text{conv}'), \text{res}') \in \text{set-spmf } (\text{exec-gpv run-resource } (\text{run-converter conv a}) \text{res})$   
**and** [*simp*]:  $\text{res}'' = \text{attach } \text{conv}' \text{res}'$  **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-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**  $?P b \text{res}''$  **by** *auto*  
**qed**  
**ultimately show** *?case ..*  
**qed**

**definition** *attach-callee*

$:: ('s \Rightarrow 'a \Rightarrow ('b \times 's, 'out, 'in) \text{gpv})$   
 $\Rightarrow ('s' \Rightarrow 'out \Rightarrow ('in \times 's') \text{spmf})$   
 $\Rightarrow ('s \times 's' \Rightarrow 'a \Rightarrow ('b \times 's \times 's') \text{spmf})$  **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= $\lambda l\ r.\ l = \text{resource-of-oracle res } r$  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-refl*)

**lemma** *attach-stateless-callee*:

*attach-callee (stateless-callee callee) oracle = extend-state-oracle ( $\lambda s\ q.\ \text{exec-gpv oracle (callee } q)\ s$ )*

**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-refl*)

**lemma** *attach-callee-parallel-intercept: includes lifting-syntax shows*

*attach-callee (parallel-intercept callee1 callee2) (plus-oracle oracle1 oracle2) = (rprodl  $\text{----} > \text{id} \text{----} > \text{map-spmf (map-prod id lprodr)}$ ) (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  $?S = \lambda res1 res2 res1. res1 res2 = \text{parallel-resource}$ 
res1 res2 and
     $A=(=)$  and  $CALL=\lambda l r. l = \text{Inl } r$  and  $R=\lambda l r. l = \text{Inl } r,$ 
    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  $?S = \lambda res1 res2 res2. res1 res2 = \text{parallel-resource}$ 
res1 res2 and
     $A=(=)$  and  $CALL=\lambda l r. l = \text{Inr } r$  and  $R=\lambda l r. l = \text{Inr } r,$ 
    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  $\Rightarrow$  ('out, 'in, 'out', 'in')
converter  $\Rightarrow$  ('a, 'b, 'out', 'in') converter where
  run-converter (comp-converter conv1 conv2) = ( $\lambda a.$ 
    map-gpv ( $\lambda (b, conv1')$ , 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  $\implies$  rel-converter C R C' R'  $\implies$  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  $\Rightarrow$  -) conversep (BNF-Def.Grp UNIV (id ::
  'in  $\Rightarrow$  -))
  ]
  apply(unfold rel-converter-Grp)
  apply(simp add: rel-fun-def Grp-iff)
  apply(rewrite at  $\forall$  - - .  $\sqsupset \longrightarrow$  - in asm conversep-iff[symmetric])
  apply(unfold rel-converter-conversep[symmetric] conversep-conversep eq-alt[symmetric])
  apply(rewrite in rel-converter - -  $\sqsupset$  - in asm conversep-eq)
  apply(rewrite in rel-converter - - -  $\sqsupset$  in asm conversep-eq[symmetric])
  apply(rewrite in rel-converter - -  $\sqsupset$  - in asm eq-alt)
  apply(rewrite in rel-converter - - -  $\sqsupset$  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  $\Rightarrow$  -) conversep (BNF-Def.Grp UNIV (id :: 'b
   $\Rightarrow$  -))
    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  $\forall$  - .  $\sqsupset \longrightarrow$  - 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 -  $\sqsupset$  in asm conversep-eq)
  apply(rewrite in rel-converter  $\sqsupset$  - in asm conversep-eq[symmetric])
  apply(rewrite in rel-converter  $\sqsupset$  - in asm eq-alt)
  apply(rewrite in rel-converter -  $\sqsupset$  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= $\lambda(l, r)$*   
*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= $\lambda(l, r)$  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]:*  
 $\llbracket \mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \checkmark; \mathcal{I}', \mathcal{I}'' \vdash_C \text{conv}' \checkmark \rrbracket \implies \mathcal{I}, \mathcal{I}'' \vdash_C \text{comp-converter conv conv}'$

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

**and inline:** *((b, conv1), conv2)  $\in results-gpv \mathcal{I}'' (inline run-converter (run-converter conv a) conv')$*   
**by** *auto*  
**from** *run-plossless-converter.results-gpv-inline[OF inline conv2] plossless-converter(2,4)*  
**have** *run: (b, conv1)  $\in results-gpv \mathcal{I}' (run-converter conv a)$*   
**and** *\*: plossless-converter  $\mathcal{I}' \mathcal{I}'' conv2 \mathcal{I}', \mathcal{I}'' \vdash_C conv2 \checkmark$  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= $\lambda$ l
r. l = r  $\wedge$  r = id-converter, THEN rel-funD, THEN rel-funD, THEN rel-funD]
        , auto simp add: id-oracle-def intro!: rel-funI gpv.rel-refl-strong)
    show ?thesis
      by (coinduction arbitrary:conv)
        (auto simp add: lem4 gpv.rel-map intro!:rel-funI gpv.rel-refl-strong)
  qed

```

```

lemma comp-converter-of-callee: comp-converter (converter-of-callee callee1 s1) (converter-of-callee
callee2 s2)
  = converter-of-callee ( $\lambda$ (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= $\lambda$ c s. c = converter-of-callee cal2 s, THEN rel-funD, THEN rel-funD, THEN
rel-funD])
    apply(auto simp add: gpv.rel-eq rel-fun-def gpv.rel-map intro!: gpv.rel-refl-strong)
    by (auto simp add: rprodl-def intro!:exI)
  done

```

**lemmas** comp-converter-of-callee' = comp-converter-eqs[OF comp-converter-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= $\lambda$ c1 c2. c1 = parallel-converter2 c2
conv2r and C= $\lambda$ l r. l = Inl r and R= $\lambda$ l r. l = Inl r and C' = (=) and R'=(=),
THEN rel-funD, THEN rel-funD, THEN rel-funD])
  subgoal by(auto split: sum.split simp add: gpv.rel-map rel-gpv-conv-rel-gpv''[symmetric]
intro!: gpv.rel-refl-strong rel-funI)
    apply(rule left-gpv-Inl-transfer)
    apply(auto 4 6 simp add: sum.map-id)
  done
subgoal for conv1l conv1r conv2l conv2r input
  apply(subst right-gpv-map[where h=id])
  apply(simp add: gpv.rel-map right-gpv-inline)

```

```

apply(unfold rel-gpv-conv-rel-gpv')
apply(rule rel-gpv''-mono[THEN predicate2D, rotated -1])
  apply(rule inline-parametric'[where S= $\lambda c1 c2$ .  $c1 = \text{parallel-converter2}$ 
conv2l  $c2$  and  $C = \lambda l r$ .  $l = \text{Inr } r$  and  $R = \lambda l r$ .  $l = \text{Inr } r$  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*  $\Rightarrow$  *enat*  
 $\Rightarrow$  *bool where*

```

interaction-any-bounded-converter conv n if
 $\bigwedge a$ . interaction-any-bounded-by (run-converter conv a) n
 $\bigwedge a b$  conv'.  $(b, \text{conv}') \in \text{results}'\text{-gpv (run-converter conv a)} \implies \text{interaction-any-bounded-converter}$ 
conv' n

```

**lemma** *interaction-any-bounded-converterD:*

```

assumes interaction-any-bounded-converter conv n
shows interaction-any-bounded-by (run-converter conv a) n  $\wedge$  ( $\forall (b, \text{conv}') \in \text{results}'\text{-gpv}$ 
(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  $n \leq m$ 

```

```

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  $\infty$ 
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 - -  $\Rightarrow$  fail | interaction-any-bounded-converter
- -  $\Rightarrow$  fail | -  $\Rightarrow$   $\langle$ solves  $\langle$ clarsimp simp add: simp $\rangle$  $\rangle$  | 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 ( $\lambda$ 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  $\bigwedge x. \textit{consider}' (Inl x) \Longrightarrow \textit{consider} x$ 
shows interaction-bounded-by consider' (left-gpv gpv) n
proof –
define ib :: ('b, 'a, 'c) gpv  $\Rightarrow$  - where ib  $\equiv$  interaction-bound consider
have interaction-bound consider' (left-gpv gpv)  $\leq$  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  $\bigwedge x. \text{consider}' (Inr x) \implies \text{consider } x$ 
  shows interaction-bounded-by consider' (right-gpv gpv) n
proof -
  define ib :: ('b, 'a, 'c) gpv  $\Rightarrow$  - where  $ib \equiv \text{interaction-bound consider}$ 
  have interaction-bound consider' (right-gpv gpv)  $\leq ib$  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: interac-
tion-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= $\lambda$ conv2. interac-
tion-any-bounded-converter conv2 m and consider'= $\lambda$ -. 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 interac-
tion-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**  $\bigwedge s x. \text{interaction-any-bounded-by } (\text{conv } s \ x) \ n$   
 **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 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  $\Rightarrow$  ('a, 'b) resource  $\Rightarrow$  ('a, 'b) resource  $\Rightarrow$  bool
( $\langle$ -  $\vdash_R$ / -  $\sim$ /  $\rightarrow$  [100, 99, 99] 99)

```

```

for A where
  eq-resource-onI: A  $\vdash_R$  res  $\sim$  res' if
     $\bigwedge a. a \in A \Longrightarrow$  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  $\bigwedge$  res res' a.  $\llbracket$  X res res'; a  $\in$  A  $\rrbracket$ 
     $\Longrightarrow$  rel-spmf (rel-prod (=) ( $\lambda$ res res'. X res res'  $\vee$  A  $\vdash_R$  res  $\sim$  res'))
    (run-resource res a) (run-resource res' a)
  shows A  $\vdash_R$  res  $\sim$  res'
  using assms(1) by(rule eq-resource-on.coinduct)(auto dest: assms(2))

```

```

lemma eq-resource-onD:

```

```

  assumes A  $\vdash_R$  res  $\sim$  res' a  $\in$  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  $\sim$  res

```

```

  by(coinduction arbitrary: res)(auto intro: rel-spmf-refl)

```

```

lemma eq-resource-on-reflI: res = res'  $\Longrightarrow$  A  $\vdash_R$  res  $\sim$  res'

```

```

  by(simp add: eq-resource-on-refl)

```

```

lemma eq-resource-on-sym: A  $\vdash_R$  res  $\sim$  res' if A  $\vdash_R$  res'  $\sim$  res

```

```

  using that

```

```

  by(coinduction arbitrary: res res')

```

```

    (drule (1) eq-resource-onD, rewrite in  $\sqsupset$  conversep-iff[symmetric]

```

```

    , auto simp add: spmf-rel-conversep[symmetric] elim!: rel-spmf-mono)

```

```

lemma eq-resource-on-trans [trans]: A  $\vdash_R$  res  $\sim$  res'' if A  $\vdash_R$  res  $\sim$  res' A  $\vdash_R$  res'
 $\sim$  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**  $UNIV \vdash_R res \sim res'$   
**using** *that* **by**(*coinduction arbitrary: res res'*)(*auto dest: eq-resource-onD*)

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

**lemma** *eq-resource-on-mono*:  $\llbracket A' \vdash_R res \sim res'; A \subseteq A' \rrbracket \implies A \vdash_R res \sim res'$   
**by**(*coinduction arbitrary: res res'*)(*auto dest: eq-resource-onD elim!: rel-spmf-mono*)

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

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

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

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

**inductive** *eq-I-generat* ::  $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('out, 'in) \mathcal{I} \Rightarrow ('c \Rightarrow 'd \Rightarrow bool)$

$\Rightarrow ('a, 'out, 'in \Rightarrow 'c) \text{ generat} \Rightarrow ('b, 'out, 'in \Rightarrow 'd) \text{ generat} \Rightarrow \text{bool}$   
**for**  $A \mathcal{I} D$  **where**  
*Pure*:  $\text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D (Pure\ x) (Pure\ y) \text{ if } A\ x\ y$   
*IO*:  $\text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D (IO\ out\ c) (IO\ out\ c') \text{ if } out \in \text{outs-}\mathcal{I}\ \mathcal{I} \wedge \text{input. input} \in \text{responses-}\mathcal{I}\ \mathcal{I}\ out \Longrightarrow D\ (c\ input)\ (c'\ input)$

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

**inductive-simps**  $\text{eq-}\mathcal{I}\text{-generat-simps}$  [*simp, code*]:

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

**inductive-simps**  $\text{eq-}\mathcal{I}\text{-generat-iff1}$ :

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

**inductive-simps**  $\text{eq-}\mathcal{I}\text{-generat-iff2}$ :

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

**lemma**  $\text{eq-}\mathcal{I}\text{-generat-mono}'$ :

$\llbracket \text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D\ x\ y; \wedge x\ y. A\ x\ y \Longrightarrow A'\ x\ y; \wedge x\ y. D\ x\ y \Longrightarrow D'\ x\ y; \mathcal{I} \leq \mathcal{I}' \rrbracket$   
 $\Longrightarrow \text{eq-}\mathcal{I}\text{-generat } A' \mathcal{I}' D'\ x\ y$   
**by**(*auto 4 4 elim!*:  $\text{eq-}\mathcal{I}\text{-generat.cases simp add: le-}\mathcal{I}\text{-def}$ )

**lemma**  $\text{eq-}\mathcal{I}\text{-generat-mono}$ :  $\text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D \leq \text{eq-}\mathcal{I}\text{-generat } A' \mathcal{I}' D'$  **if**  $A \leq A' D \leq D' \mathcal{I} \leq \mathcal{I}'$

**using** *that by*(*auto elim!*:  $\text{eq-}\mathcal{I}\text{-generat-mono}' \text{ dest: predicate2D}$ )

**lemma**  $\text{eq-}\mathcal{I}\text{-generat-mono}''$  [*mono*]:

$\llbracket \wedge x\ y. A\ x\ y \longrightarrow A'\ x\ y; \wedge x\ y. D\ x\ y \longrightarrow D'\ x\ y \rrbracket$   
 $\Longrightarrow \text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D\ x\ y \longrightarrow \text{eq-}\mathcal{I}\text{-generat } A' \mathcal{I}' D'\ x\ y$   
**by**(*auto elim!*:  $\text{eq-}\mathcal{I}\text{-generat-mono}'$ )

**lemma**  $\text{eq-}\mathcal{I}\text{-generat-conversep}$ :  $\text{eq-}\mathcal{I}\text{-generat } A^{-1-1} \mathcal{I} D^{-1-1} = (\text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D)^{-1-1}$

**by**(*fastforce elim!*:  $\text{eq-}\mathcal{I}\text{-generat.cases}$ )

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

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

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

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

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

**lemma** *eq-I-generat-map2*:  
 $eq\mathcal{I}\text{-generat } A \mathcal{I} D \ generat \text{ (map-generat } f \text{ id } ((\circ) \ g) \ generat') \longleftrightarrow$   
 $eq\mathcal{I}\text{-generat } (\lambda x \ y. A \ x \ (f \ y)) \mathcal{I} (\lambda x \ y. D \ x \ (g \ y)) \ generat \ generat'$   
**by**(*cases generat*; *cases generat'*) *auto*

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

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

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

**lemma** *eq-I-gpv-coinduct* [*consumes 1*, *case-names eq-I-gpv*, *coinduct pred: eq-I-gpv*]:  
**assumes**  $X \ gpv \ gpv'$   
**and**  $\bigwedge gpv \ gpv'. X \ gpv \ gpv'$   
 $\implies rel\text{-spmf } (eq\mathcal{I}\text{-generat } A \mathcal{I} \ (\lambda gpv \ gpv'. X \ gpv \ gpv' \vee eq\mathcal{I}\text{-gpv } A \mathcal{I} \ gpv \ gpv')) \ (the\text{-gpv } \ gpv) \ (the\text{-gpv } \ gpv')$   
**shows**  $eq\mathcal{I}\text{-gpv } A \mathcal{I} \ gpv \ gpv'$   
**using** *assms(1)* **by**(*rule eq-I-gpv.coinduct*)(*blast dest: assms(2)*)

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

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

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

**lemma** *eq-I-gpv-Pause*:

$\llbracket \text{out} \in \text{outs-}\mathcal{I} \mathcal{I}; \bigwedge \text{input. input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out} \implies \text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} (\text{rpv input})$   
 $(\text{rpv' input}) \rrbracket$   
 $\implies \text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} (\text{Pause out rpv}) (\text{Pause out rpv'})$   
**by**(rule *eq-I-gpvI*) *simp*

**lemma** *eq-I-gpv-mono*:  $\text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \leq \text{eq-}\mathcal{I}\text{-gpv } A' \mathcal{I}'$  **if**  $A: A \leq A' \mathcal{I} \leq \mathcal{I}'$

**proof**

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

**lemma** *eq-I-gpv-mono'*:

$\llbracket \text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \text{ gpv } \text{gpv}'; \bigwedge x y. A x y \implies A' x y; \mathcal{I} \leq \mathcal{I}' \rrbracket \implies \text{eq-}\mathcal{I}\text{-gpv } A' \mathcal{I}'$   
 $\text{gpv } \text{gpv}'$   
**by**(*blast intro: eq-I-gpv-mono[THEN predicate2D]*)

**lemma** *eq-I-gpv-mono''* [*mono*]:

$\text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \text{ gpv } \text{gpv}' \longrightarrow \text{eq-}\mathcal{I}\text{-gpv } A' \mathcal{I}' \text{ gpv } \text{gpv}'$  **if**  $\bigwedge x y. A x y \longrightarrow A' x y$   
**using that** **by**(*blast intro: eq-I-gpv-mono'*)

**lemma** *eq-I-gpv-conversep*:  $\text{eq-}\mathcal{I}\text{-gpv } A^{-1-1} \mathcal{I} = (\text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I})^{-1-1}$

**proof**(*intro ext iffI; simp*)

**show**  $\text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \text{ gpv } \text{gpv}'$  **if**  $\text{eq-}\mathcal{I}\text{-gpv } A^{-1-1} \mathcal{I} \text{ gpv}' \text{ gpv}$  **for**  $A$  **and**  $\text{gpv } \text{gpv}'$   
**using that**  
**by**(*coinduction arbitrary: gpv gpv'*)  
*(drule eq-I-gpvD, rewrite in  $\sqsupset$  conversep-iff[symmetric]*  
*, auto simp add: pmf.rel-conversep[symmetric] option.rel-conversep[symmetric]*  
 $\text{eq-}\mathcal{I}\text{-generat-conversep[symmetric] elim: eq-}\mathcal{I}\text{-generat-mono' rel-spmf-mono}$ )

**from this**[of *conversep A*] **show**  $\text{eq-}\mathcal{I}\text{-gpv } A^{-1-1} \mathcal{I} \text{ gpv}' \text{ gpv}$  **if**  $\text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \text{ gpv}$   
 $\text{gpv}'$  **for**  $\text{gpv } \text{gpv}'$

**using that** **by** *simp*

**qed**

**lemma** *eq-I-gpv-reflI*:

$\llbracket \bigwedge x. x \in \text{results-gpv } \mathcal{I} \text{ gpv} \implies A x x; \mathcal{I} \vdash \text{g } \text{gpv } \checkmark \rrbracket \implies \text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \text{ gpv } \text{gpv}$   
**by**(*coinduction arbitrary: gpv*)(*auto intro!: rel-spmf-reflI eq-I-generat-reflI elim!:*  
*generat.set-cases intro: results-gpv.intros dest: WT-gpvD*)

**lemma** *eq-I-gpv-into-rel-gpv*:  $\text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I}\text{-full } \text{gpv } \text{gpv}' \implies \text{rel-gpv } A (=) \text{ gpv } \text{gpv}'$

**by**(*coinduction arbitrary: gpv gpv'*)

*(drule eq-I-gpvD, auto elim: spmf-rel-mono-strong generat.rel-mono-strong dest: eq-I-generat-into-rel-generat )*

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

$\mathcal{I}$  (is ?lhs = ?rhs)  
**proof**(intro ext iffI relcomppI; (elim relcomppE)?)  
**fix** gpv gpv''  
**assume** \*: ?lhs gpv gpv''  
**define** middle **where** middle = corec-gpv ( $\lambda$ (gpv, gpv'').  
map-spmf (map-generat (relcompp-witness A A') (relcompp-witness (=) (=))  
(( $\circ$ ) Inr  $\circ$  rel-witness-fun (=) (=))  $\circ$   
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 (=) (=))  
(( $\circ$ ) middle  $\circ$  rel-witness-fun (=) (=))  $\circ$   
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'  $\longleftrightarrow$  eq- $\mathcal{I}$ -gpv  
( $\lambda$ x. A (f x))  $\mathcal{I}$  gpv gpv' (is ?lhs  $\longleftrightarrow$  ?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 - =*  $\sqsupset$  *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*:

$\llbracket \text{eq-I-gpv } A \text{ I gpv gpv'}; I s \rrbracket \implies \text{rel-spmf } (\text{rel-prod } A \text{ (eq-onp } I)) \text{ (exec-gpv callee gpv } s) \text{ (exec-gpv callee gpv'} \text{ s)}$

**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.prem**

**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-refl*)

**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**:  $\bigwedge \text{gpv gpv'}. X \text{ gpv gpv'}$

$\implies \text{rel-spmf } (\text{eq-I-generat } A \text{ I } (\lambda \text{gpv gpv'}. X \text{ gpv gpv'} \vee \text{eq-I-gpv } A \text{ I gpv gpv'} \vee$

$(\exists \text{gpv'' gpv'''} (B :: 'b \Rightarrow 'b' \Rightarrow \text{bool}) f g. \text{gpv} = \text{bind-gpv gpv'' } f \wedge \text{gpv}' = \text{bind-gpv gpv''' } g \wedge \text{eq-I-gpv } B \text{ I gpv'' gpv'''} \wedge (\text{rel-fun } B \text{ X}) f g)) \text{ (the-gpv gpv)}$

*(the-gpv gpv')*

**shows** *eq-I-gpv A I gpv gpv'*

**proof** –

**fix** *x y*

**define** *gpv''* :: ('b, 'out, 'in) *gpv* **where** *gpv''*  $\equiv \text{Done } x$

**define** *f* :: 'b  $\Rightarrow$  ('a, 'out, 'in) *gpv* **where** *f*  $\equiv \lambda-. \text{gpv}$

**define** *gpv'''* :: ('b', 'out, 'in) *gpv* **where** *gpv'''*  $\equiv \text{Done } y$

**define** *g* :: 'b'  $\Rightarrow$  ('a', 'out, 'in) *gpv* **where** *g*  $\equiv \lambda-. \text{gpv}'$

**have** *eq-I-gpv* ( $\lambda x y. X (f x) (g y)$ ) *I gpv'' gpv'''* **using** *X*

**by** (*simp add: f-def g-def gpv''-def gpv'''-def*)

**then have** *eq-I-gpv A I (bind-gpv gpv'' f) (bind-gpv gpv''' g)*

**by** (*coinduction arbitrary: gpv'' f gpv''' g*)

(*drule eq-I-gpvD, (clarsimp simp add: bind-gpv.sel spmf-rel-map simp del:*

*bind-gpv-sel' elim!: rel-spmf-bindI split!: generat.split dest!: step)*

, *erule rel-spmf-mono, (erule eq-I-generat.cases; clarsimp), (erule meta-allE,*

*erule (1) meta-impE)*

, (*fastforce | force intro: exI*[**where** *x=Done -*] *elim!: eq-I-gpv-mono' dest:*

*rel-funD*)+)

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

**context**

**fixes**  $S :: 's1 \Rightarrow 's2 \Rightarrow \text{bool}$   
**and**  $\text{callee1} :: 's1 \Rightarrow 'out \Rightarrow ('in \times 's1, 'out', 'in') \text{ gpv}$   
**and**  $\text{callee2} :: 's2 \Rightarrow 'out \Rightarrow ('in \times 's2, 'out', 'in') \text{ gpv}$   
**and**  $\mathcal{I} :: ('out, 'in) \mathcal{I}$   
**and**  $\mathcal{I}' :: ('out', 'in') \mathcal{I}$   
**assumes**  $\text{callee}: \bigwedge s1 s2 q. \llbracket S s1 s2; q \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \Longrightarrow \text{eq-}\mathcal{I}\text{-gpv} (\text{rel-prod} (\text{eq-onp} (\lambda r. r \in \text{responses-}\mathcal{I} \mathcal{I} q)) S) \mathcal{I}' (\text{callee1 } s1 q) (\text{callee2 } s2 q)$   
**begin**

**lemma** *eq- $\mathcal{I}$ -gpv-inline1*:

**includes** *lifting-syntax*  
**assumes**  $S s1 s2 \text{ eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \text{ gpv1 } \text{ gpv2}$   
**shows**  $\text{rel-spmf} (\text{rel-sum} (\text{rel-prod } A S) (\lambda(q, \text{rpv1}, \text{rpv2}) (q', \text{rpv1}', \text{rpv2}')). q = q' \wedge q' \in \text{outs-}\mathcal{I} \mathcal{I}' \wedge (\exists q'' \in \text{outs-}\mathcal{I} \mathcal{I}. (\forall r \in \text{responses-}\mathcal{I} \mathcal{I}' q'. \text{eq-}\mathcal{I}\text{-gpv} (\text{rel-prod} (\text{eq-onp} (\lambda r'. r' \in \text{responses-}\mathcal{I} \mathcal{I} q'')) S) \mathcal{I}' (\text{rpv1 } r) (\text{rpv1}' r')) \wedge (\forall r' \in \text{responses-}\mathcal{I} \mathcal{I} q''. \text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} (\text{rpv2 } r') (\text{rpv2}' r'))))) (\text{inline1 } \text{callee1 } \text{ gpv1 } s1) (\text{inline1 } \text{callee2 } \text{ gpv2 } s2)$   
**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.premis 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**  $\text{gpv}: \text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} \text{ gpv1 } \text{ gpv2}$   
**shows**  $\text{eq-}\mathcal{I}\text{-gpv} (\text{rel-prod } A S) \mathcal{I}' (\text{inline } \text{callee1 } \text{ gpv1 } s1) (\text{inline } \text{callee2 } \text{ gpv2 } s2)$   
**using**  $S \text{ gpv}$   
**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-I-gpv-left-gpv-cong*:

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

**lemma** *eq-I-gpv-right-gpv-cong*:

*eq-I-gpv A I' gpv gpv'  $\implies$  eq-I-gpv A (I  $\oplus_{\mathcal{I}}$  I') (right-gpv gpv) (right-gpv gpv')*  
**by**(*coinduction arbitrary: gpv gpv'*)  
(*drule eq-I-gpvD, auto 4 4 simp add: spmf-rel-map elim!: rel-spmf-mono eq-I-generat.cases*)

**lemma** *eq-I-gpvD-WT1*:  $\llbracket \text{eq-I-gpv } A \ \mathcal{I} \ \text{gpv} \ \text{gpv}'; \ \mathcal{I} \vdash_g \ \text{gpv} \ \checkmark \rrbracket \implies \mathcal{I} \vdash_g \ \text{gpv}' \ \checkmark$

**by**(*coinduction arbitrary: gpv gpv'*)(*fastforce simp add: eq-I-generat-iff2 dest: WT-gpv-ContD eq-I-gpvD dest!: rel-setD2 set-spmf-parametric[THEN rel-funD]*)

**lemma** *eq-I-gpvD-results-gpv2*:

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

**coinductive** *eq-I-converter* :: ('a, 'b)  $\mathcal{I} \Rightarrow$  ('out, 'in)  $\mathcal{I} \Rightarrow$  ('a, 'b, 'out, 'in) *converter*  $\Rightarrow$  ('a, 'b, 'out, 'in) *converter*  $\Rightarrow$  bool

( $\langle -, - \vdash_C / - \sim / \rightarrow [100, 0, 99, 99] \ 99$ )

**for**  $\mathcal{I} \ \mathcal{I}'$  **where**

*eq-I-converterI*:  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}'$  **if**

$\bigwedge q. q \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{eq-I-gpv} (\text{rel-prod} (\text{eq-onp} (\lambda r. r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q))) (\text{eq-I-converter } \mathcal{I} \ \mathcal{I}') \ \mathcal{I}' (\text{run-converter } \text{conv} \ q) (\text{run-converter } \text{conv}' \ q)$

**lemma** *eq-I-converter-coinduct* [*consumes 1, case-names eq-I-converter, coinduct pred: eq-I-converter*]:

**assumes**  $X \ \text{conv} \ \text{conv}'$

**and**  $\bigwedge \text{conv} \ \text{conv}' \ q. \llbracket X \ \text{conv} \ \text{conv}'; \ q \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$

$\implies \text{eq-I-gpv} (\text{rel-prod} (\text{eq-onp} (\lambda r. r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q))) (\lambda \text{conv} \ \text{conv}'. X \ \text{conv} \ \text{conv}' \vee \ \mathcal{I}, \mathcal{I}' \vdash_C \ \text{conv} \sim \ \text{conv}') \ \mathcal{I}'$

(*run-converter conv q*) (*run-converter conv' q*)

**shows**  $\mathcal{I}, \mathcal{I}' \vdash_C \ \text{conv} \sim \ \text{conv}'$

**using** *assms(1)* **by**(*rule eq-I-converter.coinduct*)(*blast dest: assms(2)*)

**lemma** *eq-I-converterD*:

*eq-I-gpv (rel-prod (eq-onp ( $\lambda r. r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q$ ))) (eq-I-converter  $\mathcal{I} \ \mathcal{I}')$   $\mathcal{I}'$  (*run-converter conv q*) (*run-converter conv' q*)*

**if**  $\mathcal{I}, \mathcal{I}' \vdash_C \ \text{conv} \sim \ \text{conv}' \ q \in \text{outs-}\mathcal{I} \ \mathcal{I}$

**using** *that* **by**(*blast elim: eq-I-converter.cases*)

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

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

**lemma** *eq-I-converter-trans* [*trans*]:  
 $\llbracket \mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}'; \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \sim \text{conv}'' \rrbracket \implies \mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}''$   
**by**(*coinduction arbitrary: conv conv' conv''*)  
(*drule* (1) *eq-I-converterD*) $+$ , *drule* (1) *eq-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-I-gpv-mono* $\wedge$ )

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

**lemma** *eq-I-converter-eq*:  $\text{conv}1 = \text{conv}2$  **if**  $\mathcal{I}\text{-full}, \mathcal{I}\text{-full} \vdash_C \text{conv}1 \sim \text{conv}2$   
**using that**  
**by**(*coinduction arbitrary: conv1 conv2*)  
(*auto simp add: eq-I-gpv-into-rel-gpv eq-onp-def intro!*: *rel-funI* *elim!*: *gpv.rel-mono-strong*  
*eq-I-gpv-into-rel-gpv dest: eq-I-converterD*)

**lemma** *eq-I-attach-on*:  
**assumes**  $\mathcal{I}' \vdash_{\text{res}} \text{res} \checkmark \mathcal{I}\text{-uniform } A \text{ UNIV}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}'$   
**shows**  $A \vdash_R \text{attach conv res} \sim \text{attach conv}' \text{res}$   
**using** *assms*  
**by**(*coinduction arbitrary: conv conv' res*)  
(*auto*  $\not\llcorner$  *simp add: eq-onp-def spmf-rel-map dest: eq-I-converterD intro!*: *rel-funI*  
*run-resource.eq-I-exec-gpv*[*THEN rel-spmf-mono*])

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

**lemma** *eq-I-attach*:

$\llbracket \mathcal{I}' \vdash_{\text{res}} \text{res } \checkmark; \mathcal{I}\text{-full}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}' \rrbracket \implies \text{attach conv res} = \text{attach conv}'_{\text{res}}$

**by**(rule eq-resource-on-UNIV-D)(simp add: eq- $\mathcal{I}$ -attach-on)

**lemma** eq- $\mathcal{I}$ -comp-cong:

$\llbracket \mathcal{I}1, \mathcal{I}2 \vdash_C \text{conv}1 \sim \text{conv}1'; \mathcal{I}2, \mathcal{I}3 \vdash_C \text{conv}2 \sim \text{conv}2' \rrbracket$

$\implies \mathcal{I}1, \mathcal{I}3 \vdash_C \text{comp-converter conv}1 \text{ conv}2 \sim \text{comp-converter conv}1' \text{ conv}2'$

**by**(coinduction arbitrary: conv1 conv2 conv1' conv2')

(clarsimp, rule eq- $\mathcal{I}$ -gpv-mono'[OF eq- $\mathcal{I}$ -gpv-inline[**where**  $S = \text{eq-}\mathcal{I}$ -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**  $\mathcal{I}\text{-full}, \mathcal{I} \vdash_C \text{conv}1 \sim \text{conv}1' \mathcal{I}, \mathcal{I}\text{-full} \vdash_C \text{conv}2 \sim \text{conv}2'$

**by**(rule eq- $\mathcal{I}$ -converter-eq)(rule eq- $\mathcal{I}$ -comp-cong[OF that])

**lemma** parallel-converter2-id-id:

$\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_C \text{parallel-converter2 id-converter id-converter} \sim \text{id-converter}$

**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:

$\llbracket \mathcal{I}1, \mathcal{I}1' \vdash_C \text{conv}1 \sim \text{conv}1'; \mathcal{I}2, \mathcal{I}2' \vdash_C \text{conv}2 \sim \text{conv}2' \rrbracket$

$\implies \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C \text{parallel-converter2 conv}1 \text{ conv}2 \sim \text{parallel-converter2 conv}1' \text{ conv}2'$

**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:  $\mathcal{I}, \mathcal{I}' \vdash_C \text{id-converter} \sim \text{id-converter}$  **if**  $\mathcal{I} \leq \mathcal{I}'$

**by**(rule eq- $\mathcal{I}$ -converter-mono[OF - order-refl that])(rule eq- $\mathcal{I}$ -converter-refl[OF WT-converter-id])

**lemma** eq- $\mathcal{I}$ -converterD-WT1:

**assumes**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}1 \sim \text{conv}2$  **and**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}1 \checkmark$

**shows**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}2 \checkmark$

**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**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}1 \sim \text{conv}2$

**shows**  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}1 \checkmark \iff \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}2 \checkmark$

**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**  *$\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \checkmark$*   
**and** *outs- $\mathcal{I}$   $\mathcal{I} \subseteq A$*   
**shows**  *$\mathcal{I}, \mathcal{I}' \vdash_C \text{restrict-converter A } \mathcal{I}' \text{cnv} \sim \text{cnv}$*   
**using** *assms(1)*  
**by**(*coinduction arbitrary: cnv*)  
*(use assms(2) in <auto intro!: eq- $\mathcal{I}$ -gpv-refl 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**  *$\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \sim \text{cnv}'$*   
**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 ( $\lambda s. \text{map-spmf fst (callee s x)}$ )*  
*| trace-callee callee p ((a, b) # xs) x =*

$trace\text{-}callee\ callee\ (cond\text{-}spmf\text{-}fst\ (bind\text{-}spmf\ p\ (\lambda s. callee\ s\ a))\ b)\ xs\ x$

**definition**  $trace\text{-}callee\text{-}eq :: ('a, 'b, 's1)\ callee \Rightarrow ('a, 'b, 's2)\ callee \Rightarrow 'a\ set \Rightarrow 's1\ spmf \Rightarrow 's2\ spmf \Rightarrow bool$  **where**  
 $trace\text{-}callee\text{-}eq\ callee1\ callee2\ A\ p\ q \iff$   
 $(\forall xs. set\ xs \subseteq A \times UNIV \longrightarrow (\forall x \in A. trace\text{-}callee\ callee1\ p\ xs\ x = trace\text{-}callee\ callee2\ q\ xs\ x))$

**abbreviation**  $trace\text{-}callee\text{-}eq' :: 'a\ set \Rightarrow ('a, 'b, 's1)\ callee \Rightarrow 's1 \Rightarrow ('a, 'b, 's2)\ callee \Rightarrow 's2 \Rightarrow bool$   
 $(\langle \cdot \vdash_C / (-'((-)') \rangle \approx / (-'((-)') \rangle [90, 0, 0, 0, 0] 91)$   
**where**  $trace\text{-}callee\text{-}eq'\ A\ callee1\ s1\ callee2\ s2 \equiv trace\text{-}callee\text{-}eq\ callee1\ callee2\ A\ (return\text{-}spmf\ s1)\ (return\text{-}spmf\ s2)$

**lemma**  $trace\text{-}callee\text{-}eqI$ :  
**assumes**  $\bigwedge xs\ x. \llbracket set\ xs \subseteq A \times UNIV; x \in A \rrbracket$   
 $\implies trace\text{-}callee\ callee1\ p\ xs\ x = trace\text{-}callee\ callee2\ q\ xs\ x$   
**shows**  $trace\text{-}callee\text{-}eq\ callee1\ callee2\ A\ p\ q$   
**using**  $assms$  **by**( $simp\ add: trace\text{-}callee\text{-}eq\text{-}def$ )

**lemma**  $trace\text{-}callee\text{-}eqD$ :  
**assumes**  $trace\text{-}callee\text{-}eq\ callee1\ callee2\ A\ p\ q$   
**and**  $set\ xs \subseteq A \times UNIV\ x \in A$   
**shows**  $trace\text{-}callee\ callee1\ p\ xs\ x = trace\text{-}callee\ callee2\ q\ xs\ x$   
**using**  $assms$  **by**( $simp\ add: trace\text{-}callee\text{-}eq\text{-}def$ )

**lemma**  $cond\text{-}spmf\text{-}fst\text{-}None$  [ $simp$ ]:  $cond\text{-}spmf\text{-}fst\ (return\text{-}pmf\ None)\ x = return\text{-}pmf\ None$   
**by**( $simp$ )

**lemma**  $trace\text{-}callee\text{-}None$  [ $simp$ ]:  
 $trace\text{-}callee\ callee\ (return\text{-}pmf\ None)\ xs\ x = return\text{-}pmf\ None$   
**by**( $induction\ xs$ )( $auto$ )

**proposition**  $trace'\text{-}eqI\text{-}sim$ :  
**fixes**  $callee1 :: ('a, 'b, 's1)\ callee$  **and**  $callee2 :: ('a, 'b, 's2)\ callee$   
**assumes**  $start: S\ p\ q$   
**and**  $step: \bigwedge p\ q\ a. \llbracket S\ p\ q; a \in A \rrbracket \implies$   
 $bind\text{-}spmf\ p\ (\lambda s. map\text{-}spmf\ fst\ (callee1\ s\ a)) = bind\text{-}spmf\ q\ (\lambda s. map\text{-}spmf\ fst\ (callee2\ s\ a))$   
**and**  $sim: \bigwedge p\ q\ a\ res\ b\ s'. \llbracket S\ p\ q; a \in A; res \in set\text{-}spmf\ q; (b, s') \in set\text{-}spmf\ (callee2\ res\ a) \rrbracket$   
 $\implies S\ (cond\text{-}spmf\text{-}fst\ (bind\text{-}spmf\ p\ (\lambda s. callee1\ s\ a))\ b)$   
 $(cond\text{-}spmf\text{-}fst\ (bind\text{-}spmf\ q\ (\lambda s. callee2\ s\ a))\ b)$   
**shows**  $trace\text{-}callee\text{-}eq\ callee1\ callee2\ A\ p\ q$   
**proof**( $rule\ trace\text{-}callee\text{-}eqI$ )  
**fix**  $xs :: ('a \times 'b)\ list$  **and**  $z$   
**assume**  $xs: set\ xs \subseteq A \times UNIV$  **and**  $z: z \in A$

```

from start show trace-callee callee1 p xs z = trace-callee callee2 q xs z using xs
proof(induction xs arbitrary: p q)
  case Nil
  then show ?case using z by(simp add: step)
next
  case (Cons xy xs)
  obtain x y where xy [simp]: xy = (x, y) by(cases xy)
  have trace-callee callee1 p (xy # xs) z =
    trace-callee callee1 (cond-spmf-fst (bind-spmf p (λs. 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 (λs. 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.prem by (auto intro:
Cons.IH)
  next
  case False
  then have cond-spmf-fst (bind-spmf q (λs. 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.prem False
  have cond-spmf-fst (bind-spmf p (λs. 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 (λres. 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) (λs. 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  $\Rightarrow$  ('a, 'b, 's2) callee  $\Rightarrow$  'a set  
 $\Rightarrow$  's1 spmf  $\Rightarrow$  's2 spmf  $\Rightarrow$  's1 spmf  $\Rightarrow$  's2 spmf  $\Rightarrow$  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  $\subseteq$  A  $\times$  UNIV

**lemma** *trace-callee-closure-start*: *trace-callee-closure* callee1 callee2 A p q p q  
**by**(*simp add: trace-callee-closure.simps exI[where x=[]]*)

**lemma** *trace-callee-closure-step*:

**assumes** *trace-callee-eq* callee1 callee2 A p q  
**and** *trace-callee-closure* callee1 callee2 A p q p' q'  
**and**  $a \in A$   
**shows** *bind-spmf* p' ( $\lambda s. \text{map-spmf fst (callee1 s a)}$ ) = *bind-spmf* q' ( $\lambda s. \text{map-spmf fst (callee2 s a)}$ )  
**proof** –  
**from** *assms(2)* **obtain** xs **where** xs: set xs  $\subseteq$  A  $\times$  UNIV  
**and** p: p' = *trace-callee-aux* callee1 p xs **and** q: q' = *trace-callee-aux* callee2 q xs **by**(*cases*)  
**from** *trace-callee-eqD[OF assms(1) xs assms(3)]* p q **show** ?thesis  
**by**(*simp add: trace-callee-conv-trace-callee-aux*)  
**qed**

**lemma** *trace-callee-closure-sim*:

**assumes** *trace-callee-closure* callee1 callee2 A p q p' q'  
**and**  $a \in A$   
**shows** *trace-callee-closure* callee1 callee2 A p q  
(*cond-spmf-fst (bind-spmf p' ( $\lambda s. \text{callee1 s a}$ )) b*)  
(*cond-spmf-fst (bind-spmf q' ( $\lambda s. \text{callee2 s a}$ )) b*)  
**using** *assms* **proof** (*cases*)  
**case** (1 xs)  
**then show** ?thesis **by**  
(*auto simp add: trace-callee-closure.simps assms trace-callee-aux-append split-def map-spmf-conv-bind-spmf intro!: exI[where x=xs @ [(a, b)]]*)  
**qed**

**proposition** *trace-callee-eq-complete*:

**assumes** *trace-callee-eq* callee1 callee2 A p q  
**obtains** S  
**where** S p q  
**and**  $\bigwedge p q a. \llbracket S p q; a \in A \rrbracket \Longrightarrow$   
*bind-spmf* p ( $\lambda s. \text{map-spmf fst (callee1 s a)}$ ) = *bind-spmf* q ( $\lambda s. \text{map-spmf fst (callee2 s a)}$ )  
**and**  $\bigwedge p q a s b s'. \llbracket S p q; a \in A; s \in \text{set-spmf } q; (b, s') \in \text{set-spmf (callee2 s a)} \rrbracket$   
 $\Longrightarrow S$  (*cond-spmf-fst (bind-spmf p ( $\lambda s. \text{callee1 s a}$ )) b*)  
(*cond-spmf-fst (bind-spmf q ( $\lambda s. \text{callee2 s a}$ )) 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*:  $set\text{-spmf} (cond\text{-spmf}\text{-fst } p \ a) = snd \text{ ` } (set\text{-spmf } p \cap \{a\} \times UNIV)$   
**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** *inv2*: *callee-invariant-on callee2 I2 I*

**and** *WT*:  $\mathcal{I} \vdash g \text{ gpv } \checkmark$

**and** *out*: *outs-gpv I gpv*  $\subseteq A$

**and** *pq*: *lossless-spmf p lossless-spmf q*

**and** *I1*:  $\forall x \in set\text{-spmf } p. I1 \ x$

**and** *I2*:  $\forall y \in set\text{-spmf } q. I2 \ y$

**shows**  $bind\text{-spmf } p (run\text{-gpv } callee1 \ gpv) = bind\text{-spmf } q (run\text{-gpv } callee2 \ gpv)$

**proof** –

**from** *trace-eq* **obtain** *S* **where** *start*:  $S \ p \ q$

**and** *sim*:  $\bigwedge p \ q \ a. \llbracket S \ p \ q; a \in A \rrbracket \implies$

$bind\text{-spmf } p (\lambda s. map\text{-spmf } fst (callee1 \ s \ a)) = bind\text{-spmf } q (\lambda s. map\text{-spmf } fst (callee2 \ s \ a))$

**and** *S*:  $\bigwedge p \ q \ a \ s \ b \ s'. \llbracket S \ p \ q; a \in A; s \in set\text{-spmf } q; (b, s') \in set\text{-spmf } (callee2 \ s \ a) \rrbracket$

$\implies S (cond\text{-spmf}\text{-fst } (bind\text{-spmf } p (\lambda s. callee1 \ s \ a)) \ b)$

$(cond\text{-spmf}\text{-fst } (bind\text{-spmf } q (\lambda s. callee2 \ s \ a)) \ b)$

**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**  $\langle S \ p \ q \rangle$  *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*]:  $generat \in set\text{-spmf } (the\text{-gpv } gpv) \implies$

$p \gg= (\lambda xa. map\text{-spmf } fst (case \ generat \ of$

*Pure*  $x \Rightarrow return\text{-spmf } (x, xa)$

$| IO \ out \ c \Rightarrow callee1 \ xa \ out \gg= (\lambda(x, y). exec\text{-gpv}' (c \ x) \ y))) =$

$q \gg= (\lambda xa. map\text{-spmf } fst (case \ generat \ of$

*Pure*  $x \Rightarrow return\text{-spmf } (x, xa)$

$| IO \ out \ c \Rightarrow callee2 \ xa \ out \gg= (\lambda(x, y). g (c \ x) \ y)))$  **for** *generat*

**proof** (*cases generat, goal-cases*)

**case** (*2 out rpv*)

**have** [*simp*]:  $IO \ out \ rpv \in set\text{-spmf } (the\text{-gpv } gpv) \implies generat = IO \ out \ rpv$

$\implies$

$map\text{-spmf } fst (p \gg= (\lambda xa. callee1 \ xa \ out)) = map\text{-spmf } fst (q \gg= (\lambda xa. callee2 \ xa \ out))$



```

unfolding map-bind-spmf o-def
by (rule sim) (use step.premis in ⟨auto intro: outs-gpv.IO⟩)

have[simp]: [[IO out rpv ∈ set-spmf (the-gpv gpv); generat = IO out rpv; x ∈
set-spmf q; (a, b) ∈ set-spmf (callee2 x out)]] ⇒
  cond-spmf-fst (p ≫ (λxa. callee1 xa out)) a ≫ (λx. map-spmf fst (exec-gpv'
(rpv a) x)) =
  cond-spmf-fst (q ≫ (λxa. callee2 xa out)) a ≫ (λx. map-spmf fst (g (rpv
a) x)) for a b x
proof (rule step.IH, goal-cases)
  case 1 then show ?case using step.premis by(auto intro!: S intro:
outs-gpv.IO)
  next
  case 2
    then show ?case using step.premis by(force intro: outs-gpv.Cont dest:
WT-calleeD[OF I2.WT-callee] WT-gpvD[OF step.premis(5)])
  next
  case 3
    then show ?case using sim[OF ⟨S p q⟩, of out] step.premis(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.premis by(auto 4 3 dest: WT-calleeD[OF
I2.WT-callee] intro: WT-gpvD)
  next
  case 6
    then show ?case using step.premis by(auto simp add: bind-UNION im-
age-Union set-spmf-cond-spmf-fst intro: I1.callee-invariant WT-gpvD)
  next
  case 7
    then show ?case using step.premis by(auto simp add: bind-UNION im-
age-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.premis)

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*:  $A \vdash_C \text{callee1}(s1) \approx \text{callee2}(s2)$

**and** *inv1*: *callee-invariant-on callee1 I1 I*

**and** *inv2*: *callee-invariant-on callee2 I2 I*

**and** *WT*:  $\mathcal{I} \vdash g \text{ gpv } \checkmark$

**and** *out*: *outs-gpv I gpv*  $\subseteq A$

**and** *I1*: *I1 s1*

**and** *I2*: *I2 s2*

**shows** *run-gpv callee1 gpv s1* = *run-gpv callee2 gpv s2*

**using** *trace-callee-eq-run-gpv[OF assms(1-5)] assms(6-)* **by** *simp*

**abbreviation** *trace-eq* :: 'a set  $\Rightarrow$  ('a, 'b) resource spmf  $\Rightarrow$  ('a, 'b) resource spmf

$\Rightarrow$  bool **where**

*trace-eq*  $\equiv$  *trace-callee-eq run-resource run-resource*

**abbreviation** *trace-eq'* :: 'a set  $\Rightarrow$  ('a, 'b) resource  $\Rightarrow$  ('a, 'b) resource  $\Rightarrow$  bool

( $\langle (-) \vdash_R / (-) / \approx (-) \rangle$  [90, 90, 90] 91) **where**

$A \vdash_R \text{res} \approx \text{res}' \equiv \text{trace-eq } A \text{ (return-spmf res) (return-spmf res')}$

**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]*:

$A \vdash_R \text{resource-of-oracle callee1 s1} \approx \text{resource-of-oracle callee2 s2} =$

$A \vdash_C \text{callee1}(s1) \approx \text{callee2}(s2)$

**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*  $\Rightarrow$  ('a, 'b) *resource*  $\Rightarrow$  bool *spmf* **where**  
*connect* d res = run-gpv run-resource d res

**definition** *absorb* :: ('a, 'b) *distinguisher*  $\Rightarrow$  ('a, 'b, 'out, 'in) *converter*  $\Rightarrow$  ('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 =  $\lambda$ res (conv', res'). res = attach conv' res'  
**have\***: *rel-spmf* (*rel-prod* (=) ?R) (*exec-gpv* run-resource d (attach conv res))  
 (*exec-gpv* ( $\lambda$ p y. *map-spmf* ( $\lambda$ 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= $\lambda$ res (conv', res'). res = attach conv' res'  
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-refl* *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 =  $\lambda$ conv (conv', conv''). conv = *comp-converter* conv' conv''  
**have\***: *rel-gpv* (*rel-prod* (=) ?R) (=) (inline run-converter d (*comp-converter*  
conv conv'))  
 (inline ( $\lambda$ p c2. *map-gpv* ( $\lambda$ 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*:  $A \vdash_R \text{res1} \approx \text{res2}$   
**and** *WT*:  $\mathcal{I} \vdash g \ d \ \checkmark$   
**and** *out*:  $\text{outs-gpv } \mathcal{I} \ d \subseteq A$   
**and** *WT1*:  $\mathcal{I} \vdash_{\text{res}} \text{res1} \ \checkmark$   
**and** *WT2*:  $\mathcal{I} \vdash_{\text{res}} \text{res2} \ \checkmark$   
**shows**  $\text{connect } d \ \text{res1} = \text{connect } d \ \text{res2}$   
**unfolding** *connect-def* **using** *trace-eq callee-invariant-run-resource callee-invariant-run-resource*  
*WT out WT1 WT2*  
**by**(*rule trace-callee-eq'-run-gpv*)

**lemma** *distinguish-trace-eq*:  
**assumes** *distinguish*:  $\bigwedge \text{distinguisher}. \mathcal{I} \vdash g \ \text{distinguisher} \ \checkmark \implies \text{connect } \text{distinguish} \ \text{res} = \text{connect } \text{distinguish} \ \text{res}'$   
**and** *WT1*:  $\mathcal{I} \vdash_{\text{res}} \text{res1} \ \checkmark$   
**and** *WT2*:  $\mathcal{I} \vdash_{\text{res}} \text{res2} \ \checkmark$   
**shows**  $\text{outs-}\mathcal{I} \ \mathcal{I} \vdash_R \text{res} \approx \text{res}'$   
**proof**(*rule ccontr*)  
**let**  $?P = \lambda xs. \exists x. \text{set } xs \subseteq \text{outs-}\mathcal{I} \ \mathcal{I} \times \text{UNIV} \wedge x \in \text{outs-}\mathcal{I} \ \mathcal{I} \wedge \text{trace-callee run-resource (return-spmf res) } xs \ x \neq \text{trace-callee run-resource (return-spmf res')} \ xs \ x$   
**assume**  $\neg ?thesis$   
**then have**  $\exists x ?P$  **unfolding** *trace-callee-eq-def* **by** *auto*  
**with** *wf-strict-prefix[unfolded wfp-eq-minimal, THEN spec, of Collect ?P]*  
**obtain**  $xs \ x$  **where**  $\text{set } xs \subseteq \text{outs-}\mathcal{I} \ \mathcal{I} \times \text{UNIV}$   
**and**  $x: x \in \text{outs-}\mathcal{I} \ \mathcal{I}$   
**and** *neq*:  $\text{trace-callee run-resource (return-spmf res) } xs \ x \neq \text{trace-callee run-resource (return-spmf res')} \ xs \ x$   
**and** *shortest*:  $\bigwedge xs' \ x. \llbracket \text{strict-prefix } xs' \ xs; \text{set } xs' \subseteq \text{outs-}\mathcal{I} \ \mathcal{I} \times \text{UNIV}; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$   
 $\implies \text{trace-callee run-resource (return-spmf res) } xs' \ x = \text{trace-callee run-resource (return-spmf res')} \ xs' \ x$   
**by**(*auto*)  
**have** *shortest*:  $\bigwedge xs' \ x. \llbracket \text{strict-prefix } xs' \ xs; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$   
 $\implies \text{trace-callee run-resource (return-spmf res) } xs' \ x = \text{trace-callee run-resource (return-spmf res')} \ xs' \ x$   
**by**(*rule shortest*)(*use xs in <auto 4 3 dest: strict-prefix-setD>*)  
  
**define** *p* **where**  $p \equiv \text{return-spmf res}$   
**define** *q* **where**  $q \equiv \text{return-spmf res}'$   
**have** *p*: *lossless-spmf* *p* **and** *q*: *lossless-spmf* *q* **by**(*simp-all add: p-def q-def*)  
**from** *neq* **obtain** *y* **where**  $y: \text{spm}f \ (\text{trace-callee run-resource } p \ xs \ x) \ y \neq \text{spm}f \ (\text{trace-callee run-resource } q \ xs \ x) \ y$   
**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 (λy'. Done (y = y'))) (λ(x, y) xs rec. Pause x
(λinput. if input = y then rec else Done False))
  have d-simps [simp]:
    d [] = Pause x (λy'. Done (y = y'))
    d ((a, b) # xs) = Pause a (λinput. if input = b then d xs else Done False) for
a b xs
  by(simp-all add: d-def fun-eq-iff)
have WT-d:  $\mathcal{I} \vdash g$  d xs ✓ 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 ≫ (λs. run-resource s x')) y'
    and ?q = cond-spmf-fst (q ≫ (λs. run-resource s x')) y'
  from Cons.prem1
  have spmf ((map-spmf fst (p ≫ (λy. run-resource y x')) ≫ (λx. map-spmf
(Pair x) (cond-spmf-fst (p ≫ (λy. 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 ≫ (λy. run-resource y x')) ≫ (λx. map-spmf (Pair
x) (cond-spmf-fst (q ≫ (λy. 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=λx. run-gpv - x -] cong del: if-weak-cong)
  hence spmf ((p ≫ (λs. map-spmf fst (run-resource s x'))) ≫
    (λa. if a = y' then cond-spmf-fst (p ≫ (λy. run-resource y x')) a ≫
connect (d xs)
      else bind-spmf (cond-spmf-fst (p ≫ (λy. run-resource y x'))
a) (λ-. return-spmf False))) True =
    spmf ((q ≫ (λs. map-spmf fst (run-resource s x'))) ≫
    (λa. if a = y' then cond-spmf-fst (q ≫ (λy. run-resource y x')) a ≫
connect (d xs)
      else bind-spmf (cond-spmf-fst (q ≫ (λy. run-resource y x'))
a) (λ-. return-spmf False))) True

```

**by**(rule box-equals; use nothing in  $\langle$ rule arg-cong2[where  $f = \text{spmf}$ ] $\rangle$ )  
(auto simp add: map-bind-spmf bind-map-spmf o-def split-def intro!: bind-spmf-cong)  
**hence** LINT  $a \mid \text{measure-spmf } (p \gg (\lambda s. \text{map-spmf fst } (\text{run-resource } s \ x')))$ . (if  
 $a = y'$  then  $\text{spmf } (?p \gg \text{connect } (d \ xs)) \text{ True else } 0) =$   
LINT  $a \mid \text{measure-spmf } (q \gg (\lambda s. \text{map-spmf fst } (\text{run-resource } s \ x')))$ . (if  $a =$   
 $y'$  then  $\text{spmf } (?q \gg \text{connect } (d \ xs)) \text{ True else } 0)$   
**by**(rule box-equals; use nothing in  $\langle$ subst spmf-bind $\rangle$ )  
(auto intro!: Bochner-Integration.integral-cong simp add: bind-spmf-const  
spmf-scale-spmf)  
**hence** LINT  $a \mid \text{measure-spmf } (p \gg (\lambda s. \text{map-spmf fst } (\text{run-resource } s \ x')))$ .  
indicator  $\{y'\} a * \text{spmf } (?p \gg \text{connect } (d \ xs)) \text{ True} =$   
LINT  $a \mid \text{measure-spmf } (q \gg (\lambda s. \text{map-spmf fst } (\text{run-resource } s \ x')))$ . indicator  
 $\{y'\} a * \text{spmf } (?q \gg \text{connect } (d \ xs)) \text{ True}$   
**by**(rule box-equals; use nothing in  $\langle$ rule Bochner-Integration.integral-cong $\rangle$ )  
auto  
**hence**  $\text{spmf } (p \gg (\lambda s. \text{map-spmf fst } (\text{run-resource } s \ x')) \ y' * \text{spmf } (?p \gg$   
 $\text{connect } (d \ xs)) \text{ True} =$   
 $\text{spmf } (q \gg (\lambda s. \text{map-spmf fst } (\text{run-resource } s \ x')) \ y' * \text{spmf } (?q \gg \text{connect}$   
 $(d \ xs)) \text{ True}$   
**by**(simp add: spmf-conv-measure-spmf)  
**moreover from** Cons.prem3[ $of \ [] \ x'$ ] Cons.prem4  
**have**  $\text{spmf } (p \gg (\lambda s. \text{map-spmf fst } (\text{run-resource } s \ x')) \ y' = \text{spmf } (q \gg (\lambda s.$   
 $\text{map-spmf fst } (\text{run-resource } s \ x')) \ y'$   
**by**(simp)  
**ultimately have**  $\text{spmf } (?p \gg \text{connect } (d \ xs)) \text{ True} = \text{spmf } (?q \gg \text{connect}$   
 $(d \ xs)) \text{ True}$   
**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**  $\text{spmf } (\text{trace-callee run-resource } ?p \ xs \ x) \ y \neq \text{spmf } (\text{trace-callee run-resource}$   
 $?q \ xs \ x) \ y$   
**using** Cons.prem3 **by** simp  
**moreover**  
**have**  $\text{spmf } (\text{trace-callee run-resource } ?p \ ys \ x) \ y = \text{spmf } (\text{trace-callee run-resource}$   
 $?q \ ys \ x) \ y$   
**if**  $ys$ : strict-prefix  $ys \ xs$  **and**  $x$ :  $x \in \text{outs-}\mathcal{I} \ \mathcal{I}$  **for**  $ys \ x \ y$   
**using** Cons.prem3[ $of \ xy \ \# \ ys \ x \ y$ ]  $ys \ x$  **by** simp  
**moreover have**  $\text{set } xs \subseteq \text{outs-}\mathcal{I} \ \mathcal{I} \times \text{UNIV}$  **using** Cons.prem4 **by** auto  
**ultimately show**  $?case$  **by**(rule Cons.IH)  
**qed**  
**qed**

**lemma** connect-eq-resource-cong:

**assumes**  $\mathcal{I} \vdash g$  distinguisher  $\checkmark$

**and**  $\text{outs-}\mathcal{I} \ \mathcal{I} \vdash_R \text{res} \sim \text{res}'$

**and**  $\mathcal{I} \vdash \text{res} \text{res}'$

**shows**  $\text{connect distinguisher res} = \text{connect distinguisher res}'$

**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*]:

$\llbracket \mathcal{I}' \vdash_g \text{gpv } \checkmark; \mathcal{I}', \mathcal{I} \vdash_C \text{conv } \checkmark \rrbracket \implies \mathcal{I} \vdash_g \text{absorb gpv conv } \checkmark$   
**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*]:  $\mathcal{I}' \vdash_g \text{gpv } \checkmark$   $\mathcal{I}', \mathcal{I} \vdash_C \text{conv } \checkmark$   
**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**  $\langle \parallel \rangle$  50)

**no-notation** *plus-oracle* (**infix**  $\langle \oplus_O \rangle$  500)

**notation** *Resource* ( $\langle \S R \S \rangle$ )

**notation** *Converter* ( $\langle \S C \S \rangle$ )

**alias** *RES* = *resource-of-oracle*

**alias** *CNV* = *converter-of-callee*

**alias** *id-intercept* = *id-oracle*

**notation** *id-oracle* ( $\langle 1_I \rangle$ )

**notation** *plus-oracle* (**infixr**  $\langle \oplus_O \rangle$  504)

**notation** *parallel-oracle* (**infixr**  $\langle \dagger_O \rangle$  504)

**notation** *plus-intercept* (**infixr**  $\langle \oplus_I \rangle$  504)

**notation** *parallel-intercept* (**infixr**  $\langle \dagger_I \rangle$  504)

**notation** *parallel-resource* (**infixr**  $\langle \parallel \rangle$  501)

**notation** *parallel-converter* (**infixr**  $\langle |_{\alpha} \rangle$  501)

**notation** *parallel-converter2* (**infixr**  $\langle |_{=} \rangle$  501)

**notation** *comp-converter* (**infixr**  $\langle \odot \rangle$  502)

**notation** *fail-converter* ( $\langle \perp_C \rangle$ )

**notation** *id-converter* ( $\langle 1_C \rangle$ )

**notation** *attach* (**infixr**  $\langle \triangleright \rangle$  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<sub>C</sub>* ::  $('a + 'b, 'c + 'd, 'b + 'a, 'd + 'c)$  *converter* **where**

*swap<sub>C</sub>* = *map-converter* *swap-sum* *swap-sum* *id* *id*  $1_C$

**definition** *rassoc<sub>C</sub>* ::  $('a + ('b + 'c), 'd + ('e + 'f), ('a + 'b) + 'c, ('d + 'e) + 'f)$  *converter* **where**

*rassoc<sub>C</sub>* = *map-converter* *lsumr* *rsuml* *id* *id*  $1_C$

**definition** *lassocr<sub>C</sub>* ::  $((('a + 'b) + 'c, ('d + 'e) + 'f, 'a + ('b + 'c), 'd + ('e + 'f))$  *converter* **where**

*lassocr<sub>C</sub>* = *map-converter* *rsuml* *lsumr* *id* *id*  $1_C$

**definition** *swap-rassocl* **where** *swap-rassocl*  $\equiv$  *lassocr<sub>C</sub>*  $\odot$  ( $1_C$   $|_{=} swap_C$ )  $\odot$  *rassocl<sub>C</sub>*

**definition** *swap-lassocr* **where** *swap-lassocr*  $\equiv$  *rassocl<sub>C</sub>*  $\odot$  ( $swap_C$   $|_{=} 1_C$ )  $\odot$  *lassocr<sub>C</sub>*

**definition** *parallel-wiring* ::  $((('a + 'b) + ('e + 'f), ('c + 'd) + ('g + 'h), ('a + 'e) + ('b + 'f), ('c + 'g) + ('d + 'h))$  *converter* **where**

*parallel-wiring* = *lassocr<sub>C</sub>*  $\odot$  ( $1_C$   $|_{=} swap\_lassocr$ )  $\odot$  *rassocl<sub>C</sub>*

**lemma** *WT-lassocr<sub>C</sub>* [*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<sub>C</sub>*  $\checkmark$

**by** (*coinduction*)(*auto simp add: lassocr<sub>C</sub>-def*)

**lemma** *WT-rassocl<sub>C</sub>* [*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<sub>C</sub>*  $\checkmark$

**by** (*coinduction*)(*auto simp add: rassocl<sub>C</sub>-def*)



**lemma** *WT-swap<sub>C</sub>* [*WT-intro*]:  $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}1 \vdash_C \text{swap}_C \checkmark$   
**by**(*coinduction*)(*auto simp add: swap<sub>C</sub>-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*  $\checkmark$   
**unfolding** *swap-lassocr-def*  
**by**(*rule WT-converter-comp WT-lassocr<sub>C</sub> WT-rasso<sub>C</sub> WT-converter-parallel-converter<sup>2</sup>*  
*WT-converter-id WT-swap<sub>C</sub>*)**+**

**lemma** *WT-swap-rasso<sub>C</sub>* [*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-rasso<sub>C</sub>*  $\checkmark$   
**unfolding** *swap-rasso<sub>C</sub>-def*  
**by**(*rule WT-converter-comp WT-lassocr<sub>C</sub> WT-rasso<sub>C</sub> WT-converter-parallel-converter<sup>2</sup>*  
*WT-converter-id WT-swap<sub>C</sub>*)**+**

**lemma** *WT-parallel-wiring* [*WT-intro*]:  
 $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} (\mathcal{I}3 \oplus_{\mathcal{I}} \mathcal{I}4), (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}3) \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}4) \vdash_C \text{parallel-wiring} \checkmark$   
**unfolding** *parallel-wiring-def*  
**by**(*rule WT-converter-comp WT-lassocr<sub>C</sub> WT-rasso<sub>C</sub> WT-converter-parallel-converter<sup>2</sup>*  
*WT-converter-id WT-swap-lassocr*)**+**

**lemma** *map-swap-sum-plus-oracle: includes lifting-syntax shows*  
 $(\text{id} \text{ ----} > \text{swap-sum} \text{ ----} > \text{map-spmf} (\text{map-prod swap-sum id})) (\text{oracle1} \oplus_O$   
 $\text{oracle2}) =$   
 $(\text{oracle2} \oplus_O \text{oracle1})$   
**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*]:  $\text{map-}\mathcal{I} \text{ rsuml lsumr } (\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)) =$   
 $((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3)$   
**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*]:  $\text{map-}\mathcal{I} \text{ lsumr rsuml } ((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3) =$   
 $(\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3))$   
**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*]:  $\text{map-}\mathcal{I} \text{ swap-sum swap-sum } (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) = \mathcal{I}2$   
 $\oplus_{\mathcal{I}} \mathcal{I}1$   
**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*  $\checkmark$   
**unfolding** *parallel-resource1-wiring-def* **by**(rule *WT-swap-lassocr*)

**lemma** *plossless-rassocl<sub>C</sub>* [*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<sub>C</sub>*  
**by** *coinduction* (*auto simp add: rassocl<sub>C</sub>-def*)

**lemma** *plossless-lassocr<sub>C</sub>* [*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<sub>C</sub>*  
**by** *coinduction* (*auto simp add: lassocr<sub>C</sub>-def*)

**lemma** *plossless-swap<sub>C</sub>* [*plossless-intro*]: *plossless-converter*  $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2)$   $(\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}1)$  *swap<sub>C</sub>*  
**by** *coinduction* (*auto simp add: swap<sub>C</sub>-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*  $((conv1 \models conv2) \models conv3) =$   
*map-converter* *rsuml lsumr id id*  $(conv1 \models conv2 \models conv3)$   
**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<sub>C</sub>*:  $((conv1 \models conv2) \models conv3) \odot$  *lassocr<sub>C</sub>* = *lassocr<sub>C</sub>*  $\odot$   
 $(conv1 \models conv2 \models conv3)$   
**unfolding** *lassocr<sub>C</sub>-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<sub>C</sub>'* = *comp-converter-egs[OF comp-lassocr<sub>C</sub>]*

**lemma** *lsumr-rsuml-parallel-converter2*:  
*map-converter* *id id lsumr rsuml*  $(conv1 \models (conv2 \models conv3)) =$

$map\text{-}converter\ lsumr\ rsuml\ id\ id\ ((conv1 \models conv2) \models conv3)$   
**by**(*coinduction arbitrary: conv1 conv2 conv3, clarsimp split!: sum.split simp add:*  
*rel-fun-def map-gpv-conv-map-gpv'[symmetric]*)  
 $((subst\ left\text{-}gpv\text{-}map[\mathbf{where}\ h=id] \mid subst\ right\text{-}gpv\text{-}map[\mathbf{where}\ h=id])+$   
 $,\ simp\ add:gpv.map\text{-}comp\ sum.map\text{-}id0\ o\text{-}def\ prod.map\text{-}comp\ id\text{-}def[symmetric]$   
 $,\ subst\ map\text{-}gpv'\text{-}map\text{-}gpv\text{-}swap,\ (subst\ lsumr\text{-}rsuml\text{-}left\text{-}gpv \mid subst\ lsumr\text{-}rsuml\text{-}right\text{-}gpv\text{-}left\text{-}gpv$   
 $\mid subst\ lsumr\text{-}rsuml\text{-}right\text{-}gpv\text{-}right\text{-}gpv)$   
 $,\ auto\ 4\ 4\ intro!: gpv.rel\text{-}refl\text{-}strong\ simp\ add: gpv.rel\text{-}map)+$

**lemma**  $comp\text{-}rassocl_C$ :  
 $(conv1 \models conv2 \models conv3) \odot rassocl_C = rassocl_C \odot ((conv1 \models conv2) \models conv3)$   
**unfolding**  $rassocl_C\text{-}def$   
**by**(*subst comp-converter-map-converter2*)  
 $(simp\ add: comp\text{-}converter\text{-}id\text{-}right\ comp\text{-}converter\text{-}map1\text{-}out\ comp\text{-}converter\text{-}id\text{-}left$   
 $lsumr\text{-}rsuml\text{-}parallel\text{-}converter2)$

**lemmas**  $comp\text{-}rassocl_C' = comp\text{-}converter\text{-}eqs[OF\ comp\text{-}rassocl_C]$

**lemma**  $swap\text{-}sum\text{-}right\text{-}gpv$ :  
 $map\text{-}gpv'\ id\ swap\text{-}sum\ swap\text{-}sum\ (right\text{-}gpv\ gpv) = left\text{-}gpv\ gpv$   
**by**(*coinduction arbitrary: gpv*)  
 $(auto\ 4\ 3\ simp\ add: spmf\text{-}rel\text{-}map\ generat.rel\text{-}map\ intro!: rel\text{-}spmf\text{-}reflI\ rel\text{-}generat\text{-}reflI$   
 $rel\text{-}funI\ split: sum.split\ intro: exI[\mathbf{where}\ x=Fail])$

**lemma**  $swap\text{-}sum\text{-}left\text{-}gpv$ :  
 $map\text{-}gpv'\ id\ swap\text{-}sum\ swap\text{-}sum\ (left\text{-}gpv\ gpv) = right\text{-}gpv\ gpv$   
**by**(*coinduction arbitrary: gpv*)  
 $(auto\ 4\ 3\ simp\ add: spmf\text{-}rel\text{-}map\ generat.rel\text{-}map\ intro!: rel\text{-}spmf\text{-}reflI\ rel\text{-}generat\text{-}reflI$   
 $rel\text{-}funI\ split: sum.split\ intro: exI[\mathbf{where}\ x=Fail])$

**lemma**  $swap\text{-}sum\text{-}parallel\text{-}converter2$ :  
 $map\text{-}converter\ id\ id\ swap\text{-}sum\ swap\text{-}sum\ (conv1 \models conv2) =$   
 $map\text{-}converter\ swap\text{-}sum\ swap\text{-}sum\ id\ id\ (conv2 \models conv1)$   
**by**(*coinduction arbitrary: conv1 conv2, clarsimp simp add: rel-fun-def map-gpv-conv-map-gpv'[symmetric]*  
 $split!: sum.split)$   
 $(subst\ map\text{-}gpv'\text{-}map\text{-}gpv\text{-}swap,\ (subst\ swap\text{-}sum\text{-}right\text{-}gpv \mid subst\ swap\text{-}sum\text{-}left\text{-}gpv),$   
 $auto\ 4\ 4\ intro!: gpv.rel\text{-}refl\text{-}strong\ simp\ add: gpv.rel\text{-}map)+$

**lemma**  $comp\text{-}swap_C$ :  $(conv1 \models conv2) \odot swap_C = swap_C \odot (conv2 \models conv1)$   
**unfolding**  $swap_C\text{-}def$   
**by**(*subst comp-converter-map-converter2*)  
 $(simp\ add: comp\text{-}converter\text{-}id\text{-}right\ comp\text{-}converter\text{-}map1\text{-}out\ comp\text{-}converter\text{-}id\text{-}left$   
 $swap\text{-}sum\text{-}parallel\text{-}converter2)$

**lemmas**  $comp\text{-}swap_C' = comp\text{-}converter\text{-}eqs[OF\ comp\text{-}swap_C]$

**lemma**  $comp\text{-}swap\text{-}lassocr$ :  $(conv1 \models conv2 \models conv3) \odot swap\text{-}lassocr = swap\text{-}lassocr$   
 $\odot (conv2 \models conv1 \models conv3)$

**unfolding** *swap-lassocr-def comp-rassocl<sub>C</sub>' comp-converter-assoc comp-converter-parallel2' comp-swap<sub>C</sub>' comp-converter-id-right*  
**by**(*subst (9) comp-converter-id-left[symmetric], subst comp-converter-parallel2[symmetric]*)  
*(simp add: comp-converter-assoc comp-lassocr<sub>C</sub>)*

**lemmas** *comp-swap-lassocr' = comp-converter-eqs[OF comp-swap-lassocr]*

**lemma** *comp-parallel-wiring:*

*((C1 |= C2) |= (C3 |= C4)) ∘ parallel-wiring = parallel-wiring ∘ ((C1 |= C3) |= (C2 |= C4))*

**unfolding** *parallel-wiring-def comp-lassocr<sub>C</sub>' 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<sub>C</sub>)*

**lemmas** *comp-parallel-wiring' = comp-converter-eqs[OF comp-parallel-wiring]*

**lemma** *attach-converter-of-resource-conv-parallel-resource:*

*converter-of-resource res |<sub>∞</sub> 1<sub>C</sub> ▷ res' = res || 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-refl)*

**lemma** *attach-converter-of-resource-conv-parallel-resource2:*

*1<sub>C</sub> |<sub>∞</sub> converter-of-resource res ▷ res' = res' || 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-refl)*

**lemma** *plossless-parallel-wiring [plossless-intro]:*

*plossless-converter ((I1 ⊕<sub>I</sub> I2) ⊕<sub>I</sub> (I3 ⊕<sub>I</sub> I4)) ((I1 ⊕<sub>I</sub> I3) ⊕<sub>I</sub> (I2 ⊕<sub>I</sub> I4)) parallel-wiring*

**unfolding** *parallel-wiring-def by(rule plossless-intro WT-intro)+*

**lemma** *run-converter-lassocr [simp]:*

*run-converter lassocr<sub>C</sub> x = Pause (rsuml x) (λx. Done (lsumr x, lassocr<sub>C</sub>))*

**by**(*simp add: lassocr<sub>C</sub>-def o-def*)

**lemma** *run-converter-rassocl [simp]:*

*run-converter rassocl<sub>C</sub> x = Pause (lsumr x) (λx. Done (rsuml x, rassocl<sub>C</sub>))*

**by**(*simp add: rassocl<sub>C</sub>-def o-def*)

**lemma** *run-converter-swap [simp]: run-converter swap<sub>C</sub> x = Pause (swap-sum x) (λx. Done (swap-sum x, swap<sub>C</sub>))*

**by**(*simp add: swap<sub>C</sub>-def o-def*)

**definition** *lassocr-swap-sum* **where**  $lassocr\text{-}swap\text{-}sum = rsuml \circ map\text{-}sum\ swap\text{-}sum\ id \circ lsumr$

**lemma** *run-converter-swap-lassocr* [*simp*]:

$run\text{-}converter\ swap\text{-}lassocr\ x = Pause\ (lassocr\text{-}swap\text{-}sum\ x)\ (\ case\ lsumr\ x\ of\ Inl\ - \Rightarrow (\lambda y. case\ lsumr\ y\ of\ Inl\ - \Rightarrow Done\ (lassocr\text{-}swap\text{-}sum\ y,\ swap\text{-}lassocr)\ | - \Rightarrow Fail)\ | Inr\ - \Rightarrow (\lambda y. case\ lsumr\ y\ of\ Inl\ - \Rightarrow Fail\ | Inr\ - \Rightarrow Done\ (lassocr\text{-}swap\text{-}sum\ y,\ swap\text{-}lassocr)))$   
**by**( $subst\ sum.\ case\text{-}distrib$ [**where**  $h = \lambda x. inline\ -\ x$ ] |  $simp\ add: bind\text{-}rpv\text{-}def\ inline\text{-}map\text{-}gpv\ split\text{-}def\ map\text{-}gpv\text{-}conv\text{-}bind$ [*symmetric*]  $swap\text{-}lassocr\text{-}def\ o\text{-}def\ cong\ del: sum.\ case\text{-}cong$ ) +  $(cases\ x\ rule: lsumr.\ cases,\ simp\text{-}all\ add: o\text{-}def\ lassocr\text{-}swap\text{-}sum\text{-}def\ gpv.\ map\text{-}comp\ fun\text{-}eq\text{-}iff\ cong: sum.\ case\text{-}cong\ split: sum.\ split)$

**definition** *parallel-sum-wiring* **where**  $parallel\text{-}sum\text{-}wiring = lsumr \circ map\text{-}sum\ id\ lassocr\text{-}swap\text{-}sum \circ rsuml$

**lemma** *run-converter-parallel-wiring*:

$run\text{-}converter\ parallel\text{-}wiring\ x = Pause\ (parallel\text{-}sum\text{-}wiring\ x)\ (\ case\ rsuml\ x\ of\ Inl\ - \Rightarrow (\lambda y. case\ rsuml\ y\ of\ Inl\ - \Rightarrow Done\ (parallel\text{-}sum\text{-}wiring\ y,\ parallel\text{-}wiring)\ | - \Rightarrow Fail)\ | Inr\ x \Rightarrow (case\ lsumr\ x\ of\ Inl\ - \Rightarrow (\lambda y. case\ rsuml\ y\ of\ Inl\ - \Rightarrow Fail\ | Inr\ x \Rightarrow (case\ lsumr\ x\ of\ Inl\ - \Rightarrow Done\ (parallel\text{-}sum\text{-}wiring\ y,\ parallel\text{-}wiring)\ | Inr\ - \Rightarrow Fail)))\ | Inr\ - \Rightarrow (\lambda y. case\ rsuml\ y\ of\ Inl\ - \Rightarrow Fail\ | Inr\ x \Rightarrow (case\ lsumr\ x\ of\ Inl\ - \Rightarrow Fail\ | Inr\ - \Rightarrow Done\ (parallel\text{-}sum\text{-}wiring\ y,\ parallel\text{-}wiring))))))$   
**by**( $simp\ add: parallel\text{-}wiring\text{-}def\ o\text{-}def\ cong\ del: sum.\ case\text{-}cong\ add: split\text{-}def\ map\text{-}gpv\text{-}conv\text{-}bind$ [*symmetric*] ( $subst\ sum.\ case\text{-}distrib$ [**where**  $h = \lambda x. right\text{-}rpv\ x$ ] |  $subst\ sum.\ case\text{-}distrib$ [**where**  $h = \lambda x. inline\ -\ x$ ] |  $subst\ sum.\ case\text{-}distrib$ [**where**  $h = right\text{-}gpv$ ] |  $(auto\ simp\ add: inline\text{-}map\text{-}gpv\ bind\text{-}rpv\text{-}def\ gpv.\ map\text{-}comp\ fun\text{-}eq\text{-}iff\ parallel\text{-}sum\text{-}wiring\text{-}def\ parallel\text{-}wiring\text{-}def$ [*symmetric*]  $sum.\ case\text{-}distrib\ o\text{-}def\ intro: sym\ cong\ del: sum.\ case\text{-}cong\ split!: sum.\ split)$ ) +

**lemma** *bound-lassocr<sub>C</sub>* [*interaction-bound*]: *interaction-any-bounded-converter lassocr<sub>C</sub> 1*

**unfolding** *lassocr<sub>C</sub>-def* **by** *interaction-bound-converter simp*

**lemma** *bound-rassocl<sub>C</sub>* [*interaction-bound*]: *interaction-any-bounded-converter rassocl<sub>C</sub> 1*

**unfolding** *rassocl<sub>C</sub>-def* **by** *interaction-bound-converter simp*

**lemma** *bound-swap<sub>C</sub>* [*interaction-bound*]: *interaction-any-bounded-converter swap<sub>C</sub>*

1

**unfolding** *swap<sub>C</sub>-def* **by** *interaction-bound-converter simp*

**lemma** *bound-swap-rassoel* [*interaction-bound*]: *interaction-any-bounded-converter swap-rassoel 1*

**unfolding** *swap-rassoel-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 ⇒ 'c*) × (*'d ⇒ 'b*)

**inductive** *wiring* :: (*'a, 'b*) *I* ⇒ (*'c, 'd*) *I* ⇒ (*'a, 'b, 'c, 'd*) *converter* ⇒ (*'a, 'b, 'c, 'd*) *wiring* ⇒ *bool*

**for** *I I' cnv*

**where**

*wiring*:

*wiring I I' cnv (f, g)* **if**

*I, I' ⊢<sub>C</sub> cnv ~ map-converter id id f g 1<sub>C</sub>*

*I, I' ⊢<sub>C</sub> cnv √*

**lemmas** *wiringI* = *wiring*

**hide-fact** *wiring*

**lemma** *wiringD*:

**assumes** *wiring I I' cnv (f, g)*

**shows** *wiringD-eq*: *I, I' ⊢<sub>C</sub> cnv ~ map-converter id id f g 1<sub>C</sub>*

**and** *wiringD-WT*: *I, I' ⊢<sub>C</sub> cnv √*

**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* = ( $\lambda(f, g). \text{map-fun id (map-fun f (map-spmf (map-prod g id)))}$ )

**lemma** *apply-wiring-simps*: *apply-wiring (f, g) = map-fun id (map-fun f (map-spmf (map-prod g id)))*

**by**(*simp add: apply-wiring-def*)

**lemma** *attach-wiring-resource-of-oracle*:

**assumes** *wiring*: *wiring*  $\mathcal{I}1$   $\mathcal{I}2$  *conv fg*  
**and** *WT*:  $\mathcal{I}2 \vdash_{res} RES$  *res s*  $\checkmark$   
**and** *outs*: *outs- $\mathcal{I}$*   $\mathcal{I}1 = UNIV$   
**shows** *conv*  $\triangleright RES$  *res s* =  $RES$  (*apply-wiring fg res*) *s*  
**using** *wiring*  
**proof cases**  
**case** (*wiring fg*)  
**have**  $\mathcal{I}$ -full, $\mathcal{I}2 \vdash_C$  *conv*  $\sim$  *map-converter id id fg 1<sub>C</sub>* **using** *wiring(2)*  
**by**(*rule eq- $\mathcal{I}$ -converter-mono*)(*simp-all add: le- $\mathcal{I}$ -def outs*)  
**with** *WT* **have** *conv*  $\triangleright RES$  *res s* = *map-converter id id fg 1<sub>C</sub>*  $\triangleright RES$  *res s*  
**by**(*rule eq- $\mathcal{I}$ -attach*)  
**also have**  $\dots = RES$  (*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-refl*)

**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*  $\Rightarrow$  (*'s*  $\Rightarrow$  *'c*  $\Rightarrow$  (*'d*  $\times$  *'s, 'e, 'f*)  
*gpv*)  $\Rightarrow$  (*'s*  $\Rightarrow$  *'a*  $\Rightarrow$  (*'b*  $\times$  *'s, 'e, 'f*) *gpv*)  
**where** *attach-wiring* = ( $\lambda$ (*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*  $\checkmark$   
**shows**  $\mathcal{I}1, \mathcal{I}3 \vdash_C$  *conv*  $\odot$  *CNV callee s*  $\sim$  *CNV* (*attach-wiring w callee*) *s*  
**using** *wiring*  
**proof cases**  
**case** (*wiring fg*)  
**from** *wiring(2)* **have**  $\mathcal{I}1, \mathcal{I}3 \vdash_C$  *conv*  $\odot$  *CNV callee s*  $\sim$  *map-converter id id fg 1<sub>C</sub>*  $\odot$  *CNV callee s*  
**by**(*rule eq- $\mathcal{I}$ -comp-cong*)(*rule eq- $\mathcal{I}$ -converter-refl*[*OF WT*])  
**also have** *map-converter id id fg 1<sub>C</sub>*  $\odot$  *CNV callee s* = *map-converter fg id id* (*CNV callee s*)  
**by**(*subst comp-converter-map-converter1*)(*simp add: comp-converter-id-left*)  
**also have**  $\dots = CNV$  (*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**  $\langle \circ_w \rangle$  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<sub>C</sub>*  
and WT1:  $\mathcal{I}, \mathcal{I}' \vdash_C$  conv1  $\checkmark$   
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<sub>C</sub>*  
and WT2:  $\mathcal{I}', \mathcal{I}'' \vdash_C$  conv2  $\checkmark$   
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<sub>C</sub>*  $\odot$  *map-converter id id f' g' 1<sub>C</sub>*  
using conv1 conv2 **by** (*rule eq- $\mathcal{I}$ -comp-cong*)  
**also have** *map-converter id id f g 1<sub>C</sub>*  $\odot$  *map-converter id id f' g' 1<sub>C</sub>* = *map-converter id id (f'  $\circ$  f) (g  $\circ$  g') 1<sub>C</sub>*  
by(*simp add: comp-converter-map-converter2 comp-converter-id-right*)  
**also have**  $\mathcal{I}, \mathcal{I}'' \vdash_C$  conv1  $\odot$  conv2  $\checkmark$  **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**  $\langle |_w \rangle$  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  $|_w$  conv2) (fg  $|_w$  fg')  
**proof** –  
from *assms(1)* **obtain** f1 g1



**where**  $conv1: \mathcal{I}1, \mathcal{I}1' \vdash_C conv1 \sim map\text{-converter } id \ id \ f1 \ g1 \ 1_C$   
**and**  $WT1: \mathcal{I}1, \mathcal{I}1' \vdash_C conv1 \ \checkmark$   
**and**  $[simp]: fg = (f1, g1)$   
**by cases**  
**from**  $assms(2)$  **obtain**  $f2 \ g2$   
**where**  $conv2: \mathcal{I}2, \mathcal{I}2' \vdash_C conv2 \sim map\text{-converter } id \ id \ f2 \ g2 \ 1_C$   
**and**  $WT2: \mathcal{I}2, \mathcal{I}2' \vdash_C conv2 \ \checkmark$   
**and**  $[simp]: fg' = (f2, g2)$   
**by cases**  
**from**  $eq\mathcal{I}\text{-converterD-WT1}[OF \ conv1 \ WT1]$  **have**  $\mathcal{I}1: \mathcal{I}1 \leq map\mathcal{I} \ f1 \ g1 \ \mathcal{I}1'$   
**by**( $rule \ WT\text{-map-converter-idD}$ )  
**from**  $eq\mathcal{I}\text{-converterD-WT1}[OF \ conv2 \ WT2]$  **have**  $\mathcal{I}2: \mathcal{I}2 \leq map\mathcal{I} \ f2 \ g2 \ \mathcal{I}2'$   
**by**( $rule \ WT\text{-map-converter-idD}$ )  
**have**  $WT': \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C map\text{-converter } id \ id \ (map\text{-sum } f1 \ f2)$   
 $(map\text{-sum } g1 \ g2) \ 1_C \ \checkmark$   
**by**( $auto \ intro!: \ WT\text{-converter-map-converter } WT\text{-converter-mono}[OF \ WT\text{-converter-id}$   
 $order\text{-refl}] \ \mathcal{I}1 \ \mathcal{I}2$ )  
**have**  $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C conv1 \ |_{=} \ conv2 \ \sim map\text{-converter } id \ id \ f1 \ g1$   
 $1_C \ |_{=} map\text{-converter } id \ id \ f2 \ g2 \ 1_C$   
**using**  $conv1 \ conv2$  **by**( $rule \ parallel\text{-converter2-eq}\mathcal{I}\text{-cong}$ )  
**also have**  $map\text{-converter } id \ id \ f1 \ g1 \ 1_C \ |_{=} map\text{-converter } id \ id \ f2 \ g2 \ 1_C = (1_C$   
 $|_{=} 1_C) \odot map\text{-converter } id \ id \ (map\text{-sum } f1 \ f2) \ (map\text{-sum } g1 \ g2) \ 1_C$   
**by**( $simp \ add: \ parallel\text{-converter2-map2-out } parallel\text{-converter2-map1-out } map\text{-sum.comp}$   
 $sum.map\text{-id0 } comp\text{-converter-map-converter2}[of \ id \ id, \ simplified] \ comp\text{-converter-id-right}$ )  
**also have**  $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C \dots \sim 1_C \odot map\text{-converter } id \ id \ (map\text{-sum}$   
 $f1 \ f2) \ (map\text{-sum } g1 \ g2) \ 1_C$   
**by**( $rule \ eq\mathcal{I}\text{-comp-cong}[OF \ parallel\text{-converter2-id-id}]$ )( $rule \ eq\mathcal{I}\text{-converter-refl}[OF$   
 $WT']$ )  
**finally show**  $?thesis$  **using**  $WT1 \ WT2$   
**by**( $auto \ simp \ add: \ parallel2\text{-wiring-simps } comp\text{-converter-id-left } intro!: \ wiringI$   
 $WT\text{-converter-parallel-converter2}$ )  
**qed**

**lemma**  $apply\text{-parallel2} \ [simp]:$

$apply\text{-wiring} \ (fg \ |_{w} \ fg') \ (res1 \ \oplus_O \ res2) = (apply\text{-wiring} \ fg \ res1 \ \oplus_O \ apply\text{-wiring} \ fg' \ res2)$

**proof** –

**have** $[simp]: fg = (f1, g1) \implies fg' = (f2, g2) \implies$   
 $map\text{-spmf} \ (map\text{-prod} \ (map\text{-sum } g1 \ g2) \ id) \ ((res1 \ \oplus_O \ res2) \ s \ (map\text{-sum } f1$   
 $f2 \ q)) =$   
 $((\lambda s \ q. \ map\text{-spmf} \ (map\text{-prod } g1 \ id) \ (res1 \ s \ (f1 \ q))) \ \oplus_O \ (\lambda s \ q. \ map\text{-spmf}$   
 $(map\text{-prod } g2 \ id) \ (res2 \ s \ (f2 \ q)))) \ s \ q$  **for**  $f1 \ g1 \ f2 \ g2 \ s \ q$   
**by**( $cases \ q$ )( $simp\text{-all } add: \ spmf.map\text{-comp } o\text{-def } apfst\text{-def } prod.map\text{-comp } split!: \ sum.splits$ )

**show**  $?thesis$

**by**( $cases \ fg; \ cases \ fg'$ )( $clarsimp \ simp \ add: \ parallel2\text{-wiring-simps } apply\text{-wiring-simps}$   
 $fun\text{-eq-iff } map\text{-fun-def } o\text{-def}$ )

**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<sub>w</sub>* :: (( $'a + 'b$ ) +  $'c$ , ( $'d + 'e$ ) +  $'f$ ,  $'a + ('b + 'c)$ ,  $'d + ('e + 'f)$ ) *wiring*

**where** *lassocr<sub>w</sub>* = (*rsuml*, *lsumr*)

**definition** *rassoctl<sub>w</sub>* :: ( $'a + ('b + 'c)$ ,  $'d + ('e + 'f)$ , ( $'a + 'b$ ) +  $'c$ , ( $'d + 'e$ ) +  $'f$ ) *wiring*

**where** *rassoctl<sub>w</sub>* = (*lsumr*, *rsuml*)

**definition** *swap<sub>w</sub>* :: ( $'a + 'b$ ,  $'c + 'd$ ,  $'b + 'a$ ,  $'d + 'c$ ) *wiring* **where**

*swap<sub>w</sub>* = (*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<sub>C</sub>* *lassocr<sub>w</sub>*

**unfolding** *lassocr<sub>C</sub>-def* *lassocr<sub>w</sub>-def*

**by**(*subst* *map-converter-id-move-right*)(*auto* *intro!*: *wiringI* *eq- $\mathcal{I}$ -converter-refl* *WT-converter-map-converter*)

**lemma** *wiring-rassoctl* [simp, *wiring-intro*]:

*wiring* ( $\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)$ ) (( $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$ )  $\oplus_{\mathcal{I}} \mathcal{I}3$ ) *rassoctl<sub>C</sub>* *rassoctl<sub>w</sub>*

**unfolding** *rassoctl<sub>C</sub>-def* *rassoctl<sub>w</sub>-def*

**by**(*subst* *map-converter-id-move-right*)(*auto* *intro!*: *wiringI* *eq- $\mathcal{I}$ -converter-refl* *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<sub>C</sub>* *swap<sub>w</sub>*

**unfolding** *swap<sub>C</sub>-def* *swap<sub>w</sub>-def*

**by**(*subst* *map-converter-id-move-right*)(*auto* *intro!*: *wiringI* *eq- $\mathcal{I}$ -converter-refl* *WT-converter-map-converter*)

**lemma** *apply-lassocr<sub>w</sub>* [simp]: *apply-wiring* *lassocr<sub>w</sub>* ( $res1 \oplus_O res2 \oplus_O res3$ ) = ( $res1 \oplus_O res2$ )  $\oplus_O res3$

**by**(*simp* *add*: *apply-wiring-def* *lassocr<sub>w</sub>-def* *map-rsuml-plus-oracle*)

**lemma** *apply-rassoctl<sub>w</sub>* [simp]: *apply-wiring* *rassoctl<sub>w</sub>* (( $res1 \oplus_O res2$ )  $\oplus_O res3$ ) =  $res1 \oplus_O res2 \oplus_O res3$

**by**(*simp* *add*: *apply-wiring-def* *rassoctl<sub>w</sub>-def* *map-lsumr-plus-oracle*)

**lemma** *apply-swap<sub>w</sub>* [simp]: *apply-wiring* *swap<sub>w</sub>* ( $res1 \oplus_O res2$ ) =  $res2 \oplus_O res1$

**by**(*simp* *add*: *apply-wiring-def* *swap<sub>w</sub>-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*  $\Rightarrow$  ('*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** *I-real* :: *security*  $\Rightarrow$  ('*a*, '*b*)  $\mathcal{I}$   
**and** *I-ideal* :: *security*  $\Rightarrow$  ('*c*, '*d*)  $\mathcal{I}$   
**and** *I-common* :: *security*  $\Rightarrow$  ('*e*, '*f*)  $\mathcal{I}$   
**and** *bound* :: *security*  $\Rightarrow$  *enat*  
**and** *lossless* :: *bool*

**assumes** *WT-real* [*WT-intro*]:  $\bigwedge \eta. \mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{\text{res}} \text{real-resource } \eta \checkmark$

**and** *WT-ideal* [*WT-intro*]:  $\bigwedge \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{\text{res}} \text{ideal-resource } \eta \checkmark$

**and** *WT-sim* [*WT-intro*]:  $\bigwedge \eta. \mathcal{I}\text{-real } \eta, \mathcal{I}\text{-ideal } \eta \vdash_C \text{sim } \eta \checkmark$

**and** *adv*:  $\bigwedge \mathcal{A} :: \text{security} \Rightarrow$  ('*a* + '*e*, '*b* + '*f*) *distinguisher*.

$\llbracket \bigwedge \eta. \mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{A} \eta \checkmark;$

$\bigwedge \eta. \text{interaction-bounded-by } (\lambda-. \text{True}) (\mathcal{A} \eta) (\text{bound } \eta);$

$\bigwedge \eta. \text{lossless} \Rightarrow \text{plossless-gpv } (\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{A} \eta) \rrbracket$

$\Rightarrow \text{negligible } (\lambda \eta. \text{advantage } (\mathcal{A} \eta) (\text{sim } \eta |_{= 1_C} \triangleright \text{ideal-resource } \eta) (\text{real-resource } \eta))$

**locale** *constructive-security* =

*constructive-security-aux* *real-resource* *ideal-resource* *sim* *I-real* *I-ideal* *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** *I-real* :: *security*  $\Rightarrow$  ('*a*, '*b*)  $\mathcal{I}$   
**and** *I-ideal* :: *security*  $\Rightarrow$  ('*c*, '*d*)  $\mathcal{I}$   
**and** *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** *correct*:  $\exists \text{cnv}. \forall \mathcal{D} :: \text{security} \Rightarrow$  ('*c* + '*e*, '*d* + '*f*) *distinguisher*.

$(\forall \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{D} \eta \checkmark)$

$\rightarrow (\forall \eta. \text{interaction-bounded-by } (\lambda-. \text{True}) (\mathcal{D} \eta) (\text{bound } \eta))$

$\rightarrow (\forall \eta. \text{lossless} \rightarrow \text{plossless-gpv } (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D} \eta))$

$\rightarrow (\forall \eta. \text{wiring } (\mathcal{I}\text{-ideal } \eta) (\mathcal{I}\text{-real } \eta) (\text{cnv } \eta) (\text{w } \eta)) \wedge$

$\text{negligible } (\lambda\eta. \text{advantage } (\mathcal{D} \ \eta) \ (\text{ideal-resource } \ \eta) \ (\text{cnv } \ \eta \mid = 1_C \triangleright \text{real-resource } \ \eta))$

**locale** *constructive-security2* =

*constructive-security-aux* *real-resource* *ideal-resource* *sim* *I-real* *I-ideal* *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** *I-real* :: *security*  $\Rightarrow$  ('a, 'b) *I*  
**and** *I-ideal* :: *security*  $\Rightarrow$  ('c, 'd) *I*  
**and** *I-common* :: *security*  $\Rightarrow$  ('e, 'f) *I*  
**and** *bound* :: *security*  $\Rightarrow$  *enat*  
**and** *lossless* :: *bool*  
**and** *w* :: *security*  $\Rightarrow$  ('c, 'd, 'a, 'b) *wiring*

+

**assumes** *sim*:  $\exists \text{cnv}. \forall \eta. \text{wiring } (\text{I-ideal } \ \eta) \ (\text{I-real } \ \eta) \ (\text{cnv } \ \eta) \ (w \ \eta) \wedge \text{wiring } (\text{I-ideal } \ \eta) \ (\text{I-ideal } \ \eta) \ (\text{cnv } \ \eta \odot \text{sim } \ \eta) \ (\text{id}, \text{id})$

**begin**

**lemma** *constructive-security*:

*constructive-security* *real-resource* *ideal-resource* *sim* *I-real* *I-ideal* *I-common*  
*bound* *lossless* *w*

**proof**

**from** *sim* **obtain** *cnv*

**where** *w*:  $\bigwedge \eta. \text{wiring } (\text{I-ideal } \ \eta) \ (\text{I-real } \ \eta) \ (\text{cnv } \ \eta) \ (w \ \eta)$   
**and** *inverse*:  $\bigwedge \eta. \text{wiring } (\text{I-ideal } \ \eta) \ (\text{I-ideal } \ \eta) \ (\text{cnv } \ \eta \odot \text{sim } \ \eta) \ (\text{id}, \text{id})$   
**by** *blast*

**show**  $\exists \text{cnv}. \forall \mathcal{D}. (\forall \eta. \text{I-ideal } \ \eta \oplus_{\mathcal{I}} \text{I-common } \ \eta \vdash_g \mathcal{D} \ \eta \ \checkmark)$

$\longrightarrow (\forall \eta. \text{interaction-any-bounded-by } (\mathcal{D} \ \eta) \ (\text{bound } \ \eta))$

$\longrightarrow (\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv } (\text{I-ideal } \ \eta \oplus_{\mathcal{I}} \text{I-common } \ \eta) \ (\mathcal{D} \ \eta))$

$\longrightarrow (\forall \eta. \text{wiring } (\text{I-ideal } \ \eta) \ (\text{I-real } \ \eta) \ (\text{cnv } \ \eta) \ (w \ \eta)) \wedge$

$\text{negligible } (\lambda\eta. \text{advantage } (\mathcal{D} \ \eta) \ (\text{ideal-resource } \ \eta) \ (\text{cnv } \ \eta \mid = 1_C \triangleright \text{real-resource } \ \eta))$

**proof**(*intro strip exI conjI*)

**fix**  $\mathcal{D}$  :: *security*  $\Rightarrow$  ('c + 'e, 'd + 'f) *distinguisher*

**assume** *WT-D* [*rule-format*, *WT-intro*]:  $\forall \eta. \text{I-ideal } \ \eta \oplus_{\mathcal{I}} \text{I-common } \ \eta \vdash_g \mathcal{D} \ \eta \ \checkmark$

**and** *bound* [*rule-format*, *interaction-bound*]:  $\forall \eta. \text{interaction-bounded-by } (\lambda. \text{True}) \ (\mathcal{D} \ \eta) \ (\text{bound } \ \eta)$

**and** *lossless* [*rule-format*]:  $\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv } (\text{I-ideal } \ \eta \oplus_{\mathcal{I}} \text{I-common } \ \eta) \ (\mathcal{D} \ \eta)$

**show** *wiring* (*I-ideal*  $\ \eta$ ) (*I-real*  $\ \eta$ ) (*cnv*  $\ \eta$ ) (*w*  $\ \eta$ ) **for**  $\eta$  **by** *fact*

**let**  $?A = \lambda\eta. \text{outs-}\mathcal{I} \ (\text{I-ideal } \ \eta)$

**let**  $?cnv = \lambda\eta. \text{restrict-converter } (?A \ \eta) \ (\text{I-real } \ \eta) \ (\text{cnv } \ \eta)$

**let**  $?A = \lambda\eta. \text{absorb } (\mathcal{D} \ \eta) \ (?cnv \ \eta \mid = 1_C)$

**have**  $eq$ :  $advantage (\mathcal{D} \eta) (ideal-resource \eta) (cnv \eta \models 1_C \triangleright real-resource \eta) =$   
 $advantage (?A \eta) (sim \eta \models 1_C \triangleright ideal-resource \eta) (real-resource \eta)$  **for**  $\eta$   
**proof** –  
**from**  $w[of \eta]$  **have**  $[WT-intro]: \mathcal{I}\text{-ideal } \eta, \mathcal{I}\text{-real } \eta \vdash_C cnv \eta \checkmark$  **by cases**  
**have**  $\mathcal{I}\text{-ideal } \eta, \mathcal{I}\text{-ideal } \eta \vdash_C ?cnv \eta \odot sim \eta \sim cnv \eta \odot sim \eta$   
**by**( $rule\ eq\text{-}\mathcal{I}\text{-comp}\text{-cong}\ eq\text{-}\mathcal{I}\text{-restrict}\text{-converter}\ WT\text{-intro}\ order\text{-refl}\ eq\text{-}\mathcal{I}\text{-converter}\text{-refl}$ )**+**  
**also from**  $inverse[of \eta]$  **have**  $\mathcal{I}\text{-ideal } \eta, \mathcal{I}\text{-ideal } \eta \vdash_C cnv \eta \odot sim \eta \sim 1_C$   
**by cases simp**  
**finally have**  $inverse'$ :  $\mathcal{I}\text{-ideal } \eta, \mathcal{I}\text{-ideal } \eta \vdash_C ?cnv \eta \odot sim \eta \sim 1_C$  .  
**hence**  $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta, \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_C ?cnv \eta \odot$   
 $sim \eta \models 1_C \sim 1_C \models 1_C$   
**by**( $rule\ parallel\text{-converter}2\text{-eq}\text{-}\mathcal{I}\text{-cong}$ )( $intro\ eq\text{-}\mathcal{I}\text{-converter}\text{-refl}\ WT\text{-intro}$ )  
**also have**  $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta, \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_C 1_C \models$   
 $1_C \sim 1_C$   
**by**( $rule\ parallel\text{-converter}2\text{-id}\text{-id}$ )  
**also**  
**have**  $eq1$ :  $connect (\mathcal{D} \eta) (?cnv \eta \models 1_C \triangleright sim \eta \models 1_C \triangleright ideal-resource \eta) =$   
 $connect (\mathcal{D} \eta) (1_C \triangleright ideal-resource \eta)$   
**unfolding**  $attach\text{-compose}[symmetric]$   $comp\text{-converter}\text{-parallel}2$   $comp\text{-converter}\text{-id}\text{-right}$   
**by**( $rule\ connect\text{-eq}\text{-resource}\text{-cong}\ WT\text{-intro}\ eq\text{-}\mathcal{I}\text{-attach}\text{-on}'\ calculation$ )**+**( $fastforce$   
 $intro: WT\text{-intro}$ )**+**  
  
**have**  $*$ :  $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta, \mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_C ?cnv \eta \models$   
 $1_C \sim cnv \eta \models 1_C$   
**by**( $rule\ parallel\text{-converter}2\text{-eq}\text{-}\mathcal{I}\text{-cong}\ eq\text{-}\mathcal{I}\text{-restrict}\text{-converter}$ )**+**( $auto\ intro:$   
 $WT\text{-intro}\ eq\text{-}\mathcal{I}\text{-converter}\text{-refl}$ )  
**have**  $eq2$ :  $connect (\mathcal{D} \eta) (?cnv \eta \models 1_C \triangleright real-resource \eta) = connect (\mathcal{D} \eta)$   
 $(cnv \eta \models 1_C \triangleright real-resource \eta)$   
**by**( $rule\ connect\text{-eq}\text{-resource}\text{-cong}\ WT\text{-intro}\ eq\text{-}\mathcal{I}\text{-attach}\text{-on}'\ *$ )**+**( $auto\ intro:$   
 $WT\text{-intro}$ )  
**show**  $?thesis$  **unfolding**  $advantage\text{-def}$  **by**( $simp\ add: distinguish\text{-attach}[symmetric]$   
 $eq1\ eq2$ )  
**qed**  
**have**  $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g ?A \eta \checkmark$  **for**  $\eta$   
**proof** –  
**from**  $w$  **have**  $[WT-intro]: \mathcal{I}\text{-ideal } \eta, \mathcal{I}\text{-real } \eta \vdash_C cnv \eta \checkmark$  **by cases**  
**show**  $?thesis$  **by**( $rule\ WT\text{-intro}$ )**+**  
**qed**  
**moreover**  
**have**  $interaction\text{-any}\text{-bounded}\text{-by}$  ( $absorb (\mathcal{D} \eta) (?cnv \eta \models 1_C)$ ) ( $bound \eta$ ) **for**  
 $\eta$   
**proof** –  
**from**  $w[of \eta]$  **obtain**  $f\ g$  **where**  $[simp]: w \eta = (f, g)$   
**and**  $[WT-intro]: \mathcal{I}\text{-ideal } \eta, \mathcal{I}\text{-real } \eta \vdash_C cnv \eta \checkmark$   
**and**  $eq$ :  $\mathcal{I}\text{-ideal } \eta, \mathcal{I}\text{-real } \eta \vdash_C cnv \eta \sim map\text{-converter}\ id\ id\ f\ g\ 1_C$  **by cases**  
**from**  $eq\text{-}\mathcal{I}\text{-restrict}\text{-converter}\text{-cong}[OF\ eq\ order\text{-refl}]$   
**have**  $*$ :  $restrict\text{-converter} (?A \eta) (\mathcal{I}\text{-real } \eta) (cnv \eta) =$   
 $restrict\text{-converter} (?A \eta) (\mathcal{I}\text{-real } \eta) (map\text{-converter}\ f\ g\ id\ id\ 1_C)$   
**by**( $subst\ map\text{-converter}\text{-id}\text{-move}\text{-right}$ )  $simp$

**show** *?thesis unfolding* \* **by** *interaction-bound-converter simp*  
**qed**  
**moreover have** *plossless-gpv* ( $\mathcal{I}$ -real  $\eta \oplus_{\mathcal{I}} \mathcal{I}$ -common  $\eta$ ) (*?A*  $\eta$ )  
**if** *lossless* **for**  $\eta$   
**proof** –  
**from** *w*[*of*  $\eta$ ] **obtain** *f g* **where** [*simp*]:  $w \eta = (f, g)$   
**and** *cnv* [*WT-intro*]:  $\mathcal{I}$ -ideal  $\eta$ ,  $\mathcal{I}$ -real  $\eta \vdash_C \text{cnv } \eta \checkmark$   
**and** *eq*:  $\mathcal{I}$ -ideal  $\eta$ ,  $\mathcal{I}$ -real  $\eta \vdash_C \text{cnv } \eta \sim \text{map-converter } \text{id } \text{id } f g \ 1_C$  **by** *cases*  
**from** *eq- $\mathcal{I}$ -converterD-WT1*[*OF eq cnv*] **have**  $\mathcal{I}$ :  $\mathcal{I}$ -ideal  $\eta \leq \text{map-}\mathcal{I} f g$  ( $\mathcal{I}$ -real  
 $\eta$ )  
**by**(*rule WT-map-converter-idD*)  
**with** *WT-converter-id* **have** [*WT-intro*]:  $\mathcal{I}$ -ideal  $\eta$ ,  $\text{map-}\mathcal{I} f g$  ( $\mathcal{I}$ -real  $\eta$ )  $\vdash_C$   
 $1_C \checkmark$   
**by**(*rule WT-converter-mono*) *simp*  
**have** *id*: *plossless-converter* ( $\mathcal{I}$ -ideal  $\eta$ ) ( $\text{map-}\mathcal{I} f g$  ( $\mathcal{I}$ -real  $\eta$ ))  $1_C$   
**by**(*rule plossless-converter-mono*)(*rule plossless-id-converter order-refl*  $\mathcal{I}$   
 $\text{WT-intro}$ )+  
**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* ( $\mathcal{D}$   $\eta$ ) (*ideal-resource*  $\eta$ ) (*cnv*  $\eta \models$   
 $1_C \triangleright \text{real-resource } \eta$ ))  
**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*  
 $\eta$ ) (*bound-sim*  $\eta$ )  
**and** *bound-le*:  $\bigwedge \eta$ . *bound-outer*  $\eta * \max$  (*bound-sim*  $\eta$ )  $1 \leq \text{bound-inner } \eta$   
**and** *lossless-sim* [*plossless-intro*]:  $\bigwedge \eta$ . *lossless-inner*  $\implies$  *plossless-converter* ( $\mathcal{I}$ -real  
 $\eta$ ) ( $\mathcal{I}$ -middle  $\eta$ ) (*sim-outer*  $\eta$ )  
**shows** *constructive-security real ideal* ( $\lambda \eta$ . *sim-outer*  $\eta \odot \text{sim-inner } \eta$ )  $\mathcal{I}$ -real  
 $\mathcal{I}$ -inner  $\mathcal{I}$ -common *bound-outer* (*lossless-outer*  $\vee$  *lossless-inner*) ( $\lambda \eta$ .  $w1 \eta \circ_w w2$ )

$\eta$ )  
**proof**  
**interpret** *inner*: *constructive-security middle ideal sim-inner I-middle I-inner I-common bound-inner lossless-inner w1* **by fact**  
**interpret** *outer*: *constructive-security real middle sim-outer I-real I-middle I-common bound-outer lossless-outer w2* **by fact**

**show**  $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{\text{res}} \text{real } \eta \checkmark$   
**and**  $\mathcal{I}\text{-inner } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{\text{res}} \text{ideal } \eta \checkmark$   
**and**  $\mathcal{I}\text{-real } \eta, \mathcal{I}\text{-inner } \eta \vdash_C \text{sim-outer } \eta \odot \text{sim-inner } \eta \checkmark$  **for**  $\eta$  **by**(rule *WT-intro*)+

**{** **fix**  $\mathcal{A} :: \text{security} \Rightarrow ('g + 'b, 'h + 'd) \text{distinguisher}$   
**assume** *WT* [*WT-intro*]:  $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{A} \eta \checkmark$  **for**  $\eta$   
**assume** *bound-outer* [*interaction-bound*]: *interaction-bounded-by* ( $\lambda\cdot$ . *True*) ( $\mathcal{A} \eta$ ) (*bound-outer*  $\eta$ ) **for**  $\eta$   
**assume** *lossless* [*plossless-intro*]:  
 $\text{lossless-outer} \vee \text{lossless-inner} \implies \text{plossless-gpv } (\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta)$   
( $\mathcal{A} \eta$ ) **for**  $\eta$

**let**  $?A = \lambda\eta. \text{absorb } (\mathcal{A} \eta) (\text{sim-outer } \eta \mid= 1_C)$   
**have**  $\mathcal{I}\text{-middle } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g ?A \eta \checkmark$  **for**  $\eta$  **by**(rule *WT-intro*)+  
**moreover have** *interaction-any-bounded-by* ( $?A \eta$ ) (*bound-inner*  $\eta$ ) **for**  $\eta$   
**by** *interaction-bound-converter*(rule *bound-le*)  
**moreover have** *plossless-gpv* ( $\mathcal{I}\text{-middle } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta$ ) ( $?A \eta$ ) **if** *lossless-inner* **for**  $\eta$   
**by**(rule *plossless-intro WT-intro* | *simp add: that*)+  
**ultimately have** *negligible* ( $\lambda\eta. \text{advantage } (?A \eta) (\text{sim-inner } \eta \mid= 1_C \triangleright \text{ideal } \eta)$ ) (*middle*  $\eta$ )  
**by**(rule *inner.adv*)  
**also have** *negligible* ( $\lambda\eta. \text{advantage } (\mathcal{A} \eta) (\text{sim-outer } \eta \mid= 1_C \triangleright \text{middle } \eta)$ ) (*real*  $\eta$ )  
**by**(rule *outer.adv*[*OF WT bound-outer lossless*]) *simp*  
**finally** (*negligible-plus*)  
**show** *negligible* ( $\lambda\eta. \text{advantage } (\mathcal{A} \eta) (\text{sim-outer } \eta \odot \text{sim-inner } \eta \mid= 1_C \triangleright \text{ideal } \eta)$ ) (*real*  $\eta$ )  
**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*:  $\wedge \mathcal{D}. \llbracket \wedge \eta. \mathcal{I}\text{-inner } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{D} \eta \checkmark;$   
 $\wedge \eta. \text{interaction-any-bounded-by } (\mathcal{D} \eta) (\text{bound-inner } \eta);$   
 $\wedge \eta. \text{lossless-inner} \implies \text{plossless-gpv } (\mathcal{I}\text{-inner } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D} \eta)$   
 $\rrbracket$   
 $\implies (\forall \eta. \text{wiring } (\mathcal{I}\text{-inner } \eta) (\mathcal{I}\text{-middle } \eta) (\text{cnv-inner } \eta) (w1 \eta)) \wedge$   
 $\text{negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (\text{ideal } \eta) (\text{cnv-inner } \eta \mid_{= 1_C} \triangleright \text{middle}$   
 $\eta))$   
**by** *blast*  
**from** *outer.correct* **obtain** *cnv-outer*  
**where** *correct-outer*:  $\wedge \mathcal{D}. \llbracket \wedge \eta. \mathcal{I}\text{-middle } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{D} \eta \checkmark;$   
 $\wedge \eta. \text{interaction-any-bounded-by } (\mathcal{D} \eta) (\text{bound-outer } \eta);$   
 $\wedge \eta. \text{lossless-outer} \implies \text{plossless-gpv } (\mathcal{I}\text{-middle } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D}$   
 $\eta) \rrbracket$   
 $\implies (\forall \eta. \text{wiring } (\mathcal{I}\text{-middle } \eta) (\mathcal{I}\text{-real } \eta) (\text{cnv-outer } \eta) (w2 \eta)) \wedge$   
 $\text{negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (\text{middle } \eta) (\text{cnv-outer } \eta \mid_{= 1_C} \triangleright \text{real}$   
 $\eta))$   
**by** *blast*  
**show**  $\exists \text{cnv}. \forall \mathcal{D}. (\forall \eta. \mathcal{I}\text{-inner } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{D} \eta \checkmark) \longrightarrow$   
 $(\forall \eta. \text{interaction-any-bounded-by } (\mathcal{D} \eta) (\text{bound-outer } \eta)) \longrightarrow$   
 $(\forall \eta. \text{lossless-outer} \vee \text{lossless-inner} \longrightarrow \text{plossless-gpv } (\mathcal{I}\text{-inner } \eta \oplus_{\mathcal{I}}$   
 $\mathcal{I}\text{-common } \eta) (\mathcal{D} \eta)) \longrightarrow$   
 $(\forall \eta. \text{wiring } (\mathcal{I}\text{-inner } \eta) (\mathcal{I}\text{-real } \eta) (\text{cnv } \eta) (w1 \eta \circ_w w2 \eta)) \wedge$   
 $\text{negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (\text{ideal } \eta) (\text{cnv } \eta \mid_{= 1_C} \triangleright \text{real } \eta))$   
**proof** (*intro exI strip conjI*)  
**fix**  $\mathcal{D} :: \text{security} \implies ('e + 'b, 'f + 'd) \text{distinguisher}$   
**assume** *WT-D* [*rule-format, WT-intro*]:  $\forall \eta. \mathcal{I}\text{-inner } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g$   
 $\mathcal{D} \eta \checkmark$   
**and** *bound* [*rule-format, interaction-bound*]:  $\forall \eta. \text{interaction-bounded-by } (\lambda.$   
 $\text{True}) (\mathcal{D} \eta) (\text{bound-outer } \eta)$   
**and** *lossless* [*rule-format*]:  $\forall \eta. \text{lossless-outer} \vee \text{lossless-inner} \longrightarrow \text{plossless-gpv}$   
 $(\mathcal{I}\text{-inner } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D} \eta)$   
  
**let**  $?cnv = \lambda \eta. \text{cnv-inner } \eta \odot \text{cnv-outer } \eta$   
  
**have** *bound'*: *interaction-any-bounded-by* ( $\mathcal{D} \eta$ ) (*bound-inner*  $\eta$ ) **for**  $\eta$  **using**  
*bound[of  $\eta$ ]* *bound-le[of  $\eta$ ]*  
**by** (*clarsimp elim!*: *interaction-bounded-by-mono order-trans[rotated] simp*  
*add: max-def*)  
 $(\text{metis } (\text{full-types}) \text{linorder-linear more-arith-simps(6) mult-left-mono}$   
 $\text{zero-le})$   
**from** *correct-inner* [*OF WT-D bound' lossless*]  
**have**  $w1: \wedge \eta. \text{wiring } (\mathcal{I}\text{-inner } \eta) (\mathcal{I}\text{-middle } \eta) (\text{cnv-inner } \eta) (w1 \eta)$   
**and** *adv1*:  $\text{negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (\text{ideal } \eta) (\text{cnv-inner } \eta \mid_{= 1_C} \triangleright$   
 $\text{middle } \eta))$   
**by** *auto*  
  
**obtain**  $f g$  **where** *WT-inner* [*WT-intro*]:  $\wedge \eta. \mathcal{I}\text{-inner } \eta, \mathcal{I}\text{-middle } \eta \vdash_{\mathcal{C}}$   
 $\text{cnv-inner } \eta \checkmark$



```

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

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

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

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

have eq1': connect ( $\mathcal{D} \ \eta$ ) (cnv-inner  $\eta \mid = \ 1_C \triangleright \ \text{middle } \eta$ ) = connect (? $\mathcal{D} \ \eta$ )
( $\text{middle } \eta$ ) for  $\eta$ 
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 (? $\mathcal{D} \ \eta$ ) (cnv-outer  $\eta \mid = \ 1_C \triangleright \ \text{real } \eta$ ) = connect ( $\mathcal{D} \ \eta$ )

```

(*?cuv*  $\eta \models 1_C \odot 1_C \triangleright \text{real } \eta$ ) **for**  $\eta$   
**unfolding** *distinguish-attach*[*symmetric*] *attach-compose comp-converter-parallel2*[*symmetric*]  
**by**(*rule connect-eq-resource-cong WT-intro eq-I-attach-on' parallel-converter2-eq-I-cong*  
*eq1*[*symmetric*] *eq-I-converter-reflI order-refl|simp*)  
**show** *negligible* ( $\lambda\eta. \text{advantage } (\mathcal{D} \ \eta) \ (\text{ideal } \eta) \ (\text{?cuv } \eta \models 1_C \triangleright \text{real } \eta)$ )  
**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-cuv* [*WT-intro*]:  $\bigwedge\eta. \mathcal{I}\text{-common}' \ \eta, \mathcal{I}\text{-common } \eta \vdash_C \text{cuv } \eta \checkmark$   
**and** *bound* [*interaction-bound*]:  $\bigwedge\eta. \text{interaction-any-bounded-converter } (\text{cuv } \eta)$   
(*bound-cuv*  $\eta$ )  
**and** *bound-le*:  $\bigwedge\eta. \text{bound}' \ \eta * \max(\text{bound-cuv } \eta) \ 1 \leq \text{bound } \eta$   
**and** *lossless* [*plossless-intro*]:  $\bigwedge\eta. \text{lossless} \implies \text{plossless-converter } (\mathcal{I}\text{-common}'$   
 $\eta) \ (\mathcal{I}\text{-common } \eta) \ (\text{cuv } \eta)$   
**shows** *constructive-security*  
( $\lambda\eta. 1_C \models \text{cuv } \eta \triangleright \text{real-resource } \eta$ ) ( $\lambda\eta. 1_C \models \text{cuv } \eta \triangleright \text{ideal-resource } \eta$ )  
*sim*  
 $\mathcal{I}\text{-real } \mathcal{I}\text{-ideal } \mathcal{I}\text{-common}' \ \text{bound}' \ \text{lossless } w$   
**proof**  
**fix**  $\mathcal{A} :: \text{security} \implies ('a + 'g, 'b + 'h) \text{distinguisher}$   
**assume** *WT-A* [*WT-intro*]:  $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \ \eta \vdash_g \mathcal{A} \ \eta \checkmark$  **for**  $\eta$   
**assume** *bound-A* [*interaction-bound*]: *interaction-any-bounded-by* ( $\mathcal{A} \ \eta$ ) (*bound'*  
 $\eta$ ) **for**  $\eta$   
**assume** *lossless-A* [*plossless-intro*]: *lossless*  $\implies$  *plossless-gpv* ( $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}'$   
 $\eta$ ) ( $\mathcal{A} \ \eta$ ) **for**  $\eta$

**let**  $\text{?A} = \lambda\eta. \text{absorb } (\mathcal{A} \ \eta) \ (1_C \models \text{cuv } \eta)$

**have** *ideal*: *connect* ( $\mathcal{A} \ \eta$ ) (*sim*  $\eta \models 1_C \triangleright 1_C \models \text{cuv } \eta \triangleright \text{ideal-resource } \eta$ ) =  
*connect* ( $\text{?A } \eta$ ) (*sim*  $\eta \models 1_C \triangleright \text{ideal-resource } \eta$ ) **for**  $\eta$   
**by**(*simp add: distinguish-attach*[*symmetric*] *attach-compose*[*symmetric*] *comp-converter-parallel2*  
*comp-converter-id-left comp-converter-id-right*)  
**have** *real*: *connect* ( $\mathcal{A} \ \eta$ ) ( $1_C \models \text{cuv } \eta \triangleright \text{real-resource } \eta$ ) = *connect* ( $\text{?A } \eta$ )  
(*real-resource*  $\eta$ ) **for**  $\eta$   
**by**(*simp add: distinguish-attach*[*symmetric*])  
**have**  $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \ \eta \vdash_g \text{?A } \eta \checkmark$  **for**  $\eta$  **by**(*rule WT-intro*)  
**moreover** **have** *interaction-any-bounded-by* ( $\text{?A } \eta$ ) (*bound*  $\eta$ ) **for**  $\eta$   
**by** *interaction-bound-converter*(*use bound-le*[*of*  $\eta$ ] **in**  $\langle \text{simp add: max commute} \rangle$ )  
**moreover** **have** *plossless-gpv* ( $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \ \eta$ ) (*absorb* ( $\mathcal{A} \ \eta$ ) ( $1_C \models$   
 $\text{cuv } \eta$ )) **if** *lossless* **for**  $\eta$   
**by**(*rule plossless-intro WT-intro | simp add: that*)  
**ultimately show** *negligible* ( $\lambda\eta. \text{advantage } (\mathcal{A} \ \eta) \ (\text{sim } \eta \models 1_C \triangleright 1_C \models \text{cuv } \eta$   
 $\triangleright \text{ideal-resource } \eta) \ (1_C \models \text{cuv } \eta \triangleright \text{real-resource } \eta)$ )

**unfolding advantage-def ideal real** **by**(rule adv[unfolding advantage-def])  
**next**  
**from correct obtain** *cnv*  
**where** *correct'*:  $\bigwedge \mathcal{D}. \llbracket \bigwedge \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{D} \eta \checkmark; \bigwedge \eta. \text{interaction-any-bounded-by } (\mathcal{D} \eta) (\text{bound } \eta); \bigwedge \eta. \text{lossless} \implies \text{plossless-gpv } (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D} \eta) \rrbracket$   
 $\implies (\forall \eta. \text{wiring } (\mathcal{I}\text{-ideal } \eta) (\mathcal{I}\text{-real } \eta) (\text{cnv } \eta) (w \eta)) \wedge$   
 $\text{negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (\text{ideal-resource } \eta) (\text{cnv } \eta \models 1_C \triangleright$   
*real-resource*  $\eta))$   
**by** *blast*  
**show**  $\exists \text{cnv}. \forall \mathcal{D}. (\forall \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta \vdash_g \mathcal{D} \eta \checkmark) \longrightarrow$   
 $(\forall \eta. \text{interaction-any-bounded-by } (\mathcal{D} \eta) (\text{bound}' \eta)) \longrightarrow$   
 $(\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv } (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta) (\mathcal{D} \eta)) \longrightarrow$   
 $(\forall \eta. \text{wiring } (\mathcal{I}\text{-ideal } \eta) (\mathcal{I}\text{-real } \eta) (\text{cnv } \eta) (w \eta)) \wedge$   
 $\text{negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (1_C \models \text{cnv } \eta \triangleright \text{ideal-resource } \eta) (\text{cnv } \eta$   
 $\models 1_C \triangleright 1_C \models \text{cnv } \eta \triangleright \text{real-resource } \eta))$   
**proof**(intro exI conjI strip)  
**fix**  $\mathcal{D} :: \text{security} \Rightarrow ('c + 'g, 'd + 'h) \text{distinguisher}$   
**assume** *WT-D* [rule-format, *WT-intro*]:  $\forall \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta \vdash_g \mathcal{D}$   
 $\eta \checkmark$   
**and** *bound* [rule-format, *interaction-bound*]:  $\forall \eta. \text{interaction-bounded-by } (\lambda.$   
*True*)  $(\mathcal{D} \eta) (\text{bound}' \eta)$   
**and** *lossless* [rule-format]:  $\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv } (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}}$   
 $\mathcal{I}\text{-common}' \eta) (\mathcal{D} \eta)$   
  
**let**  $?D = \lambda \eta. \text{absorb } (\mathcal{D} \eta) (1_C \models \text{cnv } \eta)$   
**have** *WT-D'* [*WT-intro*]:  $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g ?D \eta \checkmark$  **for**  $\eta$  **by**(rule  
*WT-intro*)+  
**have** *bound'*: *interaction-any-bounded-by* ( $?D \eta$ ) (*bound*  $\eta$ ) **for**  $\eta$   
**by** *interaction-bound*(use *bound-le*[of  $\eta$ ] **in** *auto simp add: max-def split:*  
*if-split-asm*)  
**have** *plossless-gpv* ( $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta$ ) ( $?D \eta$ ) **if** *lossless* **for**  $\eta$   
**by**(rule *lossless that WT-intro plossless-intro*)+  
**from** *correct'*[*OF WT-D' bound' this*]  
**have** *w1*: *wiring* ( $\mathcal{I}\text{-ideal } \eta$ ) ( $\mathcal{I}\text{-real } \eta$ ) (*cnv*  $\eta$ ) (*w*  $\eta$ )  
**and** *adv'*: *negligible* ( $\lambda \eta. \text{advantage } (?D \eta) (\text{ideal-resource } \eta) (\text{cnv } \eta \models 1_C \triangleright$   
*real-resource*  $\eta))$  **for**  $\eta$   
**by** *auto*  
**show** *wiring* ( $\mathcal{I}\text{-ideal } \eta$ ) ( $\mathcal{I}\text{-real } \eta$ ) (*cnv*  $\eta$ ) (*w*  $\eta$ ) **for**  $\eta$  **by**(rule *w1*)  
**have**  $\text{cnv } \eta \models 1_C \triangleright 1_C \models \text{cnv } \eta \triangleright \text{real-resource } \eta = 1_C \models \text{cnv } \eta \triangleright \text{cnv } \eta$   
 $\models 1_C \triangleright \text{real-resource } \eta$  **for**  $\eta$   
**by**(*simp add: attach-compose[symmetric] comp-converter-parallel2 comp-converter-id-left*  
*comp-converter-id-right*)  
**with** *adv'*  
**show** *negligible* ( $\lambda \eta. \text{advantage } (\mathcal{D} \eta) (1_C \models \text{cnv } \eta \triangleright \text{ideal-resource } \eta) (\text{cnv } \eta$   
 $\models 1_C \triangleright 1_C \models \text{cnv } \eta \triangleright \text{real-resource } \eta))$   
**by**(*simp add: advantage-def distinguish-attach[symmetric]*)  
**qed**  
**qed**(rule *WT-intro*)+

**theorem** *constructive-security-trivial*:

**fixes** *res*  
**assumes** [*WT-intro*]:  $\bigwedge \eta. \mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \eta \vdash_{\text{res}} \text{res} \eta \checkmark$   
**shows** *constructive-security res res* ( $\lambda\text{-} 1_C$ )  $\mathcal{I} \mathcal{I} \mathcal{I}\text{-common}$  *bound lossless* ( $\lambda\text{-}$   
*(id, id)*)

**proof**

**show**  $\mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \eta \vdash_{\text{res}} \text{res} \eta \checkmark$  **and**  $\mathcal{I} \eta, \mathcal{I} \eta \vdash_C 1_C \checkmark$  **for**  $\eta$  **by**(*rule WT-intro*)**+**

**fix**  $\mathcal{A} :: \text{security} \Rightarrow ('a + 'b, 'c + 'd)$  *distinguisher*  
**assume** *WT* [*WT-intro*]:  $\mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \eta \vdash_g \mathcal{A} \eta \checkmark$  **for**  $\eta$   
**have** *connect* ( $\mathcal{A} \eta$ ) ( $1_C \models 1_C \triangleright \text{res} \eta$ ) = *connect* ( $\mathcal{A} \eta$ ) ( $1_C \triangleright \text{res} \eta$ ) **for**  $\eta$   
**by**(*rule connect-eq-resource-cong[OF WT]*)(*fastforce intro: WT-intro eq-I-attach-on'*  
*parallel-converter2-id-id*)**+**  
**then show** *negligible* ( $\lambda\eta. \text{advantage} (\mathcal{A} \eta) (1_C \models 1_C \triangleright \text{res} \eta) (\text{res} \eta)$ )  
**unfolding** *advantage-def* **by** *simp*

**next**

**show**  $\exists \text{cnv}. \forall \mathcal{D}. (\forall \eta. \mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \eta \vdash_g \mathcal{D} \eta \checkmark) \longrightarrow$   
 $(\forall \eta. \text{interaction-any-bounded-by} (\mathcal{D} \eta) (\text{bound} \eta)) \longrightarrow$   
 $(\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv} (\mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \eta) (\mathcal{D} \eta)) \longrightarrow$   
 $(\forall \eta. \text{wiring} (\mathcal{I} \eta) (\mathcal{I} \eta) (\text{cnv} \eta) (\text{id}, \text{id})) \wedge$   
 $\text{negligible} (\lambda\eta. \text{advantage} (\mathcal{D} \eta) (\text{res} \eta) (\text{cnv} \eta \models 1_C \triangleright \text{res} \eta))$

**proof**(*intro exI strip conjI*)

**fix**  $\mathcal{D} :: \text{security} \Rightarrow ('a + 'b, 'c + 'd)$  *distinguisher*  
**assume** *WT-D* [*rule-format, WT-intro*]:  $\forall \eta. \mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \eta \vdash_g \mathcal{D} \eta \checkmark$   
**and** *bound* [*rule-format, interaction-bound*]:  $\forall \eta. \text{interaction-bounded-by} (\lambda\text{-}$   
*True*) ( $\mathcal{D} \eta$ ) (*bound*  $\eta$ )  
**and** *lossless* [*rule-format*]:  $\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv} (\mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}$   
 $\eta) (\mathcal{D} \eta)$   
**show** *wiring* ( $\mathcal{I} \eta$ ) ( $\mathcal{I} \eta$ )  $1_C$  (*id, id*) **for**  $\eta$  **by** *simp*  
**have** *connect* ( $\mathcal{D} \eta$ ) ( $1_C \models 1_C \triangleright \text{res} \eta$ ) = *connect* ( $\mathcal{D} \eta$ ) ( $1_C \triangleright \text{res} \eta$ ) **for**  $\eta$   
**by**(*rule connect-eq-resource-cong*)(*rule WT-intro eq-I-attach-on' parallel-converter2-id-id*  
*order-refl*)**+**  
**then show** *negligible* ( $\lambda\eta. \text{advantage} (\mathcal{D} \eta) (\text{res} \eta) (1_C \models 1_C \triangleright \text{res} \eta)$ )  
**by**(*auto simp add: advantage-def*)

**qed**

**qed**

**theorem** *parallel-constructive-security*:

**assumes** *constructive-security real1 ideal1 sim1 I-real1 I-inner1 I-common1*  
*bound1 lossless1 w1*  
**assumes** *constructive-security real2 ideal2 sim2 I-real2 I-inner2 I-common2*  
*bound2 lossless2 w2*

**and** *lossless-real1* [*plossless-intro*]:  $\bigwedge \eta. \text{lossless2} \Longrightarrow \text{lossless-resource} (\mathcal{I}\text{-real1}$   
 $\eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common1} \eta) (\text{real1} \eta)$

**and** *lossless-sim2* [*plossless-intro*]:  $\bigwedge \eta. \text{lossless1} \Longrightarrow \text{plossless-converter} (\mathcal{I}\text{-real2}$   
 $\eta) (\mathcal{I}\text{-inner2} \eta) (\text{sim2} \eta)$

**and** *lossless-ideal2* [*plossless-intro*]:  $\bigwedge \eta. \text{lossless1} \implies \text{lossless-resource } (\mathcal{I}\text{-inner2 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) \text{ (ideal2 } \eta)$

**shows** *constructive-security*  $(\lambda \eta. \text{parallel-wiring } \triangleright \text{real1 } \eta \parallel \text{real2 } \eta) (\lambda \eta. \text{parallel-wiring } \triangleright \text{ideal1 } \eta \parallel \text{ideal2 } \eta) (\lambda \eta. \text{sim1 } \eta \models \text{sim2 } \eta)$

$(\lambda \eta. \mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta) (\lambda \eta. \mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta) (\lambda \eta. \mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta)$

$(\lambda \eta. \text{min } (\text{bound1 } \eta) (\text{bound2 } \eta)) (\text{lossless1} \vee \text{lossless2}) (\lambda \eta. w1 \eta \mid_w w2 \eta)$

**proof**

**interpret** *sec1*: *constructive-security* *real1* *ideal1* *sim1* *I-real1* *I-inner1* *I-common1* *bound1* *lossless1* *w1* **by fact**

**interpret** *sec2*: *constructive-security* *real2* *ideal2* *sim2* *I-real2* *I-inner2* *I-common2* *bound2* *lossless2* *w2* **by fact**

**show**  $(\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) \vdash_{\text{res}} \text{parallel-wiring } \triangleright \text{real1 } \eta \parallel \text{real2 } \eta \checkmark$

**and**  $(\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) \vdash_{\text{res}} \text{parallel-wiring } \triangleright \text{ideal1 } \eta \parallel \text{ideal2 } \eta \checkmark$

**and**  $\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta, \mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta \vdash_C \text{sim1 } \eta \models \text{sim2 } \eta$   
 $\checkmark$  **for**  $\eta$  **by**(*rule WT-intro*)**+**

**fix**  $\mathcal{A} :: \text{security} \implies (('a + 'g) + 'b + 'h, ('c + 'i) + 'd + 'j) \text{distinguisher}$

**assume** *WT* [*WT-intro*]:  $(\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) \vdash_g \mathcal{A} \eta \checkmark$  **for**  $\eta$

**assume** *bound* [*interaction-bound*]: *interaction-any-bounded-by*  $(\mathcal{A} \eta) (\text{min } (\text{bound1 } \eta) (\text{bound2 } \eta))$  **for**  $\eta$

**assume** *lossless* [*plossless-intro*]:  $\text{lossless1} \vee \text{lossless2} \implies \text{plossless-gpv } ((\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta)) (\mathcal{A} \eta)$  **for**  $\eta$

**let**  $?A = \lambda \eta. \text{absorb } (\mathcal{A} \eta) (\text{parallel-wiring } \odot \text{parallel-converter } (\text{converter-of-resource } (\text{real1 } \eta)) 1_C)$

**have**  $*:\mathcal{I}\text{-uniform } (\text{outs-}\mathcal{I} ((\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta)))$

$\text{UNIV}, (\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common1 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-inner2 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) \vdash_C$

$((1_C \models \text{sim2 } \eta) \models 1_C) \odot \text{parallel-wiring} \sim ((1_C \models \text{sim2 } \eta) \models 1_C \models 1_C) \odot \text{parallel-wiring}$  **for**  $\eta$

**by**(*rule eq-I-comp-cong*, *rule eq-I-converter-mono*)

(*auto simp add: le-I-def intro: parallel-converter2-eq-I-cong eq-I-converter-refl WT-converter-parallel-converter2*)

*WT-converter-id* *sec2*. *WT-sim parallel-converter2-id-id[symmetric]* *eq-I-converter-refl WT-parallel-wiring*)

**have**  $**:$   $\text{outs-}\mathcal{I} ((\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta)) \vdash_R$

$((1_C \models \text{sim2 } \eta) \models 1_C \models 1_C) \odot \text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{ideal2 } \eta \sim$

$\text{parallel-wiring} \odot (\text{converter-of-resource } (\text{real1 } \eta) \mid_{\infty} 1_C) \triangleright \text{sim2 } \eta \models 1_C \triangleright \text{ideal2 } \eta$  **for**  $\eta$

**unfolding** *comp-parallel-wiring*

**by**(*rule eq-resource-on-trans*, *rule eq-I-attach-on*[**where**  $\text{conv}' = \text{parallel-wiring}$ ])

$\odot (1_C \models \text{sim2 } \eta \models 1_C)$   
, (rule *WT-intro*)+, rule *eq-I-comp-cong*, rule *eq-I-converter-mono*)  
(auto simp add: *le-I-def attach-compose attach-parallel2 attach-converter-of-resource-conv-parallel-resource*  
intro: *WT-intro parallel-converter2-eq-I-cong parallel-converter2-id-id eq-I-converter-refl*)

**have** *ideal2*:  
connect ( $\mathcal{A} \eta$ ) ( $(1_C \models \text{sim2 } \eta) \models 1_C \triangleright \text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{ideal2 } \eta$ ) =  
connect ( $?A \eta$ ) ( $\text{sim2 } \eta \models 1_C \triangleright \text{ideal2 } \eta$ ) **for**  $\eta$   
**unfolding** *distinguish-attach[symmetric]*  
**proof** (rule *connect-eq-resource-cong[OF WT, rotated]*, *goal-cases*)  
**case**  $?2$   
**then show**  $?case$   
**by**(*subst attach-compose[symmetric]*, rule *eq-resource-on-trans*  
, rule *eq-I-attach-on[where conv'=((1\_C \models \text{sim2 } \eta) \models 1\_C \models 1\_C) \odot*  
*parallel-wiring]*)  
((rule *WT-intro*)+ | *intro \* | intro \*\**) +  
**qed** (rule *WT-intro*) +  
**have** *real2*: connect ( $\mathcal{A} \eta$ ) ( $\text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{real2 } \eta$ ) = connect ( $?A$   
 $\eta$ ) ( $\text{real2 } \eta$ ) **for**  $\eta$   
**unfolding** *distinguish-attach[symmetric]*  
**by**(*simp add: attach-compose attach-converter-of-resource-conv-parallel-resource*)  
**have**  $\mathcal{I}\text{-real2 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta \vdash_g ?A \eta \checkmark$  **for**  $\eta$  **by**(rule *WT-intro*) +  
**moreover have** *interaction-any-bounded-by* ( $?A \eta$ ) (*bound2*  $\eta$ ) **for**  $\eta$   
**by** *interaction-bound-converter simp*  
**moreover have** *plossless-gpv* ( $\mathcal{I}\text{-real2 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta$ ) ( $?A \eta$ ) **if** *lossless2*  
**for**  $\eta$   
**by**(rule *plossless-intro WT-intro | simp add: that*) +  
**ultimately**  
**have** *negl2*: *negligible* ( $\lambda \eta. \text{advantage } (\mathcal{A} \eta)$   
( $(1_C \models \text{sim2 } \eta) \models 1_C \triangleright \text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{ideal2 } \eta$ )  
( $\text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{real2 } \eta$ ))  
**unfolding** *advantage-def ideal2 real2* **by**(rule *sec2.adv[unfolded advantage-def]*)

**let**  $?A = \lambda \eta. \text{absorb } (\mathcal{A} \eta) (\text{parallel-wiring} \odot \text{parallel-converter } 1_C (\text{converter-of-resource}$   
( $\text{sim2 } \eta \models 1_C \triangleright \text{ideal2 } \eta$ )))  
**have** *ideal1*:  
connect ( $\mathcal{A} \eta$ ) ( $(\text{sim1 } \eta \models \text{sim2 } \eta) \models 1_C \triangleright \text{parallel-wiring} \triangleright \text{ideal1 } \eta \parallel \text{ideal2}$   
 $\eta$ ) =  
connect ( $?A \eta$ ) ( $\text{sim1 } \eta \models 1_C \triangleright \text{ideal1 } \eta$ ) **for**  $\eta$   
**proof** –  
**have**  $*$ :  $\mathcal{I}\text{-uniform}$  ( $(\text{outs-}\mathcal{I} (\mathcal{I}\text{-real1 } \eta) \langle + \rangle \text{outs-}\mathcal{I} (\mathcal{I}\text{-real2 } \eta)) \langle + \rangle \text{outs-}\mathcal{I}$   
( $\mathcal{I}\text{-common1 } \eta$ )  $\langle + \rangle$   
 $\text{outs-}\mathcal{I} (\mathcal{I}\text{-common2 } \eta)$ ) *UNIV*, ( $\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common1 } \eta$ )  $\oplus_{\mathcal{I}}$  ( $\mathcal{I}\text{-inner2}$   
 $\eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta$ )  $\vdash_C$   
( $(\text{sim1 } \eta \models \text{sim2 } \eta) \models 1_C$ )  $\odot \text{parallel-wiring} \sim ((\text{sim1 } \eta \models \text{sim2 } \eta) \models 1_C \models$   
 $1_C) \odot \text{parallel-wiring}$   
**by**(rule *eq-I-comp-cong*, rule *eq-I-converter-mono*)  
(auto simp add: *le-I-def comp-parallel-wiring' attach-compose attach-parallel2*  
*attach-converter-of-resource-conv-parallel-resource2*)

intro: WT-intro parallel-converter2-id-id[symmetric] eq-I-converter-refl  
parallel-converter2-eq-I-cong eq-I-converter-mono)

**have** \*\*:((outs-I (I-real1  $\eta$ ) <+> outs-I (I-real2  $\eta$ )) <+> outs-I (I-common1  $\eta$ ) <+> outs-I (I-common2  $\eta$ ))  $\vdash_R$

(sim1  $\eta$   $\models$  sim2  $\eta$ )  $\models$   $1_C \triangleright$  parallel-wiring  $\triangleright$  ideal1  $\eta$   $\parallel$  ideal2  $\eta \sim$

parallel-wiring  $\odot$  ( $1_C \mid_{\alpha}$  converter-of-resource (sim2  $\eta$   $\models$   $1_C \triangleright$  ideal2  $\eta$ ))  $\triangleright$   
sim1  $\eta$   $\models$   $1_C \triangleright$  ideal1  $\eta$

**unfolding** attach-compose[symmetric]

**by**(rule eq-resource-on-trans, rule eq-I-attach-on[**where** conv'=((sim1  $\eta$   $\models$   
sim2  $\eta$ )  $\models$   $1_C \mid_{\alpha}$   $1_C$ )  $\odot$  parallel-wiring])

((rule WT-intro)+  $\mid$  intro \*  $\mid$  auto simp add: le-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 (A  $\eta$ ) (( $1_C \mid_{\alpha}$  sim2  $\eta$ )  $\models$   $1_C \triangleright$  parallel-wiring  $\triangleright$  real1  $\eta$   $\parallel$  ideal2  $\eta$ ) =  
connect (?A  $\eta$ ) (real1  $\eta$ ) **for**  $\eta$

**proof** –

**have** \*\*: I-uniform (outs-I ((I-real1  $\eta$   $\oplus_I$  I-real2  $\eta$ )  $\oplus_I$  (I-common1  $\eta$   $\oplus_I$   
I-common2  $\eta$ )))

UNIV, (I-real1  $\eta$   $\oplus_I$  I-common1  $\eta$ )  $\oplus_I$  (I-inner2  $\eta$   $\oplus_I$  I-common2  $\eta$ )  $\vdash_C$

(( $1_C \mid_{\alpha}$  sim2  $\eta$ )  $\models$   $1_C$ )  $\odot$  parallel-wiring  $\sim$  (( $1_C \mid_{\alpha}$  sim2  $\eta$ )  $\models$   $1_C \mid_{\alpha}$   $1_C$ )

$\odot$  parallel-wiring

**by**(rule eq-I-comp-cong, rule eq-I-converter-mono)

(auto simp add: le-I-def intro: WT-intro parallel-converter2-eq-I-cong  
WT-converter-parallel-converter2

parallel-converter2-id-id[symmetric] eq-I-converter-refl WT-parallel-wiring)

**have** \*: outs-I ((I-real1  $\eta$   $\oplus_I$  I-real2  $\eta$ )  $\oplus_I$  (I-common1  $\eta$   $\oplus_I$  I-common2  
 $\eta$ ))  $\vdash_R$

parallel-wiring  $\odot$  (( $1_C \mid_{\alpha}$   $1_C$ )  $\models$  sim2  $\eta$   $\models$   $1_C$ )  $\triangleright$  real1  $\eta$   $\parallel$  ideal2  $\eta \sim$

parallel-wiring  $\odot$  ( $1_C \mid_{\alpha}$  converter-of-resource (sim2  $\eta$   $\models$   $1_C \triangleright$  ideal2  $\eta$ ))  $\triangleright$   
real1  $\eta$

**by**(rule eq-resource-on-trans, rule eq-I-attach-on[**where** conv'=parallel-wiring  
 $\odot$  ( $1_C \mid_{\alpha}$  sim2  $\eta$   $\models$   $1_C$ )])

, (rule WT-intro)+, rule eq-I-comp-cong, rule eq-I-converter-mono)

(auto simp add: le-I-def attach-compose attach-converter-of-resource-conv-parallel-resource2  
attach-parallel2

intro: WT-intro parallel-converter2-eq-I-cong parallel-converter2-id-id  
eq-I-converter-refl)

**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<sub>C</sub> |= sim2 η) |= 1<sub>C</sub> |= 1<sub>C</sub>) ⊙*  
*parallel-wiring ▷ 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 ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -common1 η ⊢<sub>g</sub> ? $\mathcal{A}$  η √* **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 ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -common1 η) (? $\mathcal{A}$  η)* **if** *lossless1*  
**for** η

**by**(*rule plossless-intro WT-intro | simp add: that*)  
**ultimately**

**have** *negl1: negligible (λη. advantage ( $\mathcal{A}$  η))*

*((sim1 η |= sim2 η) |= 1<sub>C</sub> ▷ parallel-wiring ▷ ideal1 η || ideal2 η)*

*((1<sub>C</sub> |= sim2 η) |= 1<sub>C</sub> ▷ parallel-wiring ▷ real1 η || ideal2 η)*

**unfolding** *advantage-def ideal1 real1* **by**(*rule sec1.adv[unfolded advantage-def]*)

**from** *negligible-plus[OF negl1 negl2]*

**show** *negligible (λη. advantage ( $\mathcal{A}$  η) ((sim1 η |= sim2 η) |= 1<sub>C</sub> ▷ parallel-wiring*  
*▷ ideal1 η || ideal2 η)*

*(parallel-wiring ▷ real1 η || real2 η))*

**by**(*rule negligible-mono*) (*auto simp add: advantage-def intro!: eventuallyI lan-*  
*dau-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: ∧ $\mathcal{D}$ . [∧η.  $\mathcal{I}$ -inner1 η ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -common1 η ⊢<sub>g</sub>  $\mathcal{D}$  η √;*

*∧η. interaction-any-bounded-by ( $\mathcal{D}$  η) (bound1 η);*

*∧η. lossless1 ⇒ plossless-gpv ( $\mathcal{I}$ -inner1 η ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -common1 η) ( $\mathcal{D}$  η) ]*

*⇒ (∃η. wiring ( $\mathcal{I}$ -inner1 η) ( $\mathcal{I}$ -real1 η) (cnv1 η) (w1 η)) ∧*

*negligible (λη. advantage ( $\mathcal{D}$  η) (ideal1 η) (cnv1 η |= 1<sub>C</sub> ▷ real1 η))*

**by blast**

**from** *sec2.correct* **obtain** *cnv2*

**where** *correct2: ∧ $\mathcal{D}$ . [∧η.  $\mathcal{I}$ -inner2 η ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -common2 η ⊢<sub>g</sub>  $\mathcal{D}$  η √;*

*∧η. interaction-any-bounded-by ( $\mathcal{D}$  η) (bound2 η);*

*∧η. lossless2 ⇒ plossless-gpv ( $\mathcal{I}$ -inner2 η ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -common2 η) ( $\mathcal{D}$  η) ]*

*⇒ (∃η. wiring ( $\mathcal{I}$ -inner2 η) ( $\mathcal{I}$ -real2 η) (cnv2 η) (w2 η)) ∧*

*negligible (λη. advantage ( $\mathcal{D}$  η) (ideal2 η) (cnv2 η |= 1<sub>C</sub> ▷ real2 η))*

**by blast**

**show** *∃cnv. ∃ $\mathcal{D}$ . (∃η. ( $\mathcal{I}$ -inner1 η ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -inner2 η) ⊕ <sub>$\mathcal{I}$</sub>  ( $\mathcal{I}$ -common1 η ⊕ <sub>$\mathcal{I}$</sub>*   
 *$\mathcal{I}$ -common2 η) ⊢<sub>g</sub>  $\mathcal{D}$  η √) →*

*(∃η. interaction-any-bounded-by ( $\mathcal{D}$  η) (min (bound1 η) (bound2 η))) →*

*(∃η. lossless1 ∨ lossless2 → plossless-gpv (( $\mathcal{I}$ -inner1 η ⊕ <sub>$\mathcal{I}$</sub>   $\mathcal{I}$ -inner2 η) ⊕ <sub>$\mathcal{I}$</sub>*



$(\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) (\mathcal{D } \eta) \longrightarrow$   
 $(\forall \eta. \text{wiring } (\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta) (\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta) (\text{cnv } \eta)$   
 $(w1 \eta \mid_w w2 \eta)) \wedge$   
 $\text{negligible } (\lambda \eta. \text{advantage } (\mathcal{D } \eta) (\text{parallel-wiring } \triangleright \text{ideal1 } \eta \parallel \text{ideal2 } \eta) (\text{cnv}$   
 $\eta \mid_{= 1_C} \triangleright \text{parallel-wiring } \triangleright \text{real1 } \eta \parallel \text{real2 } \eta))$   
**proof**(*intro exI strip conjI*)  
**fix**  $\mathcal{D} :: \text{security} \Rightarrow (('e + 'k) + 'b + 'h, ('f + 'l) + 'd + 'j) \text{distinguisher}$   
**assume**  $\text{WT-D [rule-format, WT-intro]: } \forall \eta. (\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta) \oplus_{\mathcal{I}}$   
 $(\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) \vdash_g \mathcal{D } \eta \checkmark$   
**and**  $\text{bound [rule-format, interaction-bound]: } \forall \eta. \text{interaction-any-bounded-by}$   
 $(\mathcal{D } \eta) (\text{min } (\text{bound1 } \eta) (\text{bound2 } \eta))$   
**and**  $\text{lossless [rule-format, plossless-intro]: } \forall \eta. \text{lossless1 } \vee \text{lossless2} \longrightarrow \text{ploss-}$   
 $\text{less-gpv } ((\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta)) (\mathcal{D } \eta)$   
 $\eta)$   
**let**  $?cnv = \lambda \eta. \text{cnv1 } \eta \mid_{= \text{cnv2 } \eta}$   
**let**  $?D1 = \lambda \eta. \text{absorb } (\mathcal{D } \eta) (\text{parallel-wiring } \odot \text{parallel-converter } 1_C (\text{converter-of-resource}$   
 $(\text{ideal2 } \eta)))$   
**have**  $\text{WT1: } \mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common1 } \eta \vdash_g ?D1 \eta \checkmark$  **for**  $\eta$  **by**(*rule WT-intro*)**+**  
**have**  $\text{bound1: interaction-any-bounded-by } (?D1 \eta) (\text{bound1 } \eta)$  **for**  $\eta$  **by** *inter-*  
*action-bound simp*  
**have**  $\text{plossless-gpv } (\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common1 } \eta) (?D1 \eta)$  **if**  $\text{lossless1}$  **for**  $\eta$   
**by**(*rule plossless-intro WT-intro | simp add: that*)**+**  
**from**  $\text{correct1[OF WT1 bound1 this]}$   
**have**  $w1: \text{wiring } (\mathcal{I}\text{-inner1 } \eta) (\mathcal{I}\text{-real1 } \eta) (\text{cnv1 } \eta) (w1 \eta)$   
**and**  $\text{adv1: negligible } (\lambda \eta. \text{advantage } (?D1 \eta) (\text{ideal1 } \eta) (\text{cnv1 } \eta \mid_{= 1_C} \triangleright \text{real1}$   
 $\eta))$  **for**  $\eta$   
**by** *auto*  
**from**  $w1$  **obtain**  $f g$  **where**  $fg: \wedge \eta. w1 \eta = (f \eta, g \eta)$   
**and**  $[\text{WT-intro}]: \wedge \eta. \mathcal{I}\text{-inner1 } \eta, \mathcal{I}\text{-real1 } \eta \vdash_C \text{cnv1 } \eta \checkmark$   
**and**  $\text{eq1: } \wedge \eta. \mathcal{I}\text{-inner1 } \eta, \mathcal{I}\text{-real1 } \eta \vdash_C \text{cnv1 } \eta \sim \text{map-converter id id } (f \eta)$   
 $(g \eta) 1_C$   
**apply** *atomize-elim*  
**apply**(*fold all-conj-distrib*)  
**apply**(*subst choice-iff[symmetric]*)**+**  
**apply**(*fastforce elim!: wiring.cases*)  
**done**  
**have**  $\mathcal{I}1: \mathcal{I}\text{-inner1 } \eta \leq \text{map-}\mathcal{I} (f \eta) (g \eta) (\mathcal{I}\text{-real1 } \eta)$  **for**  $\eta$   
**using**  $\text{eq-}\mathcal{I}\text{-converterD-WT1[OF eq1]}$  **by**(*rule WT-map-converter-idD*)(*rule*  
 $\text{WT-intro}$ )  
**with**  $\text{WT-converter-id order-refl}$  **have**  $[\text{WT-intro}]: \mathcal{I}\text{-inner1 } \eta, \text{map-}\mathcal{I} (f \eta) (g$   
 $\eta) (\mathcal{I}\text{-real1 } \eta) \vdash_C 1_C \checkmark$  **for**  $\eta$   
**by**(*rule WT-converter-mono*)  
**have**  $\text{lossless1 [plossless-intro]: plossless-converter } (\mathcal{I}\text{-inner1 } \eta) (\mathcal{I}\text{-real1 } \eta)$   
 $(\text{map-converter id id } (f \eta) (g \eta) 1_C)$  **for**  $\eta$   
**by**(*rule plossless-map-converter*)(*rule plossless-intro order-refl I1 WT-intro*  
 $\text{plossless-converter-mono | simp}$ )**+**

**let**  $?D2 = \lambda\eta. \text{absorb } (\mathcal{D} \eta) (\text{parallel-wiring} \odot \text{parallel-converter } (\text{converter-of-resource } (\text{map-converter } \text{id id } (f \eta) (g \eta) 1_C \models 1_C \triangleright \text{real1 } \eta)) 1_C)$   
**have**  $WT2: \mathcal{I}\text{-inner2 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta \vdash g ?D2 \eta \checkmark$  **for**  $\eta$  **by**  $(\text{rule } WT\text{-intro} \mid \text{simp})+$   
**have**  $\text{bound2: interaction-any-bounded-by } (?D2 \eta) (\text{bound2 } \eta)$  **for**  $\eta$  **by**  $\text{interaction-bound simp}$   
**have**  $\text{plossless-gpv } (\mathcal{I}\text{-inner2 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta) (?D2 \eta)$  **if**  $\text{lossless2}$  **for**  $\eta$   
**by**  $(\text{rule } \text{plossless-intro } WT\text{-intro} \mid \text{simp add: that})+$   
**from**  $\text{correct2}[OF WT2 \text{bound2 this}]$   
**have**  $w2: \text{wiring } (\mathcal{I}\text{-inner2 } \eta) (\mathcal{I}\text{-real2 } \eta) (\text{cnv2 } \eta) (w2 \eta)$   
**and**  $\text{adv2: negligible } (\lambda\eta. \text{advantage } (?D2 \eta) (\text{ideal2 } \eta) (\text{cnv2 } \eta \models 1_C \triangleright \text{real2 } \eta))$  **for**  $\eta$   
**by**  $\text{auto}$

**from**  $w2$  **have**  $[WT\text{-intro}]: \mathcal{I}\text{-inner2 } \eta, \mathcal{I}\text{-real2 } \eta \vdash_C \text{cnv2 } \eta \checkmark$  **for**  $\eta$  **by**  $\text{cases}$

**have**  $*$ :  $\text{connect } (\mathcal{D} \eta) (?cnv \eta \models 1_C \triangleright \text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{real2 } \eta) = \text{connect } (?D2 \eta) (\text{cnv2 } \eta \models 1_C \triangleright \text{real2 } \eta)$  **for**  $\eta$   
**proof**  $-$   
**have**  $\text{outs-}\mathcal{I} ((\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta)) \vdash_R$   
 $?cnv \eta \models 1_C \triangleright \text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{real2 } \eta \sim$   
 $(\text{map-converter } \text{id id } (f \eta) (g \eta) 1_C \models \text{cnv2 } \eta) \models (1_C \models 1_C) \triangleright \text{parallel-wiring}$   
 $\triangleright \text{real1 } \eta \parallel \text{real2 } \eta$   
**by**  $(\text{rule } \text{eq-}\mathcal{I}\text{-attach-on' } WT\text{-intro } \text{parallel-converter2-eq-}\mathcal{I}\text{-cong } \text{eq1 } \text{eq-}\mathcal{I}\text{-converter-refl } \text{parallel-converter2-id-id}[\text{symmetric}]) + \text{simp}$   
**also** **have**  $(\text{map-converter } \text{id id } (f \eta) (g \eta) 1_C \models \text{cnv2 } \eta) \models (1_C \models 1_C) \triangleright$   
 $\text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{real2 } \eta =$   
 $\text{parallel-wiring} \triangleright (\text{map-converter } \text{id id } (f \eta) (g \eta) 1_C \models 1_C \triangleright \text{real1 } \eta) \parallel$   
 $(\text{cnv2 } \eta \models 1_C \triangleright \text{real2 } \eta)$   
**by**  $(\text{simp add: comp-parallel-wiring' attach-compose attach-parallel2})$   
**finally** **show**  $?thesis$   
**by**  $(\text{auto intro!: connect-eq-resource-cong}[OF WT-D] \text{intro: } WT\text{-intro simp add: distinguish-attach}[\text{symmetric}] \text{attach-compose attach-converter-of-resource-conv-parallel-resource})$   
**qed**

**have**  $**$ :  $\text{connect } (?D2 \eta) (\text{ideal2 } \eta) = \text{connect } (?D1 \eta) (\text{cnv1 } \eta \models 1_C \triangleright \text{real1 } \eta)$  **for**  $\eta$   
**proof**  $-$   
**have**  $\text{connect } (?D2 \eta) (\text{ideal2 } \eta) =$   
 $\text{connect } (\mathcal{D} \eta) (\text{parallel-wiring} \triangleright ((\text{map-converter } \text{id id } (f \eta) (g \eta) 1_C \models 1_C) \models 1_C) \triangleright (\text{real1 } \eta \parallel \text{ideal2 } \eta))$   
**by**  $(\text{simp add: distinguish-attach}[\text{symmetric}] \text{attach-converter-of-resource-conv-parallel-resource attach-compose attach-parallel2})$   
**also** **have**  $\dots = \text{connect } (\mathcal{D} \eta) (\text{parallel-wiring} \triangleright ((\text{cnv1 } \eta \models 1_C) \models 1_C) \triangleright (\text{real1 } \eta \parallel \text{ideal2 } \eta))$   
**unfolding**  $\text{attach-compose}[\text{symmetric}]$  **using**  $WT\text{-D}$   
**by**  $(\text{rule } \text{connect-eq-resource-cong}[\text{symmetric}])$

(rule eq- $\mathcal{I}$ -attach-on' WT-intro eq- $\mathcal{I}$ -comp-cong eq- $\mathcal{I}$ -converter-refl parallel-converter2-eq- $\mathcal{I}$ -cong eq1 | simp)+  
**also have** ... = connect (? $\mathcal{D}1$   $\eta$ ) (cnv1  $\eta$  |<sub>=</sub> 1<sub>C</sub>  $\triangleright$  real1  $\eta$ )  
**by**(simp add: distinguish-attach[symmetric] attach-converter-of-resource-conv-parallel-resource2 attach-compose attach-parallel2)  
**finally show** ?thesis .  
**qed**

**have** \*\*\*: connect (? $\mathcal{D}1$   $\eta$ ) (ideal1  $\eta$ ) = connect ( $\mathcal{D}$   $\eta$ ) (parallel-wiring  $\triangleright$  ideal1  $\eta$  || ideal2  $\eta$ ) **for**  $\eta$

**by**(auto intro!: connect-eq-resource-cong[OF WT-D] simp add: attach-converter-of-resource-conv-parallel-resource2 distinguish-attach[symmetric] attach-compose intro: WT-intro)

**show** wiring ( $\mathcal{I}$ -inner1  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -inner2  $\eta$ ) ( $\mathcal{I}$ -real1  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -real2  $\eta$ ) (?cnv  $\eta$ ) (w1  $\eta$  |<sub>w</sub> w2  $\eta$ ) **for**  $\eta$   
**using** w1 w2 **by**(rule wiring-parallel-converter2)  
**from** negligible-plus[OF adv1 adv2]  
**show** negligible ( $\lambda\eta$ . advantage ( $\mathcal{D}$   $\eta$ ) (parallel-wiring  $\triangleright$  ideal1  $\eta$  || ideal2  $\eta$ ) (?cnv  $\eta$  |<sub>=</sub> 1<sub>C</sub>  $\triangleright$  parallel-wiring  $\triangleright$  real1  $\eta$  || real2  $\eta$ ))  
**by**(rule negligible-le)(simp add: advantage-def \* \*\* \*\*\*)  
**qed**  
**qed**

**theorem** (in constructive-security) parallel-realisation1:

**assumes** WT-res:  $\bigwedge\eta$ .  $\mathcal{I}$ -res  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common'  $\eta$   $\vdash$  res res  $\eta$   $\surd$

**and** lossless-res:  $\bigwedge\eta$ . lossless  $\implies$  lossless-resource ( $\mathcal{I}$ -res  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common'  $\eta$ ) (res  $\eta$ )

**shows** constructive-security ( $\lambda\eta$ . parallel-wiring  $\triangleright$  res  $\eta$  || real-resource  $\eta$ )

( $\lambda\eta$ . parallel-wiring  $\triangleright$  (res  $\eta$  || ideal-resource  $\eta$ )) ( $\lambda\eta$ . parallel-converter2 id-converter (sim  $\eta$ ))

( $\lambda\eta$ .  $\mathcal{I}$ -res  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -real  $\eta$ ) ( $\lambda\eta$ .  $\mathcal{I}$ -res  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -ideal  $\eta$ ) ( $\lambda\eta$ .  $\mathcal{I}$ -common'  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\eta$ ) bound lossless ( $\lambda\eta$ . (id, id) |<sub>w</sub> w  $\eta$ )

**by**(rule parallel-constructive-security[OF constructive-security-trivial[where lossless=False and bound= $\lambda$ -.  $\infty$ , OF WT-res], simplified, OF - lossless-res])

unfold-locales

**theorem** (in constructive-security) parallel-realisation2:

**assumes** WT-res:  $\bigwedge\eta$ .  $\mathcal{I}$ -res  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common'  $\eta$   $\vdash$  res res  $\eta$   $\surd$

**and** lossless-res:  $\bigwedge\eta$ . lossless  $\implies$  lossless-resource ( $\mathcal{I}$ -res  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common'  $\eta$ ) (res  $\eta$ )

**shows** constructive-security ( $\lambda\eta$ . parallel-wiring  $\triangleright$  real-resource  $\eta$  || res  $\eta$ )

( $\lambda\eta$ . parallel-wiring  $\triangleright$  (ideal-resource  $\eta$  || res  $\eta$ )) ( $\lambda\eta$ . parallel-converter2 (sim  $\eta$ ) id-converter)

( $\lambda\eta$ .  $\mathcal{I}$ -real  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -res  $\eta$ ) ( $\lambda\eta$ .  $\mathcal{I}$ -ideal  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -res  $\eta$ ) ( $\lambda\eta$ .  $\mathcal{I}$ -common  $\eta$   $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common'  $\eta$ ) bound lossless ( $\lambda\eta$ . w  $\eta$  |<sub>w</sub> (id, id))

**by**(rule parallel-constructive-security[OF - constructive-security-trivial[where lossless=False and bound= $\lambda$ -.  $\infty$ , OF WT-res], simplified, OF - lossless-res])

unfold-locales

**theorem** (in *constructive-security*) *parallel-resource1*:  
**assumes** *WT-res* [*WT-intro*]:  $\bigwedge \eta. \mathcal{I}\text{-res } \eta \vdash_{\text{res}} \text{res } \eta \checkmark$   
**and** *lossless-res* [*plossless-intro*]:  $\bigwedge \eta. \text{lossless} \implies \text{lossless-resource } (\mathcal{I}\text{-res } \eta)$  (*res*  $\eta$ )  
**shows** *constructive-security* ( $\lambda \eta. \text{parallel-resource1-wiring } \triangleright \text{res } \eta \parallel \text{real-resource } \eta$ )  
( $\lambda \eta. \text{parallel-resource1-wiring } \triangleright \text{res } \eta \parallel \text{ideal-resource } \eta$ ) *sim*  
 $\mathcal{I}\text{-real } \mathcal{I}\text{-ideal } (\lambda \eta. \mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta)$  *bound lossless w*  
**proof**  
**fix**  $\mathcal{A} :: \text{security} \implies ('a + 'g + 'e, 'b + 'h + 'f)$  *distinguisher*  
**assume** *WT* [*WT-intro*]:  $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) \vdash_g \mathcal{A} \eta \checkmark$  **for**  $\eta$   
**assume** *bound* [*interaction-bound*]: *interaction-any-bounded-by* ( $\mathcal{A} \eta$ ) (*bound*  $\eta$ )  
**for**  $\eta$   
**assume** *lossless* [*plossless-intro*]: *lossless*  $\implies$  *plossless-gpv* ( $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta)$ ) ( $\mathcal{A} \eta$ ) **for**  $\eta$

**let**  $?A = \lambda \eta. \text{absorb } (\mathcal{A} \eta)$  (*swap-lassocr*  $\odot$  *parallel-converter* (*converter-of-resource* (*res*  $\eta$ ))  $1_C$ )  
**have** *ideal*:  
*connect* ( $\mathcal{A} \eta$ ) (*sim*  $\eta \models 1_C \triangleright \text{parallel-resource1-wiring } \triangleright \text{res } \eta \parallel \text{ideal-resource } \eta$ ) =  
*connect* ( $?A \eta$ ) (*sim*  $\eta \models 1_C \triangleright \text{ideal-resource } \eta$ ) **for**  $\eta$   
**proof** –  
**have**[*intro*]: *I-uniform* (*outs-I* ( $\mathcal{I}\text{-real } \eta$ )  $\langle + \rangle$  *outs-I* ( $\mathcal{I}\text{-res } \eta$ )  $\langle + \rangle$  *outs-I* ( $\mathcal{I}\text{-common } \eta$ ))  
 $\text{UNIV}, \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) \vdash_C \text{sim } \eta \models 1_C \sim \text{sim } \eta \models 1_C \models 1_C$   
**by**(*rule eq-I-converter-mono*) (*auto simp add: le-I-def intro!*:  
 $\text{WT-intro parallel-converter2-id-id[symmetric] parallel-converter2-eq-I-cong eq-I-converter-refl}$ )

**have**  $*$ : *outs-I* ( $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta)$ )  $\vdash_R (\text{sim } \eta \models 1_C) \odot$   
*parallel-resource1-wiring*  $\triangleright \text{res } \eta \parallel \text{ideal-resource } \eta \sim$   
*swap-lassocr*  $\odot$  (*converter-of-resource* (*res*  $\eta$ )  $\mid_{\infty} 1_C$ )  $\triangleright \text{sim } \eta \models 1_C \triangleright$   
*ideal-resource*  $\eta$   
**by** (*rule eq-resource-on-trans*[**where**  $\text{res}' = (\text{sim } \eta \models 1_C \models 1_C) \odot \text{parallel-resource1-wiring } \triangleright \text{res } \eta \parallel \text{ideal-resource } \eta$ ],  
*rule eq-I-attach-on*, (*rule WT-intro*) $+$ , *rule eq-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-I-converter-refl*)

**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*:

$connect (\mathcal{A} \eta) (parallel-resource1-wiring \triangleright res \eta \parallel real-resource \eta) =$   
 $connect (? \mathcal{A} \eta) (real-resource \eta) \text{ for } \eta$   
**unfolding** *distinguish-attach[symmetric]*  
**by** (*simp add: attach-compose attach-converter-of-resource-conv-parallel-resource*  
*parallel-resource1-wiring-def*)  
**have**  $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g ? \mathcal{A} \eta \checkmark \text{ for } \eta$  **by** (*rule WT-intro*) +  
**moreover have** *interaction-any-bounded-by* ( $? \mathcal{A} \eta$ ) (*bound*  $\eta$ ) **for**  $\eta$   
**by** *interaction-bound-converter simp*  
**moreover have** *plossless-gpv* ( $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta$ ) ( $? \mathcal{A} \eta$ ) **if lossless for**  
 $\eta$   
**by** (*rule plossless-intro WT-intro | simp add: that*) +  
**ultimately show** *negligible* ( $\lambda \eta. advantage (\mathcal{A} \eta) (sim \eta \models 1_C \triangleright$   
 $parallel-resource1-wiring \triangleright res \eta \parallel ideal-resource \eta)$   
 $(parallel-resource1-wiring \triangleright res \eta \parallel real-resource \eta))$ )  
**unfolding** *advantage-def ideal real* **by** (*rule adv[unfolded advantage-def]*)  
**next**  
**from correct obtain** *cnv*  
**where** *correct'*:  $\bigwedge \mathcal{D}. \llbracket \bigwedge \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{D} \eta \checkmark;$   
 $\bigwedge \eta. interaction-any-bounded-by (\mathcal{D} \eta) (bound \eta);$   
 $\bigwedge \eta. lossless \implies plossless-gpv (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D} \eta) \rrbracket$   
 $\implies (\forall \eta. wiring (\mathcal{I}\text{-ideal } \eta) (\mathcal{I}\text{-real } \eta) (cnv \eta) (w \eta)) \wedge$   
 $negligible (\lambda \eta. advantage (\mathcal{D} \eta) (ideal-resource \eta) (cnv \eta \models 1_C \triangleright$   
 $real-resource \eta))$   
**by** *blast*  
**show**  $\exists cnv. \forall \mathcal{D}. (\forall \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) \vdash_g \mathcal{D} \eta \checkmark) \longrightarrow$   
 $(\forall \eta. interaction-any-bounded-by (\mathcal{D} \eta) (bound \eta)) \longrightarrow$   
 $(\forall \eta. lossless \longrightarrow plossless-gpv (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta)) (\mathcal{D}$   
 $\eta)) \longrightarrow$   
 $(\forall \eta. wiring (\mathcal{I}\text{-ideal } \eta) (\mathcal{I}\text{-real } \eta) (cnv \eta) (w \eta)) \wedge$   
 $negligible (\lambda \eta. advantage (\mathcal{D} \eta) (parallel-resource1-wiring \triangleright res \eta \parallel ideal-resource$   
 $\eta)$   
 $(cnv \eta \models 1_C \triangleright parallel-resource1-wiring \triangleright res \eta \parallel real-resource \eta))$   
**proof** (*intro exI conjI strip*)  
**fix**  $\mathcal{D} :: security \implies ('c + 'g + 'e, 'd + 'h + 'f) distinguisher$   
**assume** *WT-D [rule-format, WT-intro]*:  $\forall \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}$   
 $\eta) \vdash_g \mathcal{D} \eta \checkmark$   
**and** *bound [rule-format, interaction-bound]*:  $\forall \eta. interaction-any-bounded-by$   
 $(\mathcal{D} \eta) (bound \eta)$   
**and** *lossless [rule-format, plossless-intro]*:  $\forall \eta. lossless \longrightarrow plossless-gpv (\mathcal{I}\text{-ideal}$   
 $\eta \oplus_{\mathcal{I}} (\mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta)) (\mathcal{D} \eta)$   
  
**let**  $? \mathcal{D} = \lambda \eta. absorb (\mathcal{D} \eta) (swap-lassocr \odot parallel-converter (converter-of-resource$   
 $(res \eta)) 1_C)$   
**have** *WT'*:  $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g ? \mathcal{D} \eta \checkmark \text{ for } \eta$  **by** (*rule WT-intro*) +  
**have** *bound'*: *interaction-any-bounded-by* ( $? \mathcal{D} \eta$ ) (*bound*  $\eta$ ) **for**  $\eta$  **by** *interac-*  
*tion-bound simp*  
**have** *lossless'*: *plossless-gpv* ( $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta$ ) ( $? \mathcal{D} \eta$ ) **if lossless for**  
 $\eta$   
**by** (*rule plossless-intro WT-intro that*) +

```

from correct'[OF WT' bound' lossless]
have w: wiring (I-ideal η) (I-real η) (cnv η) (w η)
  and adv: negligible (λη. advantage (?D η) (ideal-resource η) (cnv η |= 1C ▷
real-resource η))
  for η by auto
show wiring (I-ideal η) (I-real η) (cnv η) (w η) for η by(rule w)
from w have [WT-intro]: I-ideal η, I-real η ⊢C cnv η √ for η by cases
have connect (D η) (swap-lassocr ▷ res η || (cnv η |= 1C ▷ real-resource η))
=
  connect (D η) (cnv η |= 1C ▷ swap-lassocr ▷ res η || real-resource η) for η
proof –
  have connect (D η) (cnv η |= 1C ▷ swap-lassocr ▷ res η || real-resource η) =
    connect (D η) (cnv η |= 1C |= 1C ▷ swap-lassocr ▷ res η || real-resource η)
  by(rule connect-eq-resource-cong[OF WT-D])
    (rule eq-I-attach-on' WT-intro parallel-converter2-eq-I-cong eq-I-converter-refl
parallel-converter2-id-id[symmetric] | simp)+
  also have ... = connect (D η) (swap-lassocr ▷ res η || (cnv η |= 1C ▷
real-resource η))
  by(simp add: comp-swap-lassocr' attach-compose attach-parallel2)
  finally show ?thesis by simp
qed
with adv show negligible (λη. advantage (D η) (parallel-resource1-wiring ▷ res
η || ideal-resource η)
  (cnv η |= 1C ▷ parallel-resource1-wiring ▷ 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**  
   $key-oracle\ None\ () = do\ \{ k \leftarrow key-gen; return-spmf\ (k, Some\ k)\}$   
   $| key-oracle\ (Some\ x)\ () = return-spmf\ (x, Some\ x)$

**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**  $Forward-auth :: auth-query\ \mathbf{where}\ Forward-auth \equiv ForwardOrEdit\ ()$

**abbreviation**  $Forward-insec :: 'a\ insec-query\ \mathbf{where}\ Forward-insec \equiv ForwardOrEdit\ None$

**abbreviation**  $Edit :: 'a \Rightarrow 'a\ insec-query\ \mathbf{where}\ Edit\ m \equiv ForwardOrEdit\ (Some\ m)$

**adhoc-overloading**  $Forward \Rightarrow Forward-auth$

**adhoc-overloading**  $Forward \Rightarrow Forward-insec$

**translations**

$(logic)\ CONST\ Forward \leq (logic)\ CONST\ ForwardOrEdit\ (CONST\ None)$

$(logic)\ CONST\ Forward \leq (logic)\ CONST\ ForwardOrEdit\ (CONST\ Product-Type.Unity)$

$(type)\ auth-query \leq (type)\ unit\ aquery$

$(type)\ 'a\ insec-query \leq (type)\ 'a\ option\ aquery$

### 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  $\equiv$  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  $\Rightarrow$  '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 (λm. 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 ())) (λc'.
  case c' of Inr (Some c) ⇒ (do {
    k ← Pause (Inl ()) Done;
    Done (dec-alg (projl k) c) })
  | - ⇒ Done None)
)) ()

```

**definition**  $\pi^E :: (\text{auth-query}, 'c \text{ option}, \text{auth-query}, 'c \text{ option}) \text{ converter } (\langle \pi^E \rangle)$   
**where**  
 $\pi^E \equiv 1_C$

**definition**  $\text{routing} \equiv (1_C \mid = \text{lassocr}_C) \odot \text{swap-lassocr} \odot (1_C \mid = (1_C \mid = \text{swap-lassocr}) \odot \text{swap-lassocr}) \odot \text{rassocl}_C$

**definition**  $\text{res} = (1_C \mid = \text{enc} \mid = \text{dec}) \triangleright (1_C \mid = \text{parallel-wiring}) \triangleright \text{parallel-resource1-wiring} \triangleright (\text{KEY.res} \parallel \text{AUTH.res})$

**lemma**  $\text{res-alt-def}$ :  $\text{res} = ((1_C \mid = \text{enc} \mid = \text{dec}) \odot (1_C \mid = \text{parallel-wiring})) \triangleright \text{parallel-resource1-wiring} \triangleright (\text{KEY.res} \parallel \text{AUTH.res})$   
**by**(*simp add: res-def attach-compose*)

**end**

## 8.5 Message authentication converter

**locale**  $\text{macode} =$   
*INSEC*: *insec-channel* + *RO*: *rorc range*  
**for**  $\text{range} :: 'r \text{ set} +$   
**fixes**  $\text{mac-alg} :: 'r \Rightarrow 'm \Rightarrow 'a \text{ spmf}$   
**begin**

**definition**  $\text{enm} :: ('m, \text{unit}, 'm + ('a \times 'm), 'r + \text{unit}) \text{ converter}$  **where**  
 $\text{enm} \equiv \text{CNV } (\lambda bs \ m. \text{if } bs$   
 then  $\text{Done } ((), \text{True})$   
 else do {  
 $r \leftarrow \text{Pause } (\text{Inl } m) \text{ Done};$   
 $a \leftarrow \text{lift-spmf } (\text{mac-alg } (\text{projl } r) m);$   
 $(- :: 'r + \text{unit}) \leftarrow \text{Pause } (\text{Inr } (a, m)) \text{ Done};$   
 $\text{Done } ((), \text{True})$   
 })  $\text{False}$

**definition**  $\text{dem} :: (\text{unit}, 'm \text{ option}, 'm + \text{unit}, 'r + ('a \times 'm) \text{ option}) \text{ converter}$   
**where**

```

dem ≡ CNV (stateless-callee (λ-. Pause (Inr ())) (λam'.
  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**  $\pi^E :: (('a \times 'm) \text{ insec-query}, ('a \times 'm) \text{ option}, ('a \times 'm) \text{ insec-query}, ('a \times 'm) \text{ option}) \text{ converter } (\langle \pi^E \rangle)$  **where**  
 $\pi^E \equiv 1_C$

**definition**  $\text{routing} \equiv (1_C \mid = \text{lassocr}_C) \odot \text{swap-lassocr} \odot (1_C \mid = (1_C \mid = \text{swap-lassocr}) \odot \text{swap-lassocr}) \odot \text{rassoctr}_C$

**definition**  $\text{res} = (1_C \mid = \text{enm} \mid = \text{dem}) \triangleright (1_C \mid = \text{parallel-wiring}) \triangleright \text{parallel-resource1-wiring} \triangleright (\text{RO.res} \parallel \text{INSEC.res})$

**end**

**lemma** *interface-wiring*:

$(\text{cnv-addr} \mid = \text{cnv-send} \mid = \text{cnv-recv}) \triangleright (1_C \mid = \text{parallel-wiring}) \triangleright \text{parallel-resource1-wiring}$

$\triangleright$

$(\text{RES}(\text{res2-send} \oplus_O \text{res2-recv}) \text{res2-s} \parallel \text{RES}(\text{res1-addr} \oplus_O \text{res1-send} \oplus_O \text{res1-recv}) \text{res1-s})$

$=$

$\text{cnv-addr} \mid = \text{cnv-send} \mid = \text{cnv-recv} \triangleright$

$\text{RES}(\dagger \text{res1-addr} \oplus_O (\text{res2-send}\dagger \oplus_O \dagger \text{res1-send}) \oplus_O \text{res2-recv}\dagger \oplus_O \dagger \text{res1-recv})$   
 $(\text{res2-s}, \text{res1-s})$

$(\text{is} - \triangleright ?L1 \triangleright ?L2 \triangleright ?L3 = - \triangleright ?R)$

**proof** –

**let**  $?wiring = (\text{id}, \text{id}) \mid_w (\text{lassocr}_w \circ_w ((\text{id}, \text{id}) \mid_w (\text{rassoctr}_w \circ_w (\text{swap}_w \mid_w (\text{id}, \text{id}) \circ_w \text{lassocr}_w)))$

$\circ_w \text{rassoctr}_w)) \circ_w (\text{rassoctr}_w \circ_w (\text{swap}_w \mid_w (\text{id}, \text{id}) \circ_w \text{lassocr}_w))$

**have**  $?L1 \triangleright ?L2 \triangleright ?L3 = ?L1 \odot ?L2 \triangleright$

$\text{RES}((\text{res2-send}\dagger \oplus_O \text{res2-recv}\dagger) \oplus_O \dagger \text{res1-addr} \oplus_O \dagger \text{res1-send} \oplus_O \dagger \text{res1-recv})$   
 $(\text{res2-s}, \text{res1-s}) (\text{is} - = - \triangleright \text{RES}(?O) ?S)$

**unfolding** *attach-compose[symmetric] resource-of-parallel-oracle[symmetric]*

**by** *(simp only: parallel-oracle-cnv-plus-oracle extend-state-oracle-plus-oracle extend-state-oracle2-plus-oracle)*

**also have**  $\dots = \text{RES}(\text{apply-wiring } ?wiring ?O) ?S$

**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**  $\dots = ?R$  **by** *simp*

**finally show**  $?thesis$  **by** *(rule arg-cong2[where f=attach, OF refl])*

**qed**

**definition**  $\text{id}'$  **where**  $\text{id}' = \text{id}$

end

## 9 Security of one-time-pad encryption

**theory** *One-Time-Pad* **imports**

*System-Construction*

**begin**

**definition** *key* :: *security*  $\Rightarrow$  *bool list spmf* **where**

*key*  $\eta \equiv$  *spmf-of-set* (*nlists UNIV*  $\eta$ )

**definition** *enc* :: *security*  $\Rightarrow$  *bool list*  $\Rightarrow$  *bool list*  $\Rightarrow$  *bool list spmf* **where**

*enc*  $\eta$  *k m*  $\equiv$  *return-spmf* (*k*  $[\oplus]$  *m*)

**definition** *dec* :: *security*  $\Rightarrow$  *bool list*  $\Rightarrow$  *bool list*  $\Rightarrow$  *bool list option* **where**

*dec*  $\eta$  *k c*  $\equiv$  *Some* (*k*  $[\oplus]$  *c*)

**definition** *sim* :: *'b list option*  $\Rightarrow$  *'a*  $\Rightarrow$  (*'b list option*  $\times$  *'b list option*, *'a*, *nat option*) *gpv* **where**

*sim* *c q*  $\equiv$  (*do* {  
  *lo*  $\leftarrow$  *Pause* *q Done*;  
  (*case* *lo* *of*  
    *Some n*  $\Rightarrow$  *if* *c = None*  
      *then* *do* {  
        *x*  $\leftarrow$  *lift-spmf* (*spmf-of-set* (*nlists UNIV n*));  
        *Done* (*Some x*, *Some x*)  
      *else Done* (*c*, *c*)  
    | *None*  $\Rightarrow$  *Done* (*None*, *c*))})

**context**

**fixes**  $\eta$  :: *security*

**begin**

**private definition** *key-channel-send* :: *bool list option*  $\times$  *bool list cstate*

$\Rightarrow$  *bool list*  $\Rightarrow$  (*unit*  $\times$  *bool list option*  $\times$  *bool list cstate*) *spmf* **where**

*key-channel-send* *s m*  $\equiv$  *do* {  
  (*k*, *s*)  $\leftarrow$  (*key.key-oracle* (*key*  $\eta$ )) $\dagger$  *s* ();  
  *c*  $\leftarrow$  *enc*  $\eta$  *k m*;  
  (*-*, *s*)  $\leftarrow$   $\dagger$ *channel.send-oracle* *s c*;  
  *return-spmf* (*()*, *s*)}

**private definition** *key-channel-recv* :: *bool list option*  $\times$  *bool list cstate*

$\Rightarrow$  *'a*  $\Rightarrow$  (*bool list option*  $\times$  *bool list option*  $\times$  *bool list cstate*) *spmf* **where**

*key-channel-recv* *s m*  $\equiv$  *do* {  
  (*c*, *s*)  $\leftarrow$   $\dagger$ *channel.recv-oracle* *s* ();  
  (*case* *c* *of* *None*  $\Rightarrow$  *return-spmf* (*None*, *s*)  
  | *Some c'*  $\Rightarrow$  *do* {  
    (*k*, *s*)  $\leftarrow$  (*key.key-oracle* (*key*  $\eta$ )) $\dagger$  *s* ();  
    *return-spmf* (*dec*  $\eta$  *k c'*, *s*)})}

**private abbreviation** *callee-sec-channel* **where**

*callee-sec-channel* *callee*  $\equiv$  *lift-state-oracle extend-state-oracle (attach-callee callee sec-channel.sec-oracle)*

**private inductive**  $S :: (\text{bool list option} \times \text{unit} \times \text{bool list cstate}) \text{ spmf} \Rightarrow$

$(\text{bool list option} \times \text{bool list cstate}) \text{ spmf} \Rightarrow \text{bool}$  **where**

$S$  (*return-spmf* (*None*, (), *Void*))

(*return-spmf* (*None*, *Void*))

|  $S$  (*return-spmf* (*None*, (), *Store plain*))

(*map-spmf* ( $\lambda \text{key. (Some key, Store (key } [\oplus] \text{ plain}))}$ ) (*spmf-of-set* (*nlists UNIV*  $\eta$ )))

**if** *length plain* = *id'  $\eta$*

|  $S$  (*return-spmf* (*None*, (), *Collect plain*))

(*map-spmf* ( $\lambda \text{key. (Some key, Collect (key } [\oplus] \text{ plain}))}$ ) (*spmf-of-set* (*nlists UNIV*  $\eta$ )))

**if** *length plain* = *id'  $\eta$*

|  $S$  (*return-spmf* (*Some (key } [\oplus] \text{ plain), (), Store plain*))

(*return-spmf* (*Some key, Store (key } [\oplus] \text{ plain}))*

**if** *length plain* = *id'  $\eta$*  *length key* = *id'  $\eta$*  **for** *key*

|  $S$  (*return-spmf* (*Some (key } [\oplus] \text{ plain), (), Collect plain*))

(*return-spmf* (*Some key, Collect (key } [\oplus] \text{ plain}))*

**if** *length plain* = *id'  $\eta$*  *length key* = *id'  $\eta$*  **for** *key*

|  $S$  (*return-spmf* (*None*, (), *Fail*))

(*map-spmf* ( $\lambda x. (\text{Some } x, \text{Fail})$ ) (*spmf-of-set* (*nlists UNIV*  $\eta$ )))

|  $S$  (*return-spmf* (*Some (key } [\oplus] \text{ plain), (), Fail*))

(*return-spmf* (*Some key, Fail*))

**if** *length plain* = *id'  $\eta$*  *length key* = *id'  $\eta$*  **for** *key plain*

**lemma** *resources-indistinguishable*:

**shows** (*UNIV*  $\langle + \rangle$  *nlists UNIV (id'  $\eta$ )*  $\langle + \rangle$  *UNIV*)  $\vdash_R$

*RES (callee-sec-channel sim*  $\oplus_O$   $\dagger\dagger$ *channel.send-oracle*  $\oplus_O$   $\dagger\dagger$ *channel.recv-oracle*)  
(*None* :: *bool list option*, (), *Void*)

$\approx$

*RES* ( $\dagger$ *auth-channel.auth-oracle*  $\oplus_O$  *key-channel-send*  $\oplus_O$  *key-channel-recv*)  
(*None* :: *bool list option*, *Void*)

(**is**  $?A \vdash_R$  *RES* ( $?L1 \oplus_O ?L2 \oplus_O ?L3$ )  $?SL \approx$  *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** \*:  $?A \vdash_C ?L1 \oplus_O ?L2 \oplus_O ?L3(?SL) \approx ?R1 \oplus_O ?R2 \oplus_O ?R3(?SR)$

**proof**(*rule trace'-eqI-sim[where S=S]*, *goal-cases Init-OK Output-OK State-OK*)

```

case Init-OK
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)) (spm-of-set (nlists UNIV (id' η))) =
            map-spmf (λx. Inl (Some x)) (map-spmf (λx. x [⊕] plain) (spm-of-set
              (nlists UNIV η))) for η
          unfolding id'-def by (subst xor-list-commute, subst one-time-pad[where
            xs=plain, symmetric]) simp-all

          from Store-State-Channel show ?thesis using Output-OK(2-) Inl Look
            by(simp add: sim-def, simp add: map-spmf-conv-bind-spmf[symmetric])
              (subst (2) spmf.map-comp[where f=λx. Inl (Some x), symmetric,
                unfolded o-def], simp only: *)
          qed (auto simp add: sim-def)
          qed (auto simp add: sim-def id'-def elim: S.cases)
        next
        case Snd-Rcv: (Inr query')
        with Output-OK show ?thesis
        proof (cases query')
          case (Inr rcv-query)
          with Output-OK Snd-Rcv show ?thesis
          proof cases
            case Collect-State-Channel: (3 plain)
            then show ?thesis using Output-OK(2-) Snd-Rcv Inr
              by (simp cong: bind-spmf-cong-simp add: in-nlists-UNIV id'-def)
            qed simp-all
            qed (auto elim: S.cases)
          qed
        next
        case (State-OK p q query state answer state')
        then show ?case
        proof (cases query)
          case (Inl adv-query)
          with State-OK show ?thesis
          proof (cases adv-query)
            case Look
            with State-OK Inl show ?thesis

```

```

proof cases
  case Store-State-Channel: (2 plain)
  have *: length plain = id' η ⇒ key ∈ nlists UNIV η ⇒
    S (cond-spmf-fst (map-spmf (λx. (Inl (Some x), Some x, (), Store plain))
      (spmf-of-set (nlists UNIV (id' η)))) (Inl (Some (key [⊕] plain))))
      (cond-spmf-fst (map-spmf (λx. (Inl (Some (x [⊕] plain)), Some x, Store
(x [⊕] plain))))
      (spmf-of-set (nlists UNIV η)))) (Inl (Some (key [⊕] plain)))) for key
  proof(subst (1 2) cond-spmf-fst-map-Pair1, goal-cases)
  case 2
  note inj-onD[OF inj-on-xor-list-nlists, rotated, simplified xor-list-commute]
  with 2 show ?case
    unfolding inj-on-def by (auto simp add: id'-def)
  next
  case 5
  note inj-onD[OF inj-on-xor-list-nlists, rotated, simplified xor-list-commute]
  with 5 show ?case
    by (subst (1 2 3) inv-into-f-f)
      ((clarsimp simp add: inj-on-def), (auto simp add: S.simps id'-def
inj-on-def in-nlists-UNIV ))
    qed (simp-all add: id'-def in-nlists-UNIV min-def inj-on-def)
    from Store-State-Channel show ?thesis using State-OK(2-) Inl Look
    by (clarsimp simp add: sim-def) (simp add: map-spmf-conv-bind-spmf[symmetric]
* )
    qed (auto simp add: sim-def map-spmf-conv-bind-spmf[symmetric] S.simps)
    qed (erule S.cases; (simp add: sim-def, auto simp add: map-spmf-conv-bind-spmf[symmetric]
S.simps))+
  next
  case Snd-Rcv: (Inr query')
  with State-OK show ?thesis
  proof (cases query')
    case (Inr rcv-query)
    with State-OK Snd-Rcv show ?thesis
  proof cases
    case Collect-State-Channel: (3 plain)
    then show ?thesis using State-OK(2-) Snd-Rcv Inr
    by clarsimp (simp add: S.simps in-nlists-UNIV id'-def map-spmf-conv-bind-spmf[symmetric]
cong: bind-spmf-cong-simp)
    qed (simp add: sim-def, auto simp add: map-spmf-conv-bind-spmf[symmetric]
S.simps)
    qed (erule S.cases,
      (simp add: sim-def, auto simp add: map-spmf-conv-bind-spmf[symmetric]
S.simps in-nlists-UNIV))
    qed
    qed

  from * show ?thesis by simp
qed

```

**lemma** *real-resource-wiring*:  
**shows** *cipher.res* (key  $\eta$ ) (enc  $\eta$ ) (dec  $\eta$ )  
= *RES* ( $\dagger$ 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. $\pi E$ -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= $\lambda$ 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*)  $\models 1_C \triangleright$  *channel.res sec-channel.sec-oracle*  
= *RES* (*callee-sec-channel callee*  $\oplus_O$   $\dagger\dagger$ *channel.send-oracle*  $\oplus_O$   $\dagger\dagger$ *channel.recv-oracle*  
) (*s*, *()*, *Void*) (**is** *?L1  $\triangleright$  - = ?R*)  
**proof** –  
**have**[*simp*]:  *$\mathcal{I}$ -full,  $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$  ( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full)  $\vdash_C$  ?L1  $\sim$  ?L1* (**is** *-*, *?I  $\vdash_C$  -  $\sim$*   
*-*)  
**by**(*rule eq- $\mathcal{I}$ -converter-mono*)  
(*rule parallel-converter2-eq- $\mathcal{I}$ -cong eq- $\mathcal{I}$ -converter-reflI WT-converter- $\mathcal{I}$ -full*  
 *$\mathcal{I}$ -full-le-plus- $\mathcal{I}$  order-refl plus- $\mathcal{I}$ -mono*)+

**have**[*simp*]: *?I  $\vdash_c$  (sec-channel.sec-oracle  $\oplus_O$  channel.send-oracle  $\oplus_O$  chan-*  
*nel.recv-oracle) s  $\surd$  for s*  
**by**(*rule WT-plus-oracleI WT-parallel-oracle WT-callee-full; (unfold split-paired-all)?*)+

**have**[*simp*]: *?L1  $\triangleright$  RES (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 ?I' = ?I and ?conv = ?L1*  
*and ?conv' = ?L1]) simp-all*  
**qed**

**end**



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

**lemma** *eq-I-gpv-Done2*:  
 $eq\text{-I-gpv } A \ \mathcal{I} \ gpv \ (Done \ x) \longleftrightarrow \text{lossless-spmf } (the\text{-gpv } gpv) \wedge (\forall a \in \text{set-spmf } (the\text{-gpv } gpv). eq\text{-I-generat } A \ \mathcal{I} \ (eq\text{-I-gpv } A \ \mathcal{I}) \ a \ (Pure \ x))$   
**by**(*auto intro: eq-I-gpv.intros simp add: rel-spmf-return-spmf2 elim: eq-I-gpv.cases*)

**context begin**

**interpretation** *CIPHER*: *cipher key  $\eta$  enc  $\eta$  dec  $\eta$  for  $\eta$  .*

**interpretation** *S-CHAN*: *sec-channel .*

**lemma** *one-time-pad*:

**defines**  $\mathcal{I}\text{-real} \equiv \lambda\eta. \mathcal{I}\text{-uniform } UNIV \ (insert \ None \ (Some \ 'nlists \ UNIV \ \eta))$   
**and**  $\mathcal{I}\text{-ideal} \equiv \lambda\eta. \mathcal{I}\text{-uniform } UNIV \ \{None, \ Some \ \eta\}$   
**and**  $\mathcal{I}\text{-common} \equiv \lambda\eta. \mathcal{I}\text{-uniform } (nlists \ UNIV \ \eta) \ UNIV \ \oplus_{\mathcal{I}} \ \mathcal{I}\text{-uniform } UNIV$   
*(insert None (Some 'nlists UNIV  $\eta$ ))*

**shows**

*constructive-security2 CIPHER.res ( $\lambda\cdot$ . S-CHAN.res) ( $\lambda\cdot$ . CNV sim None)*  
*I-real I-ideal I-common ( $\lambda\cdot$ .  $\infty$ ) False ( $\lambda\cdot$ . auth-sec-wiring)*

**proof**

**let**  $?I\text{-key} = \lambda\eta. \mathcal{I}\text{-uniform } UNIV \ (nlists \ UNIV \ \eta)$   
**let**  $?I\text{-enc} = \lambda\eta. \mathcal{I}\text{-uniform } (nlists \ UNIV \ \eta) \ UNIV$   
**let**  $?I\text{-dec} = \lambda\eta. \mathcal{I}\text{-uniform } UNIV \ (insert \ None \ (Some \ 'nlists \ UNIV \ \eta))$   
**have** *i1* [*WT-intro*]:  $\mathcal{I}\text{-uniform } (nlists \ UNIV \ \eta) \ UNIV, \ ?I\text{-key } \eta \ \oplus_{\mathcal{I}} \ ?I\text{-enc } \eta$   
 $\vdash_C \ CIPHER.\text{enc } \eta \ \surd \ \text{for } \eta$   
**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*]:  $?I\text{-dec } \eta, \ ?I\text{-key } \eta \ \oplus_{\mathcal{I}} \ ?I\text{-dec } \eta \ \vdash_C \ CIPHER.\text{dec } \eta \ \surd \ \text{for } \eta$   
**unfolding** *CIPHER.dec-def* **by**(*rule WT-converter-of-callee*)(*auto simp add: stateless-callee-def dec-def in-nlists-UNIV*)  
**have** [*WT-intro*]:  $\mathcal{I}\text{-common } \eta, \ (?I\text{-key } \eta \ \oplus_{\mathcal{I}} \ ?I\text{-enc } \eta) \ \oplus_{\mathcal{I}} \ (?I\text{-key } \eta \ \oplus_{\mathcal{I}} \ ?I\text{-dec } \eta)$   
 $\vdash_C \ CIPHER.\text{enc } \eta \ \models \ CIPHER.\text{dec } \eta \ \surd \ \text{for } \eta$   
**unfolding** *I-common-def* **by**(*rule WT-intro*)  
**have** *key*: *callee-invariant-on* (*CIPHER.KEY.key-oracle*  $\eta \ \oplus_O \ CIPHER.KEY.key-oracle$   
 $\eta$ ) (*pred-option* ( $\lambda x. \text{length } x = \eta$ ))  
 $(?I\text{-key } \eta \ \oplus_{\mathcal{I}} \ ?I\text{-key } \eta) \ \text{for } \eta$   
**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*:  $?I\text{-key } \eta \ \oplus_{\mathcal{I}} \ ?I\text{-key } \eta \ \vdash_{res} \ CIPHER.KEY.res \ \eta \ \surd \ \text{for } \eta$   
**unfolding** *CIPHER.KEY.res-def* **by**(*rule callee-invariant-on.WT-resource-of-oracle[OF key]*) *simp*

```

let ?I = λη. pred-cstate (λx. length x = η)
have WT-auth:  $\mathcal{I}$ -real η ⊢ c CIPHER.AUTH.auth-oracle s √ 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 ⊢ c S-CHAN.recv-oracle s √ if pred-cstate (λx. length x
= η) s for η s
    using that by(cases s)(auto intro!: WT-calleeI simp add: in-nlists-UNIV)
  have WT-send: ? $\mathcal{I}$ -enc η ⊢ c S-CHAN.send-oracle s √ for η s
    by(rule WT-calleeI)(auto)
  have *: callee-invariant-on (CIPHER.AUTH.auth-oracle ⊕O S-CHAN.send-oracle
⊕O S-CHAN.recv-oracle) (?I η)
    ( $\mathcal{I}$ -real η ⊕ $\mathcal{I}$   $\mathcal{I}$ -common η) for η
  apply unfold-locales
  subgoal for s x y s'
    by(cases x; cases (s, proj1 x) rule: CIPHER.AUTH.auth-oracle.cases; cases
proj1 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 ⊕ $\mathcal{I}$   $\mathcal{I}$ -common η ⊢ res
CIPHER.AUTH.res √ for η
  unfolding CIPHER.AUTH.res-def by(rule callee-invariant-on.WT-resource-of-oracle[OF
*]) simp
  show cipher:  $\mathcal{I}$ -real η ⊕ $\mathcal{I}$   $\mathcal{I}$ -common η ⊢ res CIPHER.res η √ for η
    unfolding CIPHER.res-def by(rule WT-intro i3)+

show secure:  $\mathcal{I}$ -ideal η ⊕ $\mathcal{I}$   $\mathcal{I}$ -common η ⊢ res S-CHAN.res √ for η
proof –
  have[simp]: $\mathcal{I}$ -ideal η ⊢ c S-CHAN.sec-oracle s √ if ?I η s for s
  proof (cases rule: WT-calleeI, goal-cases)
    case (1 call ret s')
    then show ?case using that by (cases (s, call) rule: S-CHAN.sec-oracle.cases)
(simp-all add:  $\mathcal{I}$ -ideal-def)
  qed
  have[simp]:  $\mathcal{I}$ -common η ⊢ c (S-CHAN.send-oracle ⊕O S-CHAN.recv-oracle) s
√
  if pred-cstate (λx. length x = η) s for s
  unfolding  $\mathcal{I}$ -common-def by(rule WT-plus-oracleI WT-send WT-recv that)+

  have *: callee-invariant-on (S-CHAN.sec-oracle ⊕O S-CHAN.send-oracle ⊕O
S-CHAN.recv-oracle) (?I η) ( $\mathcal{I}$ -ideal η ⊕ $\mathcal{I}$   $\mathcal{I}$ -common η)
  apply(unfold-locales)
  subgoal for s x y s'
    by(cases (s, proj1 x) rule: S-CHAN.sec-oracle.cases; cases proj1 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]: I-real η, I-ideal η ⊢C CNV sim s √ if s ≠ None  $\longrightarrow$ 
length (the s) = η for s η
  using that by(coinduction arbitrary: s)(auto simp add: sim-def I-ideal-def
I-real-def in-nlists-UNIV)
  show I-real η, I-ideal η ⊢C CNV sim None √ for η by(rule sim) simp

{ fix A :: security  $\Rightarrow$  (auth-query + bool list + unit, bool list option + unit +
bool list option) distinguisher
  assume WT: ∧η. I-real η ⊕I I-common η ⊢g A η √
  and bound: ∧η. interaction-bounded-by (λ-. True) (A η) ∞
  have connect (A η) (CNV sim None |= 1C ▷ S-CHAN.res) = connect (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 |=
1C ▷ S-CHAN.res ≈ CIPHER.res η
    unfolding ideal-resource-wiring real-resource-wiring
    by(rule resources-indistinguishable)
    show outs-gpv (I-real η ⊕I I-common η) (A η) ⊆ UNIV <+> nlists UNIV
(id' η) <+> UNIV
    using WT[of η, THEN WT-gpv-outs-gpv]
    by(auto simp add: I-real-def I-common-def id'-def)
    show I-real η ⊕I I-common η ⊢res CIPHER.res η √ by(rule cipher)
    show I-real η ⊕I I-common η ⊢res CNV sim None |= 1C ▷ S-CHAN.res √
    by(rule WT-intro secure | simp)+
  qed
  then show negligible (λη. advantage (A η) (CNV sim None |= 1C ▷ S-CHAN.res)
(CIPHER.res η))
    by(simp add: advantage-def)
  next
  let ?cnv = map-converter id id auth-query-of sec-response-of 1C
  :: (auth-query, nat option, auth-query, bool list option) converter
  have [simp]: I-full, map-I id (map-option length) I-full ⊢C 1C √
  using WT-converter-id order-refl
  by(rule WT-converter-mono)(simp add: le-I-def)
  have WT [WT-intro]: I-ideal η, I-real η ⊢C ?cnv √ for η
  by(rule WT-converter-map-converter)(auto simp add: I-ideal-def I-real-def in-
tro!: WT-converter-mono[OF WT-converter-id order-refl] simp add: le-I-def in-nlists-UNIV)
  with eq-I-converter-refl[OF this]
  have wiring (I-ideal η) (I-real η) ?cnv auth-sec-wiring for η ..
  moreover
  have eq: I-ideal η, I-ideal η ⊢C map-converter id (map-option length) id id
(CNV sim s) ~ 1C
  if s ≠ None  $\longrightarrow$  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-I-gpv-Pause simp add: I-ideal-def in-nlists-UNIV eq-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 I-ideal  $\eta$ , I-ideal  $\eta \vdash_C ?cnv \odot CNV \text{ sim None } \surd$  for  $\eta$  by(rule WT
WT-intro)+ simp
  with - have wiring (I-ideal  $\eta$ ) (I-ideal  $\eta$ ) (?cnv  $\odot$  CNV sim None) (id, id)
for  $\eta$ 
  by(rule wiring.intros)(auto simp add: comp-converter-map-converter1 comp-converter-id-left
eq)
  ultimately show  $\exists cnv. \forall \eta. \text{wiring (I-ideal } \eta) \text{ (I-real } \eta) \text{ (cnv } \eta) \text{ auth-sec-wiring}$ 
 $\wedge$ 
    wiring (I-ideal  $\eta$ ) (I-ideal  $\eta$ ) (cnv  $\eta \odot$  CNV sim None) (id, id)
  by meson
}
qed

end

end

```

## 10 Security of message authentication

**theory** Message-Authentication-Code **imports**

System-Construction

**begin**

**definition** rnd :: security  $\Rightarrow$  bool list set **where**

rnd  $\eta \equiv$  nlists UNIV  $\eta$

**definition** mac :: security  $\Rightarrow$  bool list  $\Rightarrow$  bool list  $\Rightarrow$  bool list spmf **where**

mac  $\eta$  r m  $\equiv$  return-spmf r

**definition** vld :: security  $\Rightarrow$  bool list set **where**

vld  $\eta \equiv$  nlists UNIV  $\eta$

**fun** valid-mac-query :: security  $\Rightarrow$  (bool list  $\times$  bool list) insec-query  $\Rightarrow$  bool **where**

valid-mac-query  $\eta$  (ForwardOrEdit (Some (a, m)))  $\longleftrightarrow$  a  $\in$  vld  $\eta \wedge$  m  $\in$  vld  $\eta$   
| valid-mac-query  $\eta$  - = True

**fun** sim :: ('b list  $\times$  'b list) option + unit  $\Rightarrow$  ('b list  $\times$  'b list) insec-query

$\Rightarrow$  (('b list  $\times$  'b list) option  $\times$  (('b list  $\times$  'b list) option + unit), auth-query, 'b list option) gpv **where**

sim (Inr ()) - = Done (None, Inr())  
| sim (Inl None) (Edit (a', m')) = do { -  $\leftarrow$  Pause Drop Done; Done (None, Inr ())}  
| sim (Inl (Some (a, m))) (Edit (a', m')) = (if a = a'  $\wedge$  m = m')

```

    then do { - ← 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**  $\eta :: \text{security}$

## begin

**private definition** *rorc-channel-send* ::  $((\text{bool} \times \text{unit}) \times (\text{bool list} \Rightarrow \text{bool list option}) \times (\text{bool list} \times \text{bool list}) \text{ cstate}, \text{bool list}, \text{unit}) \text{ oracle}'$  **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* ::  $((\text{bool} \times \text{unit}) \times (\text{bool list} \Rightarrow \text{bool list option}) \times (\text{bool list} \times \text{bool list}) \text{ cstate}, \text{unit}, \text{bool list option}) \text{ oracle}'$  **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*  $\eta$ );  
*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*  $\eta$ );  
*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'  $\eta$  ∧ length m = id'  $\eta$*   
| - ⇒ *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* ( $\lambda a.$  ((*True*, ()), [*m* ↦ *a*], *Store* (*a*, *m*))) (*spmf-of-set* (*nlists UNIV*  
 $\eta$ )))  
**if** *length m = id'  $\eta$*   
| *S* (*return-spmf* (*Collect m*, *None*, *Map.empty*))  
(*map-spmf* ( $\lambda a.$  ((*True*, ()), [*m* ↦ *a*], *Collect* (*a*, *m*))) (*spmf-of-set* (*nlists*  
*UNIV  $\eta$* )))  
**if** *length m = id'  $\eta$*   
| *S* (*return-spmf* (*Store m*, *None*, [*m* ↦ *a*]))  
(*return-spmf* ((*True*, ()), [*m* ↦ *a*], *Store* (*a*, *m*)))  
**if** *length m = id'  $\eta$  and length a = id'  $\eta$*   
| *S* (*return-spmf* (*Collect m*, *None*, [*m* ↦ *a*]))  
(*return-spmf* ((*True*, ()), [*m* ↦ *a*], *Collect* (*a*, *m*)))  
**if** *length m = id'  $\eta$  and length a = id'  $\eta$*   
| *S* (*return-spmf* (*Fail*, *None*, *Map.empty*))  
(*map-spmf* ( $\lambda a.$  ((*True*, ()), [*m* ↦ *a*], *Fail*)) (*spmf-of-set* (*nlists UNIV  $\eta$* )))

```

if length  $m = id' \eta$ 
|  $S$  (return-spmf (Fail, None, [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Fail))
if length  $m = id' \eta$  and length  $a = id' \eta$ 
|  $S$  (return-spmf (Collect  $m'$ , Some ( $a'$ ,  $m'$ ), Map.empty))
    (return-spmf ((False, ()), Map.empty, Collect ( $a'$ ,  $m'$ )))
if length  $m' = id' \eta$  and length  $a' = id' \eta$ 
|  $S$  (return-spmf (Collect  $m'$ , Some ( $a'$ ,  $m'$ ), [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Collect ( $a'$ ,  $m'$ )))
if length  $m = id' \eta$  and length  $a = id' \eta$  and length  $m' = id' \eta$  and length  $a' = id' \eta$ 
|  $S$  (return-spmf (Collect  $m'$ , Some ( $a'$ ,  $m'$ ), Map.empty))
    (map-spmf ( $\lambda x.$  ((True, ()), [m ↦ x], Collect ( $a'$ ,  $m'$ ))) (spmf-of-set (nlists UNIV  $\eta$ )))
if length  $m = id' \eta$  and length  $m' = id' \eta$  and length  $a' = id' \eta$ 
|  $S$  (map-spmf ( $\lambda x.$  (Fail, None, as( $m' \mapsto x$ ))) spmf-s)
    (map-spmf ( $\lambda x.$  ((False, ()), as( $m' \mapsto x$ ), Fail)) spmf-s)
if length  $m' = id' \eta$  and lossless-spmf spmf-s
|  $S$  (map-spmf ( $\lambda x.$  (Fail, None, as( $m' \mapsto x$ ))) spmf-s)
    (map-spmf ( $\lambda x.$  ((True, ()), as( $m' \mapsto x$ ), Fail)) spmf-s)
if length  $m' = id' \eta$  and lossless-spmf spmf-s
|  $S$  (return-spmf (Fail, None, [ $m' \mapsto a'$ ]))
    (map-spmf ( $\lambda x.$  ((True, ()), [m ↦ x,  $m' \mapsto a'$ ], Fail)) (spmf-of-set (nlists UNIV  $\eta$ )))
if length  $m = id' \eta$  and length  $m' = id' \eta$  and length  $a' = id' \eta$ 
|  $S$  (map-spmf ( $\lambda x.$  (Fail, None, [ $m' \mapsto x$ ])) (spmf-of-set (nlists UNIV  $\eta \cap \{x. x \neq a'\}$ )))
    (map-spmf ( $\lambda x.$  ((True, ()), [m ↦ fst x,  $m' \mapsto snd x$ ], Fail)) (spmf-of-set (nlists UNIV  $\eta \times$  nlists UNIV  $\eta \cap \{x. snd x \neq a'\}$ )))
if length  $m = id' \eta$  and length  $m' = id' \eta$ 
|  $S$  (map-spmf ( $\lambda x.$  (Fail, None, as( $m' \mapsto x$ ))) spmf-s)
    (map-spmf ( $\lambda p.$  ((True, ()), as( $m' \mapsto fst p$ ,  $m \mapsto snd p$ ), Fail)) (mk-lossless (pair-spmf spmf-s (spmf-of-set (nlists UNIV  $\eta$ ))))))
if length  $m = id' \eta$  and length  $m' = id' \eta$  and lossless-spmf spmf-s

```

**private lemma** trace-eq-lazy:

**assumes**  $\eta > 0$

**shows** (valid-insecQ  $\langle + \rangle$  nlists UNIV ( $id' \eta$ )  $\langle + \rangle$  UNIV)  $\vdash_R$

RES (lazy-channel-insec  $\oplus_O$  lazy-channel-send  $\oplus_O$  lazy-channel-recv) (Void, None, Map.empty)

$\approx$

RES ( $\dagger\dagger$ insec-channel.insec-oracle  $\oplus_O$  rorc-channel-send  $\oplus_O$  rorc-channel-recv) ((False, ()), Map.empty, Void)

(is  $?A \vdash_R$  RES ( $?L1 \oplus_O ?L2 \oplus_O ?L3$ )  $?SL \approx$  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



*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)  $\eta$   $\times$  nlists (UNIV::bool set)  $\eta \cap \{x. \text{snd } x \neq a'\} \neq \{\}$  (is ?aux1)
  and aux2: nlists (UNIV::bool set)  $\eta \cap \{x. x \neq a'\} \neq \{\}$  (is ?aux2) for  $a'$ 
proof -
  have  $\exists a. a \in \text{nlists } (UNIV::\text{bool set}) \eta \wedge a \neq a'$  for  $a'$ 
  proof (cases  $a' \in \text{nlists } (UNIV::\text{bool set}) \eta$ )
    case True
      show ?thesis
      proof (rule ccontr)
        have  $2 \wedge \eta = \text{card } (\text{nlists } (UNIV :: \text{bool set}) \eta)$  by (simp add: card-nlists)
        also assume  $\nexists a. a \in \text{nlists } UNIV \eta \wedge a \neq a'$ 
        then have  $\text{nlists } UNIV \eta = \{a'\}$  using True by blast
        then have  $\text{fct:card } (\text{nlists } (UNIV :: \text{bool set}) \eta) = \text{card } \{a'\}$  by simp
        also have  $\text{card } \{a'\} = 1$  by simp
        finally have  $\eta = 0$  by simp
        then show False using assms by blast
      qed
    next
      case False
        obtain  $a$  where  $\text{obt}: a \in \text{nlists } (UNIV::\text{bool set}) \eta$  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  $\text{o1}: a \in \text{nlists } (UNIV::\text{bool set}) \eta$  and  $\text{o2}: a \neq a'$  by
blast

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

  from  $\text{o1 } \text{o2}$  have  $a \in \{x. x \neq a'\}$  and  $a \in \text{nlists } UNIV \eta$  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
  next
  case (State-OK p q query state answer state')
  show ?case
  proof (cases query)
    case (Inl adv-query)
    show ?thesis
    proof (cases adv-query)
      case Look
      with State-OK Inl show ?thesis
    proof cases
      case Store-State-Channel: (2 m)
      have[simp]: a ∈ nlists UNIV η ⇒
      S (cond-spmf-fst (map-spmf (λx. (Inl (Some (x, m))), Store m, None, [m

```

```

↦ x]))
  (spmf-of-set (nlists UNIV η)) (Inl (Some (a, m))))
  (cond-spmf-fst (map-spmf (λx. (Inl (Some (x, m))), (True, ()), [m ↦ x],
Store (x, m)))
  (spmf-of-set (nlists UNIV η)) (Inl (Some (a, m)))) for a
proof(subst (1 2) cond-spmf-fst-map-Pair1, goal-cases)
  case 4
  then show ?case
    by(subst (1 2 3) inv-into-f-f, simp-all add: inj-on-def)
      (rule S.intros, simp-all add: Store-State-Channel in-nlists-UNIV
id'-def)
  qed (simp-all add: id'-def inj-on-def)

from Store-State-Channel show ?thesis using State-OK Inl Look
  by(clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])

qed (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro: S.intros)
next
case (ForwardOrEdit foe)
with State-OK Inl show ?thesis
proof (cases foe)
  case (Some am')
  obtain a' m' where am' = (a', m') by (cases am') simp
  with State-OK Inl ForwardOrEdit Some show ?thesis
  by cases (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro:
S.intros elim:S.cases)
  qed (erule S.cases, auto simp add: map-spmf-conv-bind-spmf[symmetric]
intro: S.intros)
next
case Drop
with State-OK Inl show ?thesis
  by cases (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro:
S.intros)
qed
next
case Snd-Rcv: (Inr query')
show ?thesis
proof (cases query')
  case (Inl snd-query)
  with State-OK Snd-Rcv show ?thesis
proof cases
  case 1
  with State-OK Snd-Rcv Inl show ?thesis
  by(clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
    (rule S.intros, simp add: in-nlists-UNIV)
next
case (8 m' a')
with State-OK Snd-Rcv Inl show ?thesis
  by(clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])

```

```

      (rule S.intros, simp add: in-nlists-UNIV)
    next
      case (11 m' spmf-s as)
      with State-OK Snd-Rcv Inl show ?thesis
        by(auto simp add: bind-bind-conv-pair-spmf split-def split: if-splits
option.splits intro!: S.intros)
        ((auto simp add: map-spmf-conv-bind-spmf[symmetric] intro!: S.intros),
simp only: id'-def in-nlists-UNIV)
      qed (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro: S.intros)
    next
      case (Inr rcv-query)
      with State-OK Snd-Rcv show ?thesis
      proof(cases)
        case (8 m' a')
        then show ?thesis using State-OK(2-) Snd-Rcv Inr
          by (auto simp add: map-spmf-conv-bind-spmf[symmetric] image-def
cond-spmf-fst-def vimage-def aux1 aux2 assms intro:S.intros)
      next
        case (9 m a m' a')
        then show ?thesis using State-OK(2-) Snd-Rcv Inr
          by (cases m = m') (auto simp add: map-spmf-conv-bind-spmf[symmetric]
cond-spmf-fst-def
vimage-def aux1 aux2 assms intro:S.intros split!: if-splits)
      next
        case (10 m m' a')
        show ?thesis
        proof (cases m = m')
          case True
          with 10 show ?thesis using State-OK(2-) Snd-Rcv Inr
          by (auto simp add: map-spmf-conv-bind-spmf[symmetric] cond-spmf-fst-def
vimage-def aux1 aux2 assms split!: if-split intro:S.intros)
        next
          case False
          have[simp]:  $a' \in \text{nlists UNIV } \eta \implies \text{nlists (UNIV :: \text{bool set}) } \eta \times \text{nlists UNIV } \eta \cap \{x. \text{snd } x = a'\} = \text{nlists UNIV } \eta \times \{a'\}$ 
          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  $\mathcal{I} \equiv \mathcal{I}\text{-uniform } (\text{Set.Collect } (\text{valid-mac-query } \eta)) (\text{insert None } (\text{Some } \langle \text{nlists UNIV } \eta \times \text{nlists UNIV } \eta \rangle)) \oplus_{\mathcal{I}}$ 
    ( $\mathcal{I}\text{-uniform } (\text{vld } \eta) \text{ UNIV } \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform UNIV } (\text{insert None } (\text{Some } \langle \text{vld } \eta \rangle))$ )
  assumes bound: interaction-bounded-by' ( $\lambda\cdot$ . True)  $\mathcal{A}$   $q$ 
    and lossless: plossless-gpv  $\mathcal{I}$   $\mathcal{A}$ 
    and WT:  $\mathcal{I} \vdash g \mathcal{A} \checkmark$ 
  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|
       $\leq q / \text{real } (2 \wedge \eta)$  (is ?LHS  $\leq -$ )
  proof -

    define lazy-channel-insec' where
      lazy-channel-insec' = ( $\dagger$ lazy-channel-insec :: (bool  $\times$  bool list cstate  $\times$  (bool list
       $\times$  bool list) option  $\times$  (bool list  $\Rightarrow$  bool list option),
      (bool list  $\times$  bool list) insec-query, (bool list  $\times$  bool list) option) oracle')

    define lazy-channel-send' where
      lazy-channel-send' = ( $\dagger$ lazy-channel-send :: (bool  $\times$  bool list cstate  $\times$  (bool list
       $\times$  bool list) option  $\times$  (bool list  $\Rightarrow$  bool list option),
      bool list, unit) oracle')

    define lazy-channel-recv' where
      lazy-channel-recv' = ( $\lambda$  ( $s :: \text{bool} \times \text{bool list cstate} \times (\text{bool list} \times \text{bool list}) \text{option}$ 
       $\times (\text{bool list} \Rightarrow \text{bool list option})$ ) ( $q :: \text{unit}$ ).
      (case  $s$  of
      ( $\text{flg}, \text{Collect } m, \text{None}, \text{as}$ )  $\Rightarrow$  return-spmf (Some  $m$ , ( $\text{flg}, \text{Fail}, \text{None}, \text{as}$ ))
      | ( $\text{flg}, \text{ms}, \text{Some } (a', m'), \text{as}$ )  $\Rightarrow$  (case  $\text{as } m'$  of
      None  $\Rightarrow$  do {
         $a \leftarrow \text{spmf-of-set } (\text{rnd } \eta)$ ;
        return-spmf (if  $a = a'$  then Some  $m'$  else None,  $\text{flg} \vee a = a', \text{Fail}, \text{None},$ 
         $\text{as } (m' := \text{Some } a)$ )}
      | Some  $a \Rightarrow$  return-spmf (if  $a = a'$  then Some  $m'$  else None,  $\text{flg}, \text{Fail}, \text{None},$ 
         $\text{as}$ ))
      | -  $\Rightarrow$  return-spmf (None,  $s$ ))

    define lazy-channel-recv-f' where

```

```

    lazy-channel-recv-f' = (λ (s :: bool × bool list cstate × (bool list × bool list)
option × (bool list ⇒ bool list option)) (q :: unit).
  (case s of
    (flg, Collect m, None, as) ⇒ return-spmf (Some m, (flg, Fail, None, as))
  | (flg, ms, Some (a', m'), as) ⇒ (case as m' of
    None ⇒ do {
      a ← spmf-of-set (rnd η);
      return-spmf (None, flg ∨ a = a', Fail, None, as (m' := Some a))}
    | Some a ⇒ return-spmf (if a = a' then Some m' else None, flg, Fail, None,
as))
  | - ⇒ return-spmf (None, s)))

```

**define game where**

```

  game = (λ(A :: ((bool list × bool list) insec-query + bool list + unit, (bool list
× bool list) option + unit + bool list option) distinguisher) recv-oracle. do {
    (b :: bool, (flg, ams, es, as)) ← (exec-gpv (lazy-channel-insec' ⊕O lazy-channel-send'
⊕O recv-oracle) A (False, Void, None, Map.empty));
    return-spmf (b, flg) }

```

```

have fact1: spmf (connect A (RES (lazy-channel-insec ⊕O lazy-channel-send
⊕O lazy-channel-recv-f) (Void, None, Map.empty))) True
  = spmf (connect A (RES (lazy-channel-insec' ⊕O lazy-channel-send' ⊕O
lazy-channel-recv-f') (False, Void, None, Map.empty))) True

```

**proof** –

```

  let ?orc-lft = lazy-channel-insec ⊕O lazy-channel-send ⊕O lazy-channel-recv-f
  let ?orc-rgt = lazy-channel-insec' ⊕O lazy-channel-send' ⊕O lazy-channel-recv-f'

```

```

have[simp]: rel-spmf (rel-prod (=) (λx y. x = snd y))
  (lazy-channel-insec s q) (lazy-channel-insec' (flg, s) q) for s flg q
  by (cases s) (simp add: lazy-channel-insec'-def spmf-rel-map rel-prod-sel
split-def option.rel-refl pmf.rel-refl split.option.split)

```

```

have[simp]: rel-spmf (rel-prod (=) (λx y. x = snd y))
  (lazy-channel-send s q) (lazy-channel-send' (flg, s) q) for s flg q

```

**proof** –

```

  obtain ams es as where s = (ams, es, as) by (cases s)

```

```

  then show ?thesis by (cases ams) (auto simp add: spmf-rel-map map-spmf-conv-bind-spmf
split-def lazy-channel-send'-def)

```

**qed**

```

have[simp]: rel-spmf (rel-prod (=) (λx y. x = snd y))
  (lazy-channel-recv-f s q) (lazy-channel-recv-f' (flg, s) q) for s flg q

```

**proof** –

```

  obtain ams es as where *: s = (ams, es, as) by (cases s)

```

```

  show ?thesis

```

```

  proof (cases es)

```

```

    case None

```

```

    with * show ?thesis by (cases ams) (simp-all add: lazy-channel-recv-f'-def)

```

**next**

```

    case (Some am)
    obtain a m where am = (a, m) by (cases am)
    with * show ?thesis by (cases ams) (simp-all add: lazy-channel-recv-f'-def
rel-spmf-bind-reflI split:option.split)
  qed
  qed

  have[simp]: rel-spmf (rel-prod (=) ( $\lambda x y. x = \text{snd } y$ ))
    ((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 (=) ( $\lambda x y. x = \text{snd } y$ )], 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 (=) ( $\lambda x y. x = \text{snd } y$ )]; auto)+
  qed

  have[simp]: rel-spmf ( $\lambda x y. \text{fst } x = \text{fst } y$ )
    (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 (=) ( $\lambda x y. x = \text{snd } y$ )])
      (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
spmfs-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: |spmfs (connect  $\mathcal{A}$  (RES (lazy-channel-insec'  $\oplus_O$  lazy-channel-send'
 $\oplus_O$  lazy-channel-recv-f') (False, Void, None, Map.empty))) True -
spmfs (connect  $\mathcal{A}$  (RES (lazy-channel-insec'  $\oplus_O$  lazy-channel-send'  $\oplus_O$  lazy-channel-recv')
(False, Void, None, Map.empty))) True|
     $\leq$  Sigma-Algebra.measure (measure-spmfs (game  $\mathcal{A}$  lazy-channel-recv-f')) {x.

```

```

snd x } (is | spmf ?L - - spmf ?R - | ≤ - )
proof -
  let ?orc-lft = lazy-channel-insec' ⊕○ lazy-channel-send' ⊕○ lazy-channel-recv-f'

  let ?orc-rgt = lazy-channel-insec' ⊕○ lazy-channel-send' ⊕○ 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* ( $\lambda(a, s1') (b, s2'). (fst\ s2' \longrightarrow fst\ s1') \wedge (\neg\ fst\ s2' \longrightarrow \neg\ fst\ s1' \wedge a = b \wedge s1' = s2')$ )  
(*?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**  $?I = \mathcal{I}\text{-uniform}$  (*Set.Collect* (*valid-mac-query*  $\eta$ ))  $UNIV \oplus_{\mathcal{I}}$  ( $\mathcal{I}\text{-uniform}$  (*vld*  $\eta$ ))  $UNIV \oplus_{\mathcal{I}}$   $\mathcal{I}\text{-full}$   
**let**  $?A = mk\text{-lossless-gpv}$  (*responses-I*  $\mathcal{I}$ ) *True*  $\mathcal{A}$

**have** *plossless-gpv*  $?I$   $?A$  **using** *lossless WT*  
**by**(*rule plossless-gpv-mk-lossless-gpv*)(*simp add: I-def*)  
**moreover have**  $?I \vdash_g ?A \checkmark$  **using** *WT* **by**(*rule WT-gpv-mk-lossless-gpv*)(*simp add: I-def*)  
**ultimately have** *rel-spmf* ( $\lambda x y. fst\ (snd\ x) = fst\ (snd\ y) \wedge (\neg\ fst\ (snd\ y) \longrightarrow (fst\ x \longleftrightarrow fst\ y))$ )  
(*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**

$X\text{-bad}=\lambda \text{ - . True and } ?\text{bad1.0}=\text{fst and } ?\text{bad2.0}=\text{fst and } \mathcal{I} = ?\mathcal{I},$   
**THEN** *rel-spmf-mono*])  
 (auto simp add: lazy-channel-insec'-def intro: WT-calleeI)  
**also let**  $?I = (\lambda(-, s1, s2, s3). \text{pred-cstate } (\lambda x. \text{length } x = \eta) s1 \wedge \text{pred-option}$   
 $(\lambda(x, y). \text{length } x = \eta \wedge \text{length } y = \eta) s2 \wedge \text{ran } s3 \subseteq \text{nlists UNIV } \eta)$   
**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  $s, \text{proj1 } (\text{projr } x))$  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: 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-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: op-  
tion.split-asm)  
**subgoal by**(cases (snd  $s, ()$ ) rule: lazy-channel-recv-f.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)  
**done**  
**done**  
**then have** exec-gpv  $?orc\text{-lft } ?A$  (False, Void, None, Map.empty) = exec-gpv  
 $?orc\text{-lft } A$  (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  $s, \text{proj1 } (\text{projr } x))$  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-

$\text{tion.split-asm if-split-asm cstate.split-asm simp add: in-nlists-UNIV vld-def ran-def rnd-def option.pred-set )}$   
**done**  
**subgoal for  $s$**   
**apply**( $\text{clarsimp simp add: } \mathcal{I}\text{-def; intro conjI WT-calleeI; clarsimp simp add: lazy-channel-insec'-def lazy-channel-send'-def lazy-channel-recv'-def}$ )  
**subgoal for - - -  $x$**   
**by**( $\text{cases (snd } s, x) \text{ rule: lazy-channel-insec.cases}$ )  
 $(\text{auto simp add: vld-def in-nlists-UNIV rnd-def ran-def split: option.split-asm})$   
**subgoal by**( $\text{cases (snd } s, ()) \text{ rule: lazy-channel-recv.cases}$ )( $\text{auto } 4 \ 3 \text{ split: option.split-asm if-split-asm cstate.split-asm simp add: in-nlists-UNIV vld-def ran-def rnd-def}$ )  
**done**  
**done**  
**then have**  $\text{exec-gpv ?orc-rgt ?A (False, Void, None, Map.empty) = exec-gpv ?orc-rgt } \mathcal{A} \text{ (False, Void, None, Map.empty)}$   
**using**  $\text{WT by(rule callee-invariant-on.exec-gpv-mk-lossless-gpv) simp}$   
**finally have**  $|\text{Sigma-Algebra.measure (measure-spmf (game } \mathcal{A} \text{ lazy-channel-recv-f'}) \{x. fst } x\} - \text{Sigma-Algebra.measure (measure-spmf (game } \mathcal{A} \text{ lazy-channel-recv'}) \{x. fst } x\}| \leq \text{Sigma-Algebra.measure (measure-spmf (game } \mathcal{A} \text{ lazy-channel-recv-f'}) \{x. snd } x\}$   
**unfolding game-def**  
**by**  $-\text{(rule fundamental-lemma[where ?bad2.0=snd]; auto simp add: spmf-rel-map map-spmf-conv-bind-spmf[symmetric] split-def)}$   
  
**moreover have**  $\text{Sigma-Algebra.measure (measure-spmf (game } \mathcal{A} \text{ lazy-channel-recv-f'}) \{x. fst } x\} = \text{spmf ?L True}$   
**unfolding game-def**  
**by**( $\text{auto simp add: map-spmf-conv-bind-spmf exec-gpv-resource-of-oracle connect-def spmf-conv-measure-spmf}$ )  
 $(\text{auto simp add: rel-pred-def intro!: rel-spmf-bind-reflI measure-spmf-parametric[where } A=\lambda l r. fst l \longleftrightarrow r, \text{ THEN rel-funD, THEN rel-funD]})$   
  
**moreover have**  $\text{spmf ?R True = Sigma-Algebra.measure (measure-spmf (game } \mathcal{A} \text{ lazy-channel-recv'}) \{x. fst } x\}$   
**unfolding game-def**  
**by**( $\text{auto simp add: map-spmf-conv-bind-spmf exec-gpv-resource-of-oracle connect-def spmf-conv-measure-spmf}$ )  
 $(\text{auto simp add: rel-pred-def intro!: rel-spmf-bind-reflI measure-spmf-parametric[where } A=\lambda l r. l \longleftrightarrow fst r, \text{ THEN rel-funD, THEN rel-funD]})$   
  
**ultimately show ?thesis by simp**  
**qed**  
  
**have**  $\text{fact3: spmf (connect } \mathcal{A} \text{ (RES (lazy-channel-insec' } \oplus_{\mathcal{O}} \text{ lazy-channel-send' } \oplus_{\mathcal{O}} \text{ lazy-channel-recv'}) (False, Void, None, Map.empty))) True}$

= *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* (=) ( $\lambda x y. y = \text{snd } x$ ))  
 (*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* (=) ( $\lambda x y. y = \text{snd } x$ ))  
 (*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* (=) ( $\lambda x y. y = \text{snd } x$ ))  
 (*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-refl split:option.split cstate.split*)

**have**[*simp*]: *rel-spmf* (*rel-prod* (=) ( $\lambda x y. y = \text{snd } x$ ))  
 ((*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 (=) ( $\lambda x$   
*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 (=) ( $\lambda x y. y = \text{snd } x$ )]*); *auto*)+

**qed**

**have**[*simp*]: *rel-spmf* ( $\lambda x y. \text{fst } x = \text{fst } y$ )  
 (*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 (=) ( $\lambda x y. y = \text{snd } x$ )]*)

(*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 A lazy-channel-recv-f'))
  {x. snd x} ≤ q / real (2 ^ η)
  proof -
    let ?orc-sum = lazy-channel-insec' ⊕O lazy-channel-send' ⊕O lazy-channel-recv-f'

    have Sigma-Algebra.measure (measure-spmf (game A lazy-channel-recv-f')) {x.
  snd x}
      = spmf (map-spmf (fst ∘ snd) (exec-gpv ?orc-sum A (False, Void, None,
  Map.empty))) True
    unfolding game-def
    by (simp add: split-def map-spmf-conv-bind-spmf[symmetric] measure-map-spmf)
      (simp add: spmf-conv-measure-spmf measure-map-spmf vimage-def)
    also have ... ≤ (1 / real (card (nlists (UNIV :: bool set) η))) * real q
    proof -
      have *: spmf (map-spmf (fst ∘ snd) (?orc-sum (False, ams, es, as) query))
  True
        ≤ 1 / real (card (nlists (UNIV :: bool set) η)) for ams es as
  query
    proof (cases query)
      case (Inl adv-query)
        then show ?thesis
          by (cases ((ams, es, as), adv-query) rule: lazy-channel-insec.cases)
            (auto simp add: lazy-channel-insec'-def rnd-def map-spmf-conv-bind-spmf
  bind-spmf-const split: option.split)
        next
          case (Inr query')
            note Snd-Rcv = this
            then show ?thesis
              proof (cases query')
                case (Inr rcv-query)
                  with Snd-Rcv show ?thesis
                    by (cases ams, auto simp add: lazy-channel-recv-f'-def map-spmf-conv-bind-spmf
  split-def split:option.split)
                  (auto simp add: spmf-of-set map-spmf-conv-bind-spmf[symmetric] rnd-def
  spmf-map vimage-def spmf-conv-measure-spmf[symmetric] split: split-indicator)
                qed (cases ams, simp-all add: lazy-channel-send'-def)
              qed

    show ?thesis by (rule oi-True.interaction-bounded-by-exec-gpv-bad[where
  bad=fst]) (auto simp add: * assms)

```

qed

also have ... = 1 / real (2 ^ η) \* real q by (simp add: card-nlists)  
finally show ?thesis by simp  
qed

have ?LHS ≤ Sigma-Algebra.measure (measure-spmf (game 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 × bool list) option + unit) × unit × bool list  
cstate) spmf ⇒  
(bool list cstate × (bool list × bool list) option × (bool list ⇒ bool list option))  
spmf ⇒ 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 ↦ a]))  
**if** length m = id' η  
| S' (return-spmf (Inr (), (), Collect m))  
  (return-spmf (Collect m, None, [m ↦ 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 ↦ 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' ↦ 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' η
| 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 ↦ a]))
if length m = id' η and length m' = id' η and length a' = id' η and m ≠ m'
| S' (return-spmf (Inr (), (), Fail))
  (return-spmf (Collect m', Some (a', m'), [m ↦ a]))
if length m = id' η and length m' = id' η and length a' = id' η and a ≠ a'
| S' (return-spmf (Inl None, (), Collect m'))
  (return-spmf (Collect m', Some (a', m'), [m' ↦ a']))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inr (), (), Collect m'))
  (return-spmf (Collect m', Some (a', m'), [m' ↦ a']))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inr (), (), Void))
  (map-spmf (λa'. (Fail, None, [m' ↦ a'])) (spmf-of-set (nlists UNIV η)))
if length m' = id' η
| S' (return-spmf (Inr (), (), Fail))
  (map-spmf (λa'. (Fail, None, [m' ↦ a'])) (spmf-of-set (nlists UNIV η)))
if length m' = id' η
| S' (return-spmf (Inr (), (), Store m))
  (map-spmf (λa'. (Fail, None, [m' ↦ a'])) (spmf-of-set (nlists UNIV η)))
if length m = id' η and length m' = id' η
| S' (return-spmf (Inr (), (), Fail))
  (map-spmf (λa'. (Fail, None, [m ↦ a, m' ↦ a'])) (spmf-of-set (nlists UNIV
η)))
if length m = id' η and length m' = id' η and m ≠ m'
| S' (return-spmf (Inl (Some (a', m')), (), Fail))
  (return-spmf (Fail, None, [m' ↦ a']))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inl None, (), Fail))
  (return-spmf (Fail, None, [m' ↦ a']))
if length m' = id' η and length a' = id' η

```

**private lemma** *trace-eq-sim*:

```

shows (valid-insecQ <+> nlists UNIV (id' η) <+> UNIV) ⊢R
  RES (callee-auth-channel sim ⊕O ††channel.send-oracle ⊕O ††channel.recv-oracle)
(Inl None, (), Void)
≈
  RES (lazy-channel-insec ⊕O lazy-channel-send ⊕O lazy-channel-recv-f) (Void,
None, Map.empty)
(is ?A ⊢R RES (?L1 ⊕O ?L2 ⊕O ?L3) ?SL ≈ RES (?R1 ⊕O ?R2 ⊕O ?R3)
?SR)

```

```

proof –
  note [simp] =
    spmf.map-comp o-def map-bind-spmf bind-map-spmf bind-spmf-const exec-gpv-bind
    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 ⊕O rorc-channel-send ⊕O rorc-channel-recv)
  ((False, ()), Map.empty, Void)
  (is ?L = ?R) including lifting-syntax
proof -
  have *: (λx y. map-spmf (map-prod id lprodr) ((A ⊕O B) (rprodl x) y))
    = (λx yl. map-spmf (λp. ((), lprodr (snd p))) (A (rprodl x) yl)) ⊕O
      (λx yr. map-spmf (λp. (fst p, lprodr (snd p))) (B (rprodl x) yr)) for A
  B C

```

```

proof –
  have aux: map-prod id g  $\circ$  apfst h = apfst h  $\circ$  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]
      spmfm.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 ( $\dagger\dagger$ insec-channel.insec-oracle  $\oplus_O$  (rprodl  $\dashrightarrow$  id  $\dashrightarrow$ 
map-spmf (map-prod id lprodr))
    (lift-state-oracle extend-state-oracle (attach-callee
      ( $\lambda s m.$  if s
        then Done ((), True)
        else do {
          r  $\leftarrow$  Pause (Inl m) Done;
          a  $\leftarrow$  lift-spmf (mac  $\eta$  (projl r) m);
          -  $\leftarrow$  Pause (Inr (a, m)) Done;
          (Done ((), True))) (rorc.rnd-oracle (rnd  $\eta$ ) $\dagger$   $\oplus_O$   $\dagger$ channel.send-oracle))
     $\oplus_O$ 
       $\dagger\dagger$ ( $\lambda s q.$  do {
        (amo, s')  $\leftarrow$   $\dagger$ channel.recv-oracle s ();
        case amo of
          None  $\Rightarrow$  return-spmf (None, s')
          | Some (a, m)  $\Rightarrow$  do {
            (r, s'')  $\leftarrow$  (rorc.rnd-oracle (rnd  $\eta$ ) $\dagger$  s' m);
            a'  $\leftarrow$  mac  $\eta$  r m;
            return-spmf (if a' = a then Some m else None, s'')
          })
      )) (False, (()),
Map.empty, Void)
  proof –
  note[simp] = attach-CNV-RES attach-callee-parallel-intercept attach-stateless-callee
resource-of-oracle-rprodl Rel-def prod.rel-eq[symmetric] extend-state-oracle-plus-oracle[symmetric]
conv-callee-parallel[symmetric] conv-callee-parallel-id-left[where s=(), sym-
metric]
o-def bind-map-spmf exec-gpv-bind split-def option.case-distrib[where h= $\lambda$ gpv.
exec-gpv - gpv -]

  show ?thesis
  unfolding channel.res-def rorc.res-def macode.res-def macode.routing-def
macode. $\pi$ E-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) $\dagger$ 
  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**  $f = \lambda gpv. \text{exec-gpv} - gpv - ]$  *cong: if-cong*)  
*(simp add: split-def o-def rprodl-def Pair-fst-Unity bind-map-spmf map-spmf-bind-spmf*

*if-distrib*[**where**  $f = \text{map-spmf} - ]$  *option.case-distrib*[**where**  $h = \text{map-spmf} - ]$   
*cong: if-cong option.case-cong*)

**finally show** *?thesis* .  
**qed**

**private lemma** *ideal-resource-wiring*:  $(CNV \text{ callee } s) \models 1_C \triangleright \text{channel.res auth-channel.auth-oracle}$   
 $=$   
 $RES (\text{callee-auth-channel callee} \oplus_O \dagger \dagger \text{channel.send-oracle} \oplus_O \dagger \dagger \text{channel.recv-oracle}$   
 $) (s, (), \text{Void}) \quad (\text{is } ?L1 \triangleright - = ?R)$

**proof** –

**have**[*simp*]:  $\mathcal{I}\text{-full}, \mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \vdash_C ?L1 \sim ?L1$  (**is**  $-$ ,  $?I \vdash_C - \sim$   
 $-$ )

**by**(*rule eq-I-converter-mono*)

(*rule parallel-converter2-eq-I-cong eq-I-converter-refl WT-converter-I-full*  
 $\mathcal{I}\text{-full-le-plus-I order-refl plus-I-mono}$ ) $+$

**have**[*simp*]:  $?I \vdash_c (\text{auth-channel.auth-oracle} \oplus_O \text{channel.send-oracle} \oplus_O \text{chan-}$   
 $\text{nel.recv-oracle}) s \checkmark$  **for**  $s$

**by**(*rule WT-plus-oracleI WT-parallel-oracle WT-callee-full; (unfold split-paired-all)?*) $+$

**have**[*simp*]:  $?L1 \triangleright RES (\text{auth-channel.auth-oracle} \oplus_O \text{channel.send-oracle} \oplus_O$   
 $\text{channel.recv-oracle}) \text{Void} = ?R$

**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**  $\mathcal{I}' = ?I$  **and**  $\text{conv} = ?L1$   
**and**  $\text{conv}' = ?L1$ ]) *simp-all*

**qed**

**lemma** *all-together*:

**defines**  $\mathcal{I} \equiv \mathcal{I}\text{-uniform} (\text{Set.Collect} (\text{valid-mac-query } \eta)) (\text{insert None (Some '}$   
 $(\text{nlists UNIV } \eta \times \text{nlists UNIV } \eta))) \oplus_{\mathcal{I}}$

$(\mathcal{I}\text{-uniform} (\text{vld } \eta) \text{UNIV} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform UNIV} (\text{insert None (Some 'vld } \eta)))$

**assumes**  $\eta > 0$

**and** *interaction-bounded-by'*  $(\lambda -. \text{True}) (\mathcal{A} \eta) q$

**and** *lossless*: *plossless-gpv*  $\mathcal{I} (\mathcal{A} \eta)$

**and**  $WT: \mathcal{I} \vdash_g \mathcal{A} \eta \checkmark$

**shows**

$| \text{spm}f (\text{connect } (\mathcal{A} \eta) (CNV \text{ sim} (\text{Inl None}) \models 1_C \triangleright \text{channel.res auth-channel.auth-oracle}))$

*True* –

$\text{spm}f (\text{connect } (\mathcal{A} \eta) (\text{macode.res} (\text{rnd } \eta) (\text{mac } \eta))) \text{True} \leq q / \text{real} (2 \wedge$

$\eta)$

**proof** –

```

have *:  $\forall a b. ma = Edit (a, b) \longrightarrow length\ a = \eta \wedge length\ b = \eta \implies valid\text{-}mac\text{-}query$ 
 $\eta\ ma$  for  $ma\ a\ b$ 
  by(cases ( $\eta, ma$ ) rule: valid-mac-query.cases)(auto simp add: vld-def in-nlists-UNIV)

have assm4-alt:  $outs\text{-}gpv\ \mathcal{I}\ (\mathcal{A}\ \eta) \subseteq valid\text{-}insecQ\ \langle + \rangle\ nlists\ UNIV\ (id'\ \eta)\ \langle + \rangle$ 
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  $\oplus_O$   $\dagger\dagger$ channel.send-oracle  $\oplus_O$ 
 $\dagger\dagger$ channel.recv-oracle)
  ( $\lambda(s1, -, s3). (\forall x\ y. s1 = Inl\ (Some\ (x, y)) \longrightarrow length\ x = \eta \wedge length\ y = \eta)$ 
 $\wedge\ pred\text{-}cstate\ (\lambda x. length\ x = \eta)\ s3$ ) I
  apply unfold-locales
  subgoal for  $s\ x\ y\ s'$ 
    apply(cases (fst  $s$ , proj1  $x$ ) 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  $s$ , proj1  $x$ ) 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
                      done
                    done
                  done
                done
              done
            done
          done
        done
      done
    done
  then have WT1:  $\mathcal{I} \vdash res\ RES$  (callee-auth-channel sim  $\oplus_O$   $\dagger\dagger$ channel.send-oracle

```

$\oplus_O \dagger \dagger \text{channel.recv-oracle}$  (*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*)  
 $(\lambda(s1, s2, s3). \text{pred-cstate } (\lambda x. \text{length } x = \eta) s1 \wedge \text{pred-option } (\lambda(x, y). \text{length } x = \eta \wedge \text{length } y = \eta) s2 \wedge \text{ran } s3 \subseteq \text{nlists UNIV } \eta)$   
 $\mathcal{I}$   
**apply** *unfold-locales*  
**subgoal for**  $s \ x \ y \ s'$   
**using**  $[[\text{simproc del: defined-all}]]$  **apply**(*clarsimp simp add:  $\mathcal{I}$ -def; elim PlusE; clarsimp*)  
**subgoal for**  $--- \ x'$   
**by**(*cases* ( $s, x'$ ) *rule: lazy-channel-insec.cases*)  
 $(\text{auto simp add: vld-def in-nlists-UNIV rnd-def split: option.split-asm})$   
**subgoal by**(*cases* ( $s, \text{projl } (\text{projr } x)$ ) *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*)  
 $(\text{auto simp add: vld-def in-nlists-UNIV rnd-def ran-def split: option.split-asm})$   
**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**  $WT2: \mathcal{I} \vdash \text{res RES } (\text{lazy-channel-insec } \oplus_O \text{ lazy-channel-send } \oplus_O \text{ lazy-channel-recv-f})$  (*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*)  
 $(\lambda(s1, s2, s3). \text{pred-cstate } (\lambda x. \text{length } x = \eta) s1 \wedge \text{pred-option } (\lambda(x, y). \text{length } x = \eta \wedge \text{length } y = \eta) s2 \wedge \text{ran } s3 \subseteq \text{nlists UNIV } \eta)$   
 $\mathcal{I}$   
**apply** *unfold-locales*  
**subgoal for**  $s \ x \ y \ s'$   
**using**  $[[\text{simproc del: defined-all}]]$  **apply**(*clarsimp simp add:  $\mathcal{I}$ -def; elim PlusE; clarsimp*)  
**subgoal for**  $--- \ x'$   
**by**(*cases* ( $s, x'$ ) *rule: lazy-channel-insec.cases*)  
 $(\text{auto simp add: vld-def in-nlists-UNIV rnd-def split: option.split-asm})$   
**subgoal by**(*cases* ( $s, \text{projl } (\text{projr } x)$ ) *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: I-def; intro conjI WT-calleeI; clarsimp)
  subgoal for - - -  $x$ 
    by(cases (s, x) rule: lazy-channel-insec.cases)
    (auto simp add: vld-def in-nlists-UNIV rnd-def ran-def split: option.split-asm)
    subgoal by(cases (s, ()) rule: lazy-channel-recv-f.cases)(auto 4 3 split: option.split-asm if-split-asm simp add: in-nlists-UNIV vld-def ran-def rnd-def)
  done
done
then have  $WT3: \mathcal{I} \vdash_{res} RES$  (lazy-channel-insec  $\oplus_O$  lazy-channel-send  $\oplus_O$  lazy-channel-recv) (Void, None, Map.empty)  $\checkmark$ 
  by(rule callee-invariant-on. WT-resource-of-oracle simp)

have callee-invariant-on ( $\dagger\dagger$ insec-channel.insec-oracle  $\oplus_O$  rorc-channel-send  $\oplus_O$  rorc-channel-recv)
  ( $\lambda(-, m, s). ran\ m \subseteq nlists\ UNIV\ \eta \wedge pred\ cstate\ (\lambda(x, y). length\ x = \eta \wedge length\ y = \eta)\ s$ )  $\mathcal{I}$ 
  apply(unfold-locales)
  subgoal for  $s\ x\ y\ s'$ 
    using [simp proc del: defined-all] apply(clarsimp simp add: 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: I-def; intro conjI WT-calleeI; clarsimp)
    subgoal for - - -  $x'$ 
      by(cases (snd (snd s), x') rule: insec-channel.insec-oracle.cases)
      (auto simp add: vld-def in-nlists-UNIV rnd-def insec-channel.insec-oracle.simps split: option.split-asm)
    subgoal
      by(cases snd (snd s))
      (auto simp add: rorc-channel-recv-def channel.recv-oracle.simps rorc.rnd-oracle.simps rnd-def mac-def vld-def ran-def iff: in-nlists-UNIV split: option.split-asm if-split-asm)
    done
  done
then have  $WT4: \mathcal{I} \vdash_{res} RES$  ( $\dagger\dagger$ insec-channel.insec-oracle  $\oplus_O$  rorc-channel-send)

```

$\oplus_{\mathcal{O}}$  *rorc-channel-recv* ((*False*, ()), *Map.empty*, *Void*)  $\checkmark$   
**by**(*rule callee-invariant-on.WT-resource-of-oracle simp*)

**note** *game-difference*[*OF assms(3)*, *folded I-def*, *OF assms(4,5)*]  
**also note** *connect-cong-trace*[*OF trace-eq-sim WT assm4-alt WT1 WT2*, *sym-metric*]  
**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  $\eta$  mac  $\eta$*  **for**  $\eta$  .

**interpretation** *A-CHAN*: *auth-channel* .

**lemma** *WT-enm*:

$X \neq \{\}$   $\implies$  *I-uniform* (*vld  $\eta$* ) *UNIV*, *I-uniform* (*vld  $\eta$* )  $X \oplus_{\mathcal{I}}$  *I-uniform* ( $X \times$   
*vld  $\eta$* ) *UNIV*  $\vdash_C$  *MAC.enm  $\eta$*   $\checkmark$   
**unfolding** *MAC.enm-def*  
**by**(*rule WT-converter-of-callee*) (*auto simp add: mac-def*)

**lemma** *WT-dem*: *I-uniform* *UNIV* (*insert None (Some 'vld  $\eta$ )*), *I-full*  $\oplus_{\mathcal{I}}$  *I-uniform*  
*UNIV* (*insert None (Some ' (nlists UNIV  $\eta \times$  nlists UNIV  $\eta$ ))*)  $\vdash_C$  *MAC.dem  $\eta$*   
 $\checkmark$   
**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  $\eta$*  (*insec-query-of x*)  
**by**(*cases x*) *simp-all*

**lemma** *secure-mac*:

**defines** *I-real*  $\equiv \lambda \eta. \mathcal{I}\text{-uniform } \{x. \text{valid-mac-query } \eta \ x\}$  (*insert None (Some ' (nlists UNIV  $\eta \times$  nlists UNIV  $\eta$ ))*)  
**and** *I-ideal*  $\equiv \lambda \eta. \mathcal{I}\text{-uniform UNIV}$  (*insert None (Some ' nlists UNIV  $\eta$ )*)  
**and** *I-common*  $\equiv \lambda \eta. \mathcal{I}\text{-uniform (vld } \eta)$  *UNIV*  $\oplus_{\mathcal{I}}$  *I-uniform UNIV* (*insert None (Some ' vld  $\eta$ )*)  
**shows**  
*constructive-security MAC.res* ( $\lambda \cdot. \text{A-CHAN.res}$ ) ( $\lambda \cdot. \text{CNV sim (Inl None)}$ )  
*I-real I-ideal I-common* ( $\lambda \cdot. \text{enat } q$ ) *True* ( $\lambda \cdot. \text{insec-auth-wiring}$ )

**proof**

**show** *WT-res [WT-intro]*: *I-real  $\eta \oplus_{\mathcal{I}}$  I-common  $\eta \vdash_{\text{res}} \text{MAC.res } \eta$*   $\checkmark$  **for**  $\eta$

**proof** –

**let** *?I* = *pred-cstate* ( $\lambda(x, y). \text{length } x = \eta \wedge \text{length } y = \eta$ )

```

have *: callee-invariant-on (MAC.RO.rnd-oracle  $\eta \oplus_O$  MAC.RO.rnd-oracle  $\eta$ )
( $\lambda m. \text{ran } m \subseteq \text{vld } \eta$ ) ( $\mathcal{I}$ -uniform (vld  $\eta$ ) (vld  $\eta$ )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full) for  $\eta$ 
  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  $\eta$ ) (vld  $\eta$ )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full  $\vdash_{\text{res}}$  MAC.RO.res  $\eta \ \checkmark$ 
for  $\eta$ 
  unfolding MAC.RO.res-def by(rule callee-invariant-on.WT-resource-of-oracle[OF
*]) simp
  have [simp]:  $\mathcal{I}$ -real  $\eta \vdash_c$  MAC.INSEC.insec-oracle  $s \ \checkmark$  if ?I  $s$  for  $s$ 
  apply(rule WT-calleeI)
  subgoal for  $x$  using that by(cases ( $s, x$ ) rule: MAC.INSEC.insec-oracle.cases)(auto
simp add:  $\mathcal{I}$ -real-def in-nlists-UNIV)
  done
  have [simp]:  $\mathcal{I}$ -uniform UNIV (insert None (Some ‘ (nlists UNIV  $\eta \times$  nlists
UNIV  $\eta$ )))  $\vdash_c$  A-CHAN.recv-oracle  $s \ \checkmark$ 
  if ?I  $s$  for  $s ::$  (bool list  $\times$  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  $\eta \oplus_{\mathcal{I}}$  ( $\mathcal{I}$ -uniform (vld  $\eta \times$  vld  $\eta$ ) UNIV  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform UNIV (insert
None (Some ‘ (nlists UNIV  $\eta \times$  nlists UNIV  $\eta$ ))))))
  apply unfold-locales
  subgoal for  $s \ x \ y \ s'$ 
  by(cases  $s$ ; cases ( $s, \text{proj1 } x$ ) 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  $\eta \oplus_{\mathcal{I}}$  ( $\mathcal{I}$ -uniform (vld  $\eta \times$  vld  $\eta$ ) UNIV  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform
UNIV (insert None (Some ‘ (nlists UNIV  $\eta \times$  nlists UNIV  $\eta$ ))))  $\vdash_{\text{res}}$  MAC.INSEC.res
 $\checkmark$ 
  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  $X = \text{vld } \eta$ ] WT-dem | (simp add: vld-def;
fail))+
  qed
  let ?I =  $\lambda \eta. \text{pred-cstate } (\lambda x. \text{length } x = \eta)$ 
  have WT-auth:  $\mathcal{I}$ -uniform UNIV (insert None (Some ‘ nlists UNIV  $\eta$ ))  $\vdash_c$ 
A-CHAN.auth-oracle  $s \ \checkmark$ 
  if ?I  $\eta \ s$  for  $\eta \ s$ 
  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\text{-recv}: \mathcal{I}\text{-uniform UNIV (insert None (Some `vld`  $\eta$ ))} \vdash_c A\text{-CHAN.recv-oracle}$   
 $s \sqrt$   
**if**  $?I \eta s$  **for**  $\eta s$  **using** *that*  
**by**(cases  $s$ )(auto intro!:  $WT\text{-calleeI simp add: vld-def in-nlists-UNIV}$ )  
**have**  $*$ : *callee-invariant-on* ( $A\text{-CHAN.auth-oracle} \oplus_O A\text{-CHAN.send-oracle} \oplus_O$   
 $A\text{-CHAN.recv-oracle}$ )  
 $(?I \eta)$  ( $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta$ ) **for**  $\eta$   
**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**  $WT\text{-auth WT-recv}$  **by**(auto *simp add: I-common-def*  
 $\mathcal{I}\text{-ideal-def intro: WT-calleeI}$ )  
**done**  
**show**  $WT\text{-auth}: \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{res} A\text{-CHAN.res} \sqrt$  **for**  $\eta$   
**unfolding**  $A\text{-CHAN.res-def}$  **by**(*rule callee-invariant-on.WT-resource-of-oracle[OF*  
 $*$ ]) *simp*

**let**  $?I\text{-sim} = \lambda \eta (s :: (\text{bool list} \times \text{bool list}) \text{ option} + \text{unit}). \forall x y. s = \text{Inl (Some}$   
 $(x, y)) \longrightarrow \text{length } x = \eta \wedge \text{length } y = \eta$

**have**  $\mathcal{I}\text{-real } \eta, \mathcal{I}\text{-ideal } \eta \vdash_C \text{CNV sim } s \sqrt$  **if**  $?I\text{-sim } \eta s$  **for**  $\eta s$  **using** *that*  
**apply**(*coinduction arbitrary: s*)  
**subgoal** **for**  $q r \text{ conv}' s$  **by**(cases ( $s$ ,  $q$ ) *rule: sim.cases*)(auto *simp add: I-real-def*  
 $\mathcal{I}\text{-ideal-def vld-def in-nlists-UNIV}$ )  
**done**  
**then show** [ $WT\text{-intro}$ ]:  $\mathcal{I}\text{-real } \eta, \mathcal{I}\text{-ideal } \eta \vdash_C \text{CNV sim (Inl None)} \sqrt$  **for**  $\eta$  **by**  
*simp*

**{ fix**  $\mathcal{A} :: \text{security} \Rightarrow ((\text{bool list} \times \text{bool list}) \text{ insec-query} + \text{bool list} + \text{unit}, (\text{bool}$   
 $\text{list} \times \text{bool list}) \text{ option} + \text{unit} + \text{bool list option}) \text{ distinguisher}$   
**assume**  $WT: \mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{A} \eta \sqrt$  **for**  $\eta$   
**assume** *bounded[simplified]: interaction-bounded-by'* ( $\lambda \cdot \text{True}$ ) ( $\mathcal{A} \eta$ )  $q$  **for**  $\eta$   
**assume** *lossless[simplified]: True  $\implies$  plossless-gpv* ( $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta$ )  
 $(\mathcal{A} \eta)$  **for**  $\eta$   
**show** *negligible* ( $\lambda \eta. \text{advantage } (\mathcal{A} \eta) (\text{CNV sim (Inl None)} \mid= 1_C \triangleright A\text{-CHAN.res})$   
 $(\text{MAC.res } \eta)$ )  
**proof** –  
**have**  $f1: \text{negligible } (\lambda \eta. q * (1 / \text{real } (2 \wedge \eta)))$  (**is negligible**  $?ov$ )  
**by**(*rule negligible-poly-times[where n=0]*) (*simp add: negligible-inverse-powerI*) +  
  
**have**  $f2: (\lambda \eta. \text{spmf (connect } (\mathcal{A} \eta) (\text{CNV sim (Inl None)} \mid= 1_C \triangleright A\text{-CHAN.res}))$   
 $\text{True} -$   
 $\text{spmf (connect } (\mathcal{A} \eta) (\text{MAC.res } \eta)) \text{ True}) \in O(?ov)$  (**is**  $?L \in -$ )  
**by** (auto *simp add: bigo-def intro!: all-together[simplified] bounded eventu-*  
*ally-at-top-linorderI[where c=1]*)  
 $\text{exI[where } x=1]$  *lossless[unfolded I-real-def I-common-def] WT[unfolded*  
 $\mathcal{I}\text{-real-def I-common-def}$ ])  
**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 1C
show ∃ cnv. ∀  $\mathcal{D}$ . (∀  $\eta$ .  $\mathcal{I}$ -ideal  $\eta \oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\eta \vdash_g \mathcal{D} \eta \checkmark$ )  $\longrightarrow$ 
  (∀  $\eta$ . interaction-bounded-by' ( $\lambda$ -. True) ( $\mathcal{D} \eta$ )  $q$ )  $\longrightarrow$ 
  (∀  $\eta$ . True  $\longrightarrow$  plossless-gpv ( $\mathcal{I}$ -ideal  $\eta \oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\eta$ ) ( $\mathcal{D} \eta$ ))  $\longrightarrow$ 
  (∀  $\eta$ . wiring ( $\mathcal{I}$ -ideal  $\eta$ ) ( $\mathcal{I}$ -real  $\eta$ ) (cnv  $\eta$ ) (insec-query-of, map-option
snd))  $\wedge$ 
  negligible ( $\lambda$  $\eta$ . advantage ( $\mathcal{D} \eta$ ) A-CHAN.res (cnv  $\eta \mid= 1_C \triangleright$  MAC.res
 $\eta$ ))
proof(intro exI[where  $x=\lambda$ -. ?cnv] strip conjI)
  fix  $\mathcal{D} ::$  security  $\Rightarrow$  (auth-query + bool list + unit, bool list option + unit +
bool list option) distinguisher
  assume WT-D [rule-format, WT-intro]: ∀  $\eta$ .  $\mathcal{I}$ -ideal  $\eta \oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\eta \vdash_g \mathcal{D}
\eta \checkmark$ 

  let ?A =  $\lambda$  $\eta$ . UNIV  $\langle + \rangle$  nlists UNIV  $\eta \langle + \rangle$  UNIV
  have WT1:  $\mathcal{I}$ -ideal  $\eta$ , map- $\mathcal{I}$  insec-query-of (map-option snd) ( $\mathcal{I}$ -real  $\eta$ )  $\vdash_C
1_C \checkmark$  for  $\eta$ 
  using WT-converter-id order-refl by(rule WT-converter-mono)(auto simp
add: le- $\mathcal{I}$ -def  $\mathcal{I}$ -ideal-def  $\mathcal{I}$ -real-def)
  have WT0:  $\mathcal{I}$ -ideal  $\eta \oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\eta \vdash_{res}$  map-converter id id insec-query-of
(map-option snd)  $1_C \mid= 1_C \triangleright$  MAC.res  $\eta \checkmark$  for  $\eta$ 
  by(rule WT1 WT-intro | simp)+

  have WT2:  $\mathcal{I}$ -ideal  $\eta$ , map- $\mathcal{I}$  insec-query-of (map-option snd) ( $\mathcal{I}$ -real  $\eta$ )  $\vdash_C
1_C \checkmark$  for  $\eta$ 
  using WT-converter-id order-refl
  by(rule WT-converter-mono)(auto simp add: le- $\mathcal{I}$ -def  $\mathcal{I}$ -ideal-def  $\mathcal{I}$ -real-def)
  have WT-cnv:  $\mathcal{I}$ -ideal  $\eta$ ,  $\mathcal{I}$ -real  $\eta \vdash_C$  ?cnv  $\checkmark$  for  $\eta$ 
  by(rule WT-converter-map-converter)(simp-all add: WT2)

from eq- $\mathcal{I}$ -converter-refl[OF this] this
show wiring ( $\mathcal{I}$ -ideal  $\eta$ ) ( $\mathcal{I}$ -real  $\eta$ ) ?cnv insec-auth-wiring for  $\eta$  ..

define res' :: security  $\Rightarrow$  -  $\Rightarrow$  auth-query + (bool list + bool list  $\times$  bool list)
+ bool list + unit  $\Rightarrow$  -
  where res'  $\eta =$ 
    map-fun id (map-fun insec-query-of (map-spmf (map-prod (map-option
snd) id)))  $\dagger$ MAC.INSEC.insec-oracle  $\oplus_O$ 
    ((MAC.RO.rnd-oracle  $\eta$ ) $\dagger \oplus_O \dagger$ A-CHAN.send-oracle)  $\oplus_O$  (MAC.RO.rnd-oracle
 $\eta$ ) $\dagger \oplus_O \dagger$ A-CHAN.recv-oracle
  for  $\eta$ 

  have push: map-resource (map-sum f id) (map-sum g id) (( $1_C \mid=$  conv)  $\triangleright$ 
res) =
    ( $1_C \mid=$  conv)  $\triangleright$  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<sub>C</sub>*  $\models$  *conv*)  $\triangleright$  *res*) =  
*map-converter* *f* *g* *id* *id* *1<sub>C</sub>*  $\models$  *conv*  $\triangleright$  *res*  
**by** (*simp* *add*: *attach-map-converter* *parallel-converter2-map1-out* *sum.map-id0*)  
**also have** ... = (*1<sub>C</sub>*  $\models$  *conv*)  $\triangleright$  *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*  $\eta$ ) =  
*1<sub>C</sub>*  $\models$  *MAC.enm*  $\eta$   $\models$  *MAC.dem*  $\eta$   $\triangleright$  *RES* (*res'*  $\eta$ ) (*Map.empty*, *Void*) **for**  
 $\eta$   
**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*  $\Rightarrow$  (*unit*  $\times$  *bool*  $\times$  *unit*)  $\times$  (*bool* *list*  $\Rightarrow$  *bool* *list* *option*)  
 $\times$  - *cstate*  $\Rightarrow$  *auth-query* + *bool* *list* + *unit*  $\Rightarrow$  -  
**where** *res''*  $\eta$  = *map-fun* *rprodl* (*map-fun* *id* (*map-spmf* (*map-prod* *id*  
*lprodr*)))  
(*lift-state-oracle* *extend-state-oracle*  $\dagger$  (*map-fun* *id* (*map-fun* *insec-query-of*  
(*map-spmf* (*map-prod* (*map-option* *snd*) *id*)))  $\dagger$  *MAC.INSEC.insec-oracle*)  $\oplus_O$   
 $\dagger$  (*map-fun* *rprodl* (*map-fun* *id* (*map-spmf* (*map-prod* *id* *lprodr*)))  
(*lift-state-oracle* *extend-state-oracle*  
(*attach-callee*  
( $\lambda$  *bs* *m*. *if* *bs* *then* *Done* ( $\lambda$  (), *True*) *else* *Pause* (*Inl* *m*) *Done*  $\ggg$  ( $\lambda$  *r*.  
*lift-spmf* (*mac*  $\eta$  (*projl* *r*) *m*)  $\ggg$  ( $\lambda$  *a*. *Pause* (*Inr* (*a*, *m*)) *Done*  $\ggg$  ( $\lambda$  -. *Done* ( $\lambda$  (),  
*True*))))))  
((*MAC.RO.rnd-oracle*  $\eta$ )  $\dagger$   $\oplus_O$   $\dagger$  *A-CHAN.send-oracle*))  $\oplus_O$   
 $\dagger$  ( $\lambda$  *s* *q*.  $\dagger$  *A-CHAN.recv-oracle* *s* ())  $\ggg$   
( $\lambda$  *x*. *case* *x* *of* (*None*, *s'*)  $\Rightarrow$  *return-spmf* (*None*, *s'*)  
| (*Some* (*x1*, *x2a*), *s'*)  $\Rightarrow$  (*MAC.RO.rnd-oracle*  $\eta$ )  $\dagger$  *s'*  
 $\ggg$  ( $\lambda$  (*x*, *s'*). *mac*  $\eta$  *x* *x2a*  $\ggg$  ( $\lambda$  *y*. *return-spmf* (*if* *y* = *x1* *then* *Some* *x2a* *else*  
*None*, *s'*))))))  
**for**  $\eta$   
**have** ?*cnv*  $\models$  *1<sub>C</sub>*  $\triangleright$  *MAC.res*  $\eta$  = *1<sub>C</sub>*  $\models$  *MAC.enm*  $\eta$   $\models$  *MAC.dem*  $\eta$   $\triangleright$  *RES*  
(*res'*  $\eta$ ) (*Map.empty*, *Void*) **for**  $\eta$   
**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** ...  $\eta$  = *RES* (*res''*  $\eta$ ) ((((), *False*, ()), *Map.empty*, *Void*) **for**  $\eta$   
**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  $h=\lambda x. \text{exec-gpv} - x$  -]
  option.case-distrib[where  $h=\lambda x. \text{exec-gpv} - x$  -] prod.case-distrib[where
 $h=\lambda x. \text{exec-gpv} - x$  -] exec-gpv-bind bind-map-spmf o-def
  cong: sum.case-cong option.case-cong)
also
define  $S :: \text{security} \Rightarrow \text{bool list} \text{ cstate} \Rightarrow (\text{unit} \times \text{bool} \times \text{unit}) \times (\text{bool list} \Rightarrow$ 
 $\text{bool list option}) \times (\text{bool list} \times \text{bool list}) \text{ cstate} \Rightarrow \text{bool}$ 
  where  $S \eta l r = (\text{case } (l, r) \text{ of}$ 
     $(\text{Void}, ((-, \text{False}, -), m, \text{Void})) \Rightarrow m = \text{Map.empty}$ 
    |  $(\text{Store } x, ((-, \text{True}, -), m, \text{Store } (y, z))) \Rightarrow \text{length } x = \eta \wedge \text{length } y = \eta \wedge$ 
 $\text{length } z = \eta \wedge m = [z \mapsto y] \wedge x = z$ 
    |  $(\text{Collect } x, ((-, \text{True}, -), m, \text{Collect } (y, z))) \Rightarrow \text{length } x = \eta \wedge \text{length } y =$ 
 $\eta \wedge \text{length } z = \eta \wedge m = [z \mapsto y] \wedge x = z$ 
    |  $(\text{Fail}, ((-, \text{True}, -), m, \text{Fail})) \Rightarrow \text{True}$ 
    |  $- \Rightarrow \text{False}$ ) for  $\eta l r$ 

  note  $[\text{simp}] = \text{spmfm-rel-map bind-map-spmf exec-gpv-bind}$ 
  have  $\text{connect } (\mathcal{D} \eta) (?cnv \models 1_C \triangleright \text{MAC.res } \eta) = \text{connect } (\mathcal{D} \eta) \text{ A-CHAN.res}$ 
for  $\eta$ 
  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  $\eta$  mac  $\eta$  for  $\eta$*  .

**interpretation** *AUTH*: *auth-channel* .

**interpretation** *CIPHER*: *cipher key  $\eta$  enc  $\eta$  dec  $\eta$  for  $\eta$*  .

**interpretation** *SEC*: *sec-channel* .

**lemma** *plossless-enc* [*plossless-intro*]:

*plossless-converter* ( $\mathcal{I}$ -uniform (*nlists UNIV  $\eta$* ) UNIV) ( $\mathcal{I}$ -uniform UNIV (*nlists UNIV  $\eta$* )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform (*nlists UNIV  $\eta$* ) UNIV) (*CIPHER.enc  $\eta$* )

**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  $\eta$ )*)) ( $\mathcal{I}$ -uniform UNIV (*nlists UNIV  $\eta$* )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform UNIV (*insert None (Some ‘ nlists UNIV  $\eta$ )*)) (*CIPHER.dec  $\eta$* )

**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  $\eta$*   $\oplus_{\mathcal{O}}$  *CIPHER.KEY.key-oracle  $\eta$* )

( $\lambda x$ . *case x of None  $\Rightarrow$  True | Some  $x' \Rightarrow$  length  $x' = \eta$* )

( $\mathcal{I}$ -uniform UNIV (*nlists UNIV  $\eta$* )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -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  $\eta$*   $\oplus_{\mathcal{O}}$  *CIPHER.KEY.key-oracle  $\eta$*

$\lambda x$ . *case x of None  $\Rightarrow$  True | Some  $x' \Rightarrow$  length  $x' = \eta$*

$\mathcal{I}$ -uniform UNIV (*nlists UNIV  $\eta$* )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full **for**  $\eta$

**by**(*rule callee-invariant-on-key-oracle*)

**lemma** *WT-enc* [*WT-intro*]:  $\mathcal{I}$ -uniform (*nlists UNIV  $\eta$* ) UNIV,

$\mathcal{I}$ -uniform UNIV (*nlists UNIV  $\eta$* )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform (*vld  $\eta$* ) UNIV  $\vdash_C$  *CIPHER.enc  $\eta$*   $\checkmark$

**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*  $\eta$ )),  
 $\mathcal{I}$ -uniform *UNIV* (nlists *UNIV*  $\eta$ )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform *UNIV* (insert None (Some ‘  
*vld*  $\eta$ ))  $\vdash_C$   
*CIPHER.dec*  $\eta$   $\surd$   
**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*  
 $\eta$ ) (*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*  
 $\eta$ ) (*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*  $\eta$  *x*} *UNIV*  
**and**  $\mathcal{I}$ -ideal  $\equiv \lambda\cdot. \mathcal{I}$ -full  
**and**  $\mathcal{I}$ -common  $\equiv \lambda\eta. \mathcal{I}$ -uniform (*vld*  $\eta$ ) *UNIV*  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -full

**shows**

*constructive-security*

( $\lambda\eta. 1_C \models (CIPHER.enc \eta \models CIPHER.dec \eta) \odot parallel-wiring \triangleright$   
 $parallel-resource1-wiring \triangleright$   
 $CIPHER.KEY.res \eta \parallel$   
 $(1_C \models MAC.enm \eta \models MAC.dem \eta \triangleright$   
 $1_C \models parallel-wiring \triangleright$   
 $parallel-resource1-wiring \triangleright MAC.RO.res \eta \parallel INSEC.res))$

( $\lambda\cdot. SEC.res$ )

( $\lambda\eta. CNV Message-Authentication-Code.sim (Inl None) \odot CNV One-Time-Pad.sim$   
*None*)

( $\lambda\eta. \mathcal{I}$ -uniform (Set.Collect (*valid-mac-query*  $\eta$ )) (insert None (Some ‘ (nlists  
*UNIV*  $\eta \times$  nlists *UNIV*  $\eta$ ))))

( $\lambda\eta. \mathcal{I}$ -uniform *UNIV* {None, Some  $\eta$ })

( $\lambda\eta. \mathcal{I}$ -uniform (nlists *UNIV*  $\eta$ ) *UNIV*  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform *UNIV* (insert None  
(Some ‘ nlists *UNIV*  $\eta$ ))))

( $\lambda\cdot. enat$  *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  $\eta$ )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform UNIV
(nlists UNIV  $\eta$ ) and ?bound-conv1= $\lambda$ -. 2 and ?q3 = 2 * q and bound-sim =  $\lambda$ -.
 $\infty$ ,
simplified]
  , goal-cases)
  case (1  $\eta$ )
  have [simp]:  $\mathcal{I}$ -uniform UNIV (nlists UNIV  $\eta$ )  $\vdash_c$  CIPHER.KEY.key-oracle  $\eta$  s
 $\checkmark$ 
  if pred-option ( $\lambda x$ . length  $x = \eta$ ) s for s  $\eta$ 
  apply(rule WT-calleeI)
  subgoal for call using that by(cases s; cases call; clarsimp; auto simp add:
key-def in-nlists-UNIV)
  done
  have *: callee-invariant-on (CIPHER.KEY.key-oracle  $\eta \oplus_{\mathcal{O}}$  CIPHER.KEY.key-oracle
 $\eta$ ) (pred-option ( $\lambda x$ . length  $x = \eta$ ))
  ( $\mathcal{I}$ -uniform UNIV (nlists UNIV  $\eta$ )  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform UNIV (nlists UNIV  $\eta$ )) for
 $\eta$ 
  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  $\eta$ )
  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  $\eta$ )
  show ?case by(rule WT-intro)+

  case (4  $\eta$ )
  show ?case by interaction-bound-converter code-simp

  case (5  $\eta$ )
  show ?case by (simp add: mult-2-right)

  case (6  $\eta$ )
  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.