

Constructive Cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar

February 23, 2021

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

1	Resources	3
1.1	Type definition	3
1.2	Functor	3
1.3	Relator	4
1.4	Losslessness	7
1.5	Operations	8
1.6	Well-typing	9
2	Converters	11
2.1	Type definition	11
2.2	Functor	11
2.3	Set functions with interfaces	12
2.4	Relator	13
2.5	Well-typing	17
2.6	Losslessness	19
2.7	Operations	20
2.8	Attaching converters to resources	26
2.9	Composing converters	27
2.10	Interaction bound	29

3	Equivalence of converters restricted by interfaces	32
4	Trace equivalence for resources	40
5	Distinguisher	43
6	Wiring	44
6.1	Notation	44
6.2	Wiring primitives	45
6.3	Characterization of wirings	50
7	Security	52
7.1	Composition theorems	54
8	Examples	56
8.1	Random oracle resource	56
8.2	Key resource	56
8.3	Channel resource	56
8.3.1	Generic channel	57
8.3.2	Insecure channel	57
8.3.3	Authenticated channel	58
8.3.4	Secure channel	58
8.4	Cipher converter	59
8.5	Message authentication converter	60
9	Security of one-time-pad encryption	61
10	Security of message authentication	63
11	Secure composition: Encrypt then MAC	71

```

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)
  <proof>

```

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'))
  <proof>

```

```

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)
  <proof>

```

```

end

```

```

lemma map-resource-id1: map-resource id f res = map-resource' f res
  <proof>

```

```

lemma map-resource-id [simp]: map-resource id id res = res
  <proof>

```

```

lemma map-resource-compose [simp]:
  map-resource a b (map-resource a' b' res) = map-resource (a'  $\circ$  a) (b  $\circ$  b') res

```

$\langle proof \rangle$

functor *resource*: *map-resource* $\langle proof \rangle$

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*

$\langle proof \rangle$

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*

$\langle proof \rangle$

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*)

$\langle proof \rangle$

lemma *rel-resource-eq1*: *rel-resource* (=) = *rel-resource'*

$\langle proof \rangle$

lemma *rel-resource-eq*: *rel-resource* (=) (=) = (=)

$\langle proof \rangle$

lemma *rel-resource-mono*:

assumes $A' \leq A\ B \leq B'$

shows *rel-resource* *A B* \leq *rel-resource* *A' B'*

$\langle proof \rangle$

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

$\langle proof \rangle$

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>

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

$rel-resource\ A\ B\ res1\ (map-resource'\ f\ res2) = rel-resource\ A\ (\lambda x\ y. B\ x\ (f\ y))\ res1\ res2$
 <proof>

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>

lemma *left-unique-rel-resource*:

$\llbracket left-total\ A; left-unique\ B \rrbracket \implies left-unique\ (rel-resource\ A\ B)$
 <proof>

lemma *right-unique-rel-resource*:

$\llbracket right-total\ A; right-unique\ B \rrbracket \implies right-unique\ (rel-resource\ A\ B)$
 <proof>

lemma *bi-unique-rel-resource [transfer-rule]*:

$\llbracket bi-total\ A; bi-unique\ B \rrbracket \implies bi-unique\ (rel-resource\ A\ B)$
 <proof>

definition *rel-witness-resource* :: $('a \Rightarrow 'e \Rightarrow bool) \Rightarrow ('e \Rightarrow 'c \Rightarrow bool) \Rightarrow ('b \Rightarrow 'd \Rightarrow bool) \Rightarrow ('a, 'b)\ resource \times ('c, 'd)\ resource \Rightarrow ('e, 'b \times 'd)\ resource$ **where**
 $rel-witness-resource\ A\ A'\ B = corec-resource\ (\lambda(res1, res2).$
 $map-spmf\ (map-prod\ id\ Inr \circ rel-witness-prod) \circ$
 $rel-witness-spmf\ (rel-prod\ B\ (rel-resource\ (A\ OO\ A')\ B)) \circ$
 $rel-witness-fun\ A\ A'\ (run-resource\ res1, run-resource\ res2))$

lemma *rel-witness-resource-sel [simp]*:

$run-resource\ (rel-witness-resource\ A\ A'\ B\ (res1, res2)) =$
 $map-spmf\ (map-prod\ id\ (rel-witness-resource\ A\ A'\ B) \circ rel-witness-prod) \circ$
 $rel-witness-spmf\ (rel-prod\ B\ (rel-resource\ (A\ OO\ A')\ B)) \circ$
 $rel-witness-fun\ A\ A'\ (run-resource\ res1, run-resource\ res2)$
 <proof>

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>

lemma *rel-resource-neg-distr*:

assumes A : *left-unique A right-total A*
and A' : *right-unique A' left-total A'*
shows $rel-resource (A OO A') (B OO B') \leq rel-resource A B OO rel-resource A' B'$
<proof>

lemma *left-total-rel-resource*:

$\llbracket left-unique A; right-total A; left-total B \rrbracket \implies left-total (rel-resource A B)$
<proof>

lemma *right-total-rel-resource*:

$\llbracket right-unique A; left-total A; right-total B \rrbracket \implies right-total (rel-resource A B)$
<proof>

lemma *bi-total-rel-resource [transfer-rule]*:

$\llbracket bi-total A; bi-unique A; bi-total B \rrbracket \implies bi-total (rel-resource A B)$
<proof>

context includes *lifting-syntax begin*

lemma *Resource-parametric [transfer-rule]*:

$((A \implies rel-spmf (rel-prod B (rel-resource A B))) \implies rel-resource A B)$
Resource Resource
<proof>

lemma *run-resource-parametric [transfer-rule]*:

$(rel-resource A B \implies A \implies rel-spmf (rel-prod B (rel-resource A B)))$
run-resource run-resource
<proof>

lemma *corec-resource-parametric [transfer-rule]*:

$((S \implies A \implies rel-spmf (rel-prod B (rel-sum (rel-resource A B) S))) \implies S \implies rel-resource A B)$
corec-resource corec-resource
<proof>

lemma *map-resource-parametric [transfer-rule]*:

$((A' \implies A) \implies (B \implies B') \implies rel-resource A B \implies rel-resource A' B')$
map-resource map-resource
<proof>

lemma *map-resource'-parametric [transfer-rule]*:

$((B \implies B') \implies rel-resource (=) B \implies rel-resource (=) B')$
map-resource'
<proof>

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
 $\langle \text{proof} \rangle$

end

lemma *rel-resource-Grp*:
 $\text{rel-resource} (\text{conversep} (\text{BNF-Def.Grp } \text{UNIV } f)) (\text{BNF-Def.Grp } \text{UNIV } g) =$
 $\text{BNF-Def.Grp } \text{UNIV} (\text{map-resource } f \ g)$
 $\langle \text{proof} \rangle$

1.4 Losslessness

coinductive *lossless-resource* :: $('a, 'b) \mathcal{I} \Rightarrow ('a, 'b) \text{resource} \Rightarrow \text{bool}$
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*
 $\langle \text{proof} \rangle$

lemma *lossless-resourceD*:
 $\llbracket \text{lossless-resource } \mathcal{I} \ \text{res}; a \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$
 $\Longrightarrow \text{lossless-spmf} (\text{run-resource } \text{res } a) \wedge (\forall (x, \text{res}') \in \text{set-spmf} (\text{run-resource } \text{res}$
 $a). \text{lossless-resource } \mathcal{I} \ \text{res}')$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *lossless-resource-mono'*:
 $\llbracket \text{lossless-resource } \mathcal{I}' \ \text{res}; \mathcal{I} \leq \mathcal{I}' \rrbracket \Longrightarrow \text{lossless-resource } \mathcal{I} \ \text{res}$
 $\langle \text{proof} \rangle$

1.5 Operations

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

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

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

end

lemma $resource\text{-of-oracle-parametric}$ [*transfer-rule*]: **includes** *lifting-syntax* **shows**

$((S \text{ ==== } A \text{ ==== } rel\text{-spm}f (rel\text{-prod } B \ S)) \text{ ==== } S \text{ ==== } rel\text{-resource } A \ B) \text{ resource-of-oracle } resource\text{-of-oracle}$
 $\langle proof \rangle$

lemma $map\text{-resource-resource-of-oracle}$:

$map\text{-resource } f \ g \ (resource\text{-of-oracle } oracle \ s) = resource\text{-of-oracle} (map\text{-fun } id \ (map\text{-fun } f \ (map\text{-spm}f (map\text{-prod } g \ id)))) \ oracle) \ s$

for $s :: 's$

$\langle proof \rangle$

lemma (**in** *callee-invariant-on*) $lossless\text{-resource-of-oracle}$:

assumes $*$: $\bigwedge s \ x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; \ I \ s \rrbracket \implies lossless\text{-spm}f (callee \ s \ x)$

and $I \ s$

shows $lossless\text{-resource } \mathcal{I} (resource\text{-of-oracle } callee \ s)$

$\langle proof \rangle$

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

lemma $resource\text{-of-oracle-rprodl}$: **includes** *lifting-syntax* **shows**

$resource\text{-of-oracle} ((rprodl \ ----> \ id \ ----> \ map\text{-spm}f (map\text{-prod } id \ lprodr)) \ oracle) ((s1, s2), s3) =$

$resource\text{-of-oracle } oracle \ (s1, s2, s3)$

$\langle proof \rangle$

lemma $resource\text{-of-oracle-extend-state-oracle}$ [*simp*]:

$resource\text{-of-oracle} (extend\text{-state-oracle } oracle) (s', s) = resource\text{-of-oracle } oracle \ s$

$\langle proof \rangle$

end

lemma $exec\text{-gpv-resource-of-oracle}$:

$exec\text{-gpv } run\text{-resource } gpv \ (resource\text{-of-oracle } oracle \ s) = map\text{-spm}f (map\text{-prod } id \ (resource\text{-of-oracle } oracle)) (exec\text{-gpv } oracle \ gpv \ s)$

$\langle proof \rangle$

primcorec $parallel\text{-resource} :: ('a, 'b) \text{ resource} \Rightarrow ('c, 'd) \text{ resource} \Rightarrow ('a + 'c, 'b + 'd) \text{ resource}$ **where**

$run\text{-resource} (parallel\text{-resource } res1 \ res2) =$

$(\lambda ac. \text{case } ac \ \text{of } Inl \ a \Rightarrow map\text{-spm}f (map\text{-prod } Inl \ (\lambda res1'. parallel\text{-resource } res1'$

res2)) (run-resource res1 a)
 | Inr c ⇒ map-spmf (map-prod Inr (λres2'. parallel-resource res1 res2'))
 (run-resource res2 c))

lemma *parallel-resource-parametric* [transfer-rule]: **includes** *lifting-syntax* **shows**
 (rel-resource A B ===> rel-resource C D ===> rel-resource (rel-sum A C)
 (rel-sum B D))
 parallel-resource parallel-resource
 ⟨proof⟩

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)
 ⟨proof⟩

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))
 ⟨proof⟩

lemma *lossless-parallel-resource*:
assumes *lossless-resource* \mathcal{I} res1 *lossless-resource* \mathcal{I}' res2
shows *lossless-resource* $(\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}')$ (parallel-resource res1 res2)
 ⟨proof⟩

1.6 Well-typing

coinductive *WT-resource* :: ('a, 'b) \mathcal{I} ⇒ ('a, 'b) resource ⇒ bool (- /⊢res - √
 [100, 0] 99)

for \mathcal{I} **where**

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

if $\bigwedge q r \text{res}' . \llbracket q \in \text{outs-}\mathcal{I} \ \mathcal{I}; (r, \text{res}') \in \text{set-spmf} (\text{run-resource } \text{res } q) \rrbracket \implies r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q \wedge \mathcal{I} \vdash_{\text{res}} \text{res}' \checkmark$

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 $\bigwedge \text{res } q r \text{res}' . \llbracket X \text{res}; q \in \text{outs-}\mathcal{I} \ \mathcal{I}; (r, \text{res}') \in \text{set-spmf} (\text{run-resource } \text{res } q) \rrbracket$

$\implies r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q \wedge (X \text{res}' \vee \mathcal{I} \vdash_{\text{res}} \text{res}' \checkmark)$

shows $\mathcal{I} \vdash_{\text{res}} \text{res} \checkmark$

⟨proof⟩

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 $r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ q \wedge \mathcal{I} \vdash_{\text{res}} \text{res}' \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-resource-of-oracle* [*simp*]:
assumes $\bigwedge s. \mathcal{I} \vdash_c \text{oracle} \ s \checkmark$
shows $\mathcal{I} \vdash_{\text{res}} \text{resource-of-oracle} \ \text{oracle} \ s \checkmark$
 $\langle \text{proof} \rangle$

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

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

lemma (**in** *callee-invariant-on*) *WT-resource-of-oracle*:
 $I \ s \implies \mathcal{I} \vdash_{\text{res}} \text{resource-of-oracle} \ \text{callee} \ s \checkmark$
 $\langle \text{proof} \rangle$

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 $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_{\text{res}} \text{parallel-resource} \ \text{res}1 \ \text{res}2 \checkmark$
 $\langle \text{proof} \rangle$

lemma *callee-invariant-run-resource*: *callee-invariant-on run-resource* $(\lambda \text{res}. \ \mathcal{I} \vdash_{\text{res}} \text{res} \checkmark) \ \mathcal{I}$
 $\langle \text{proof} \rangle$

lemma *callee-invariant-run-lossless-resource*:
callee-invariant-on run-resource $(\lambda \text{res}. \ \text{lossless-resource} \ \mathcal{I} \ \text{res} \wedge \mathcal{I} \vdash_{\text{res}} \text{res} \checkmark) \ \mathcal{I}$
 $\langle \text{proof} \rangle$

interpretation *run-lossless-resource*:
callee-invariant-on run-resource $\lambda \text{res}. \ \text{lossless-resource} \ \mathcal{I} \ \text{res} \wedge \mathcal{I} \vdash_{\text{res}} \text{res} \checkmark \ \mathcal{I}$
for \mathcal{I}
 $\langle \text{proof} \rangle$

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*)
(*proof*)

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'*))
(*proof*)

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*)
(*proof*)

end

lemma *map-converter-id14*: *map-converter id b out id res* = *map-converter' b out*
res
(*proof*)

lemma *map-converter-id* [*simp*]: *map-converter id id id id conv* = *conv*
(*proof*)

lemma *map-converter-compose* [*simp*]:
 $map\text{-}converter\ a\ b\ f\ g\ (map\text{-}converter\ a'\ b'\ f'\ g'\ conv) = map\text{-}converter\ (a' \circ a)$
 $(b \circ b')\ (f \circ f')\ (g' \circ g)\ conv$
 ⟨*proof*⟩

functor *converter*: *map-converter* ⟨*proof*⟩

2.3 Set functions with interfaces

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

qualified inductive *outsp-converter* :: $'out \Rightarrow ('a, 'b, 'out, 'in)\ converter \Rightarrow bool$
for *out* **where**

Out: *outsp-converter out conv* **if** $out \in outs\text{-}gpv\ \mathcal{I}'\ (run\text{-}converter\ conv\ a)\ a \in outs\text{-}\mathcal{I}\ \mathcal{I}$

| *Cont*: *outsp-converter out conv*

if $(b, conv') \in results\text{-}gpv\ \mathcal{I}'\ (run\text{-}converter\ conv\ a)\ outsp\text{-}converter\ out\ conv'\ a \in outs\text{-}\mathcal{I}\ \mathcal{I}$

definition *outs-converter* :: $('a, 'b, 'out, 'in)\ converter \Rightarrow 'out\ set$
where *outs-converter conv* $\equiv \{x.\ outsp\text{-}converter\ x\ conv\}$

qualified inductive *resultsp-converter* :: $'b \Rightarrow ('a, 'b, 'out, 'in)\ converter \Rightarrow bool$
for *b* **where**

Result: *resultsp-converter b conv*

if $(b, conv') \in results\text{-}gpv\ \mathcal{I}'\ (run\text{-}converter\ conv\ a)\ a \in outs\text{-}\mathcal{I}\ \mathcal{I}$

| *Cont*: *resultsp-converter b conv*

if $(b', conv') \in results\text{-}gpv\ \mathcal{I}'\ (run\text{-}converter\ conv\ a)\ resultsp\text{-}converter\ b\ conv'\ a \in outs\text{-}\mathcal{I}\ \mathcal{I}$

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

end

lemma *outsp-converter-outs-converter-eq* [*pred-set-conv*]: *Converter.outsp-converter*
 $\mathcal{I}\ \mathcal{I}'\ x = (\lambda conv.\ x \in outs\text{-}converter\ \mathcal{I}\ \mathcal{I}'\ conv)$
 ⟨*proof*⟩

context **begin**
 ⟨*ML*⟩

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
  I I' x (Converter conv)

lemma resultsp-converter-results-converter-eq [pred-set-conv]:
  Converter.resultsp-converter I I' x = (λconv. x ∈ results-converter I I' conv)
  ⟨proof⟩

context begin
  ⟨ML⟩

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
  I I' x (Converter conv)

```

2.4 Relator

```

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

```

```

lemma rel-converter-coinduct [consumes 1, case-names rel-converter, coinduct pred:
  rel-converter]:
  assumes X conv1 conv2
  and ∧ conv1 conv2. X conv1 conv2 ⇒
    rel-fun A (rel-gpv'' (rel-prod B (λconv1 conv2. X conv1 conv2 ∨ rel-converter
  A B C R conv1 conv2)) C R)
    (run-converter conv1) (run-converter conv2)
  shows rel-converter A B C R conv1 conv2
  ⟨proof⟩

```

lemma *rel-converter-simps* [*simp, code*]:
 $rel\text{-}converter\ A\ B\ C\ R\ (Converter\ f)\ (Converter\ g) \longleftrightarrow$
 $rel\text{-}fun\ A\ (rel\text{-}gpv''\ (rel\text{-}prod\ B\ (rel\text{-}converter\ A\ B\ C\ R))\ C\ R)\ f\ g$
 $\langle proof \rangle$

lemma *rel-converterD*:
 $rel\text{-}converter\ A\ B\ C\ R\ conv1\ conv2$
 $\implies rel\text{-}fun\ A\ (rel\text{-}gpv''\ (rel\text{-}prod\ B\ (rel\text{-}converter\ A\ B\ C\ R))\ C\ R)\ (run\text{-}converter$
 $conv1)\ (run\text{-}converter\ conv2)$
 $\langle proof \rangle$

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

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

lemma *rel-converter-mono* [*relator-mono*]:
assumes $A' \leq A\ B \leq B'\ C \leq C'\ R' \leq R$
shows $rel\text{-}converter\ A\ B\ C\ R \leq rel\text{-}converter\ A'\ B'\ C'\ R'$
 $\langle proof \rangle$

lemma *rel-converter-conversep*: $rel\text{-}converter\ A^{-1-1}\ B^{-1-1}\ C^{-1-1}\ R^{-1-1} = (rel\text{-}converter$
 $A\ B\ C\ R)^{-1-1}$
 $\langle proof \rangle$

lemma *rel-converter-map-converter'1*:
 $rel\text{-}converter\ A\ B\ C\ R\ (map\text{-}converter'\ f\ g\ conv1)\ conv2 = rel\text{-}converter\ A\ (\lambda x.$
 $B\ (f\ x))\ (\lambda x.\ C\ (g\ x))\ R\ conv1\ conv2$
(is *?lhs = ?rhs*)
 $\langle proof \rangle$

lemma *rel-converter-map-converter'2*:
 $rel\text{-}converter\ A\ B\ C\ R\ conv1\ (map\text{-}converter'\ f\ g\ conv2) = rel\text{-}converter\ A\ (\lambda x\ y.$
 $B\ x\ (f\ y))\ (\lambda x\ y.\ C\ x\ (g\ y))\ R\ conv1\ conv2$
 $\langle proof \rangle$

lemmas $converter\text{-}rel\text{-}map' = rel\text{-}converter\text{-}map\text{-}converter'1[abs\text{-}def]\ rel\text{-}converter\text{-}map\text{-}converter'2$

lemma *rel-converter-pos-distr* [*relator-distr*]:
 $rel\text{-}converter\ A\ B\ C\ R\ OO\ rel\text{-}converter\ A'\ B'\ C'\ R' \leq rel\text{-}converter\ (A\ OO\ A')$
 $(B\ OO\ B')\ (C\ OO\ C')\ (R\ OO\ R')$
 $\langle proof \rangle$

lemma *left-unique-rel-converter*:
 $\llbracket left\text{-}total\ A;\ left\text{-}unique\ B;\ left\text{-}unique\ C;\ left\text{-}total\ R \rrbracket \implies left\text{-}unique\ (rel\text{-}converter$
 $A\ B\ C\ R)$
 $\langle proof \rangle$

lemma *right-unique-rel-converter*:

$\llbracket \text{right-total } A; \text{right-unique } B; \text{right-unique } C; \text{right-total } R \rrbracket \implies \text{right-unique}$
 $(\text{rel-converter } A \ B \ C \ R)$
 $\langle \text{proof} \rangle$

lemma *bi-unique-rel-converter [transfer-rule]*:

$\llbracket \text{bi-total } A; \text{bi-unique } B; \text{bi-unique } C; \text{bi-total } R \rrbracket \implies \text{bi-unique } (\text{rel-converter } A$
 $B \ C \ R)$
 $\langle \text{proof} \rangle$

definition *rel-witness-converter* $:: ('a \Rightarrow 'e \Rightarrow \text{bool}) \Rightarrow ('e \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow$
 $'d \Rightarrow \text{bool}) \Rightarrow ('out \Rightarrow 'out' \Rightarrow \text{bool}) \Rightarrow ('in \Rightarrow 'in'' \Rightarrow \text{bool}) \Rightarrow ('in'' \Rightarrow 'in' \Rightarrow$
 $\text{bool})$

$\Rightarrow ('a, 'b, 'out, 'in) \text{ converter} \times ('c, 'd, 'out', 'in') \text{ converter} \Rightarrow ('e, 'b \times 'd, 'out$
 $\times 'out', 'in'') \text{ converter}$ **where**

$\text{rel-witness-converter } A \ A' \ B \ C \ R \ R' = \text{corec-converter } (\lambda(\text{conv1}, \text{conv2}).$

$\text{map-gpv } (\text{map-prod } \text{id } \text{Inr} \circ \text{rel-witness-prod}) \ \text{id} \circ$

$\text{rel-witness-gpv } (\text{rel-prod } B \ (\text{rel-converter } (A \ OO \ A') \ B \ C \ (R \ OO \ R'))) \ C \ R \ R'$

\circ

$\text{rel-witness-fun } A \ A' \ (\text{run-converter } \text{conv1}, \text{run-converter } \text{conv2}))$

lemma *rel-witness-converter-sel [simp]*:

$\text{run-converter } (\text{rel-witness-converter } A \ A' \ B \ C \ R \ R' \ (\text{conv1}, \text{conv2})) =$

$\text{map-gpv } (\text{map-prod } \text{id} \ (\text{rel-witness-converter } A \ A' \ B \ C \ R \ R') \circ \text{rel-witness-prod})$
 $\text{id} \circ$

$\text{rel-witness-gpv } (\text{rel-prod } B \ (\text{rel-converter } (A \ OO \ A') \ B \ C \ (R \ OO \ R'))) \ C \ R \ R'$

\circ

$\text{rel-witness-fun } A \ A' \ (\text{run-converter } \text{conv1}, \text{run-converter } \text{conv2})$

$\langle \text{proof} \rangle$

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* $(\lambda b \ (b', c). b = b' \wedge B \ b' \ c) \ (\lambda c \ (c',$
 $d). c = c' \wedge C \ c' \ d) \ R \ \text{conv} \ (\text{rel-witness-converter } A \ A' \ B \ C \ R \ R' \ (\text{conv}, \text{conv}'))$
(is ?thesis1)

and *rel-witness-converter2: rel-converter A'* $(\lambda(b, c') \ c. c = c' \wedge B \ b \ c') \ (\lambda(c, d')$
 $d. d = d' \wedge C \ c \ d') \ R' \ (\text{rel-witness-converter } A \ A' \ B \ C \ R \ R' \ (\text{conv}, \text{conv}')) \ \text{conv}'$
(is ?thesis2)

$\langle \text{proof} \rangle$

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 $\text{rel-converter } (A \text{ OO } A') (B \text{ OO } B') (C \text{ OO } C') (R \text{ OO } R') \leq \text{rel-converter}$
 $A \ B \ C \ R \ \text{OO} \ \text{rel-converter } A' \ B' \ C' \ R'$
 $\langle \text{proof} \rangle$

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
 $\implies \text{left-total } (\text{rel-converter } A \ B \ C \ R)$
 $\langle \text{proof} \rangle$

lemma *right-total-rel-converter*:

$\llbracket \text{right-unique } A; \text{left-total } A; \text{right-total } B; \text{right-total } C; \text{right-unique } R; \text{left-total}$
 $R \rrbracket$
 $\implies \text{right-total } (\text{rel-converter } A \ B \ C \ R)$
 $\langle \text{proof} \rangle$

lemma *bi-total-rel-converter* [*transfer-rule*]:

$\llbracket \text{bi-total } A; \text{bi-unique } A; \text{bi-total } B; \text{bi-total } C; \text{bi-total } R; \text{bi-unique } R \rrbracket$
 $\implies \text{bi-total } (\text{rel-converter } A \ B \ C \ R)$
 $\langle \text{proof} \rangle$

inductive *pred-converter* :: $'a \ \text{set} \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow ('out \Rightarrow \text{bool}) \Rightarrow 'in \ \text{set} \Rightarrow$
 $('a, 'b, 'out, 'in) \ \text{converter} \Rightarrow \text{bool}$

for $A \ B \ C \ R \ \text{conv}$ **where**

$\text{pred-converter } A \ B \ C \ R \ \text{conv}$ **if**

$\forall x \in \text{results-converter } (\mathcal{I}\text{-uniform } A \ \text{UNIV}) (\mathcal{I}\text{-uniform } \text{UNIV } R) \ \text{conv. } B \ x$

$\forall out \in \text{outs-converter } (\mathcal{I}\text{-uniform } A \ \text{UNIV}) (\mathcal{I}\text{-uniform } \text{UNIV } R) \ \text{conv. } C \ out$

lemma *pred-gpv'-mono-weak*:

$\text{pred-gpv}' \ A \ C \ R \leq \text{pred-gpv}' \ A' \ C' \ R$ **if** $A \leq A' \ C \leq C'$

$\langle \text{proof} \rangle$

lemma *Domainp-rel-converter-le*:

$\text{Domainp } (\text{rel-converter } A \ B \ C \ R) \leq \text{pred-converter } (\text{Collect } (\text{Domainp } A))$
 $(\text{Domainp } B) (\text{Domainp } C) (\text{Collect } (\text{Domainp } R))$

(is ?lhs \leq ?rhs)

$\langle \text{proof} \rangle$

lemma *rel-converter-Grp*:

$\text{rel-converter } (\text{BNF-Def.Grp } \text{UNIV } f)^{-1-1} (\text{BNF-Def.Grp } B \ g) (\text{BNF-Def.Grp}$
 $C \ h) (\text{BNF-Def.Grp } \text{UNIV } k)^{-1-1} =$

$\text{BNF-Def.Grp } \{ \text{conv. results-converter } (\mathcal{I}\text{-uniform } (\text{range } f) \ \text{UNIV}) (\mathcal{I}\text{-uniform}$
 $\text{UNIV } (\text{range } k)) \ \text{conv} \subseteq B \wedge$

$\text{outs-converter } (\mathcal{I}\text{-uniform } (\text{range } f) \ \text{UNIV}) (\mathcal{I}\text{-uniform } \text{UNIV } (\text{range } k)) \ \text{conv}$
 $\subseteq C \}$

$(\text{map-converter } f \ g \ h \ k)$

(is ?lhs = ?rhs)

including *lifting-syntax*

$\langle proof \rangle$

context

includes *lifting-syntax*

notes [*transfer-rule*] = *map-gpv-parametric'*

begin

lemma *Converter-parametric* [*transfer-rule*]:

$((A \text{====>} \text{rel-gpv}'' (\text{rel-prod } B (\text{rel-converter } A B C R)) C R) \text{====>} \text{rel-converter } A B C R)$ *Converter Converter*
 $\langle proof \rangle$

lemma *run-converter-parametric* [*transfer-rule*]:

$(\text{rel-converter } A B C R \text{====>} A \text{====>} \text{rel-gpv}'' (\text{rel-prod } B (\text{rel-converter } A B C R)) C R)$
run-converter run-converter
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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'
 $\langle proof \rangle$

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
 $\langle proof \rangle$

end

2.5 Well-typing

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

(\cdot , / \vdash_C / $\cdot \sqrt{[100, 0, 0] 99}$)

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

WT-converterI: $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark$ **if**
 $\bigwedge q. q \in \text{outs-}\mathcal{I} \mathcal{I} \implies \mathcal{I}' \vdash_g \text{run-converter} \text{conv} q \checkmark$
 $\bigwedge q r \text{conv}' . \llbracket q \in \text{outs-}\mathcal{I} \mathcal{I}; (r, \text{conv}') \in \text{results-gpv} \mathcal{I}' (\text{run-converter} \text{conv} q) \rrbracket$
 $\implies r \in \text{responses-}\mathcal{I} \mathcal{I} q \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \checkmark$

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 $\bigwedge \text{conv} q r \text{conv}' . \llbracket X \text{conv}; q \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket$
 $\implies \mathcal{I}' \vdash_g \text{run-converter} \text{conv} q \checkmark \wedge$
 $((r, \text{conv}') \in \text{results-gpv} \mathcal{I}' (\text{run-converter} \text{conv} q) \longrightarrow r \in \text{responses-}\mathcal{I} \mathcal{I} q \wedge$
 $(X \text{conv}' \vee \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \checkmark))$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark$
 $\langle \text{proof} \rangle$

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*: $(r, \text{conv}') \in \text{results-gpv} \mathcal{I}' (\text{run-converter} \text{conv} q)$
 $\implies r \in \text{responses-}\mathcal{I} \mathcal{I} q \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-converterD'*:

assumes $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark q \in \text{outs-}\mathcal{I} \mathcal{I}$
shows $\mathcal{I}' \vdash_g \text{run-converter} \text{conv} q \checkmark \wedge (\forall (r, \text{conv}') \in \text{results-gpv} \mathcal{I}' (\text{run-converter} \text{conv} q). r \in \text{responses-}\mathcal{I} \mathcal{I} q \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \checkmark)$
 $\langle \text{proof} \rangle$

lemma *WT-converter-bot1* [*simp*]: *bot*, $\mathcal{I} \vdash_C \text{conv} \checkmark$

$\langle \text{proof} \rangle$

lemma *WT-converter-mono*:

$\llbracket \mathcal{I}1, \mathcal{I}2 \vdash_C \text{conv} \checkmark; \mathcal{I}1' \leq \mathcal{I}1; \mathcal{I}2 \leq \mathcal{I}2' \rrbracket \implies \mathcal{I}1', \mathcal{I}2' \vdash_C \text{conv} \checkmark$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\mathcal{I} \mathcal{I}'$ for $\mathcal{I} \mathcal{I}'$
 $\langle \text{proof} \rangle$

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

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
 $\langle \text{proof} \rangle$

2.6 Losslessness

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

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

lemma *plossless-converterD*:
 $\llbracket \text{plossless-converter} \mathcal{I} \mathcal{I}' \text{ conv}; a \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket$
 $\implies \text{plossless-gpv} \mathcal{I}' (\text{run-converter conv } a) \wedge$
 $(\forall (b, \text{conv}') \in \text{results-gpv} \mathcal{I}' (\text{run-converter conv } a). \text{plossless-converter} \mathcal{I} \mathcal{I}' \text{ conv}')$
 $\langle \text{proof} \rangle$

lemma *plossless-converter-bot1* [*simp*]: *plossless-converter bot* \mathcal{I} conv
 $\langle \text{proof} \rangle$

lemma *plossless-converter-mono*:
assumes $*$: *plossless-converter* $\mathcal{I}1 \mathcal{I}2$ 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 : $\mathcal{I}1, \mathcal{I}2 \vdash_C$ conv \checkmark
shows *plossless-converter* $\mathcal{I}1' \mathcal{I}2'$ conv
 $\langle \text{proof} \rangle$

lemma *raw-converter-invariant-run-plossless-converter: raw-converter-invariant* \mathcal{I}
 \mathcal{I}' *run-converter* ($\lambda conv. plossless-converter \mathcal{I} \mathcal{I}' conv \wedge \mathcal{I}, \mathcal{I}' \vdash_C conv \checkmark$)
 $\langle proof \rangle$

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}'
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *map-converter-of-callee:*

map-converter $f \ g \ h \ k$ (*converter-of-callee* *callee* s) =
converter-of-callee (*map-fun* id (*map-fun* f (*map-gpv'* (*map-prod* $g \ id$) $h \ k$))
callee) s
 $\langle proof \rangle$

lemma *WT-converter-of-callee:*

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

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

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

lemma parallel-callee-parametric [transfer-rule]: **includes lifting-syntax shows**
 $(\text{rel-converter } A \ B \ C \ R \ ==> \text{ rel-converter } A' \ B' \ C \ R \ ==> \text{ rel-converter}$
 $(\text{rel-sum } A \ A') \ (\text{rel-sum } B \ B') \ C \ R)$
 $\text{parallel-converter } \text{parallel-converter}$
 $\langle \text{proof} \rangle$

lemma parallel-converter-assoc:

$\text{parallel-converter } (\text{parallel-converter } \text{conv1 } \text{conv2}) \ \text{conv3} =$
 $\text{map-converter } \text{rsuml } \text{lsumr } \text{id } \text{id} \ (\text{parallel-converter } \text{conv1} \ (\text{parallel-converter}$
 $\text{conv2 } \text{conv3}))$
 $\langle \text{proof} \rangle$

lemma plossless-parallel-converter [plossless-intro]:

$\llbracket \text{plossless-converter } \mathcal{I}1 \ \mathcal{I} \ \text{conv1}; \text{ plossless-converter } \mathcal{I}2 \ \mathcal{I} \ \text{conv2}; \mathcal{I}1, \mathcal{I} \vdash_C \text{ conv1}$
 $\sqrt{}; \mathcal{I}2, \mathcal{I} \vdash_C \text{ conv2 } \sqrt{} \rrbracket$
 $\Rightarrow \text{plossless-converter } (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \ \mathcal{I} \ (\text{parallel-converter } \text{conv1 } \text{conv2})$
 $\langle \text{proof} \rangle$

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

$\text{run-converter } \text{id-converter} = (\lambda a.$
 $\text{map-gpv } (\text{map-prod } \text{id} \ (\lambda -. \text{id-converter})) \ \text{id} \ (\text{Pause } a \ (\lambda b. \text{ Done } (b, ())))))$

lemma id-converter-parametric [transfer-rule]: $\text{rel-converter } A \ B \ A \ B \ \text{id-converter}$
 id-converter
 $\langle \text{proof} \rangle$

lemma converter-of-callee-id-oracle [simp]:

$\text{converter-of-callee } \text{id-oracle } s = \text{id-converter}$
 $\langle \text{proof} \rangle$

lemma conv-callee-plus-id-left: $\text{converter-of-callee } (\text{plus-intercept } \text{id-oracle } \text{callee})$
 $s =$

$\text{parallel-converter } \text{id-converter} \ (\text{converter-of-callee } \text{callee } s)$
 $\langle \text{proof} \rangle$

lemma conv-callee-plus-id-right: $\text{converter-of-callee } (\text{plus-intercept } \text{callee } \text{id-oracle})$
 $s =$

$\text{parallel-converter} \ (\text{converter-of-callee } \text{callee } s) \ \text{id-converter}$
 $\langle \text{proof} \rangle$

lemma *plossless-id-converter* [*simp, plossless-intro*]: *plossless-converter* $\mathcal{I} \mathcal{I}$ *id-converter*
 ⟨*proof*⟩

lemma *WT-converter-id* [*simp, intro, WT-intro*]: $\mathcal{I}, \mathcal{I} \vdash_C$ *id-converter* \surd
 ⟨*proof*⟩

lemma *WT-map-converter-idD*:
 $\mathcal{I}, \mathcal{I}' \vdash_C$ *map-converter id id f g id-converter* $\surd \implies \mathcal{I} \leq \text{map-}\mathcal{I} f g \mathcal{I}'$
 ⟨*proof*⟩

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

lemma *fail-converter-sel* [*simp*]: *run-converter fail-converter a* = *Fail*
 ⟨*proof*⟩

lemma *fail-converter-parametric* [*transfer-rule*]: *rel-converter A B C R fail-converter*
fail-converter
 ⟨*proof*⟩

lemma *plossless-fail-converter* [*simp*]: *plossless-converter* $\mathcal{I} \mathcal{I}'$ *fail-converter* \longleftrightarrow
 $\mathcal{I} = \text{bot}$ (**is** ?*lhs* \longleftrightarrow ?*rhs*)
 ⟨*proof*⟩

lemma *plossless-fail-converterI* [*plossless-intro*]: *plossless-converter bot* \mathcal{I}' *fail-converter*
 ⟨*proof*⟩

lemma *WT-fail-converter* [*simp, WT-intro*]: $\mathcal{I}, \mathcal{I}' \vdash_C$ *fail-converter* \surd
 ⟨*proof*⟩

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
 ⟨*proof*⟩

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
 ⟨*proof*⟩

And here is the version for parallel composition that assumes disjoint inter-
 faces.

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) = (λ ac. *case ac of*
Inl a \Rightarrow *map-gpv (map-prod Inl (λ conv1'. parallel-converter2 conv1' conv2)) id*

(*left-gpv* (*run-converter* *conv1* *a*))
 | *Inr* *b* \Rightarrow *map-gpv* (*map-prod* *Inr* (λ *conv2'*. *parallel-converter2* *conv1* *conv2'*))
id (*right-gpv* (*run-converter* *conv2* *b*)))

lemma *parallel-converter2-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 (*rel-converter* *A* *B* *C* *R* \Longrightarrow *rel-converter* *A'* *B'* *C'* *R'*
 \Longrightarrow *rel-converter* (*rel-sum* *A* *A'*) (*rel-sum* *B* *B'*) (*rel-sum* *C* *C'*) (*rel-sum* *R*
R'))
parallel-converter2 *parallel-converter2*
 ⟨*proof*⟩

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)
 ⟨*proof*⟩

lemma *WT-converter-parallel-converter2* [*WT-intro*]:
assumes $\mathcal{I}1, \mathcal{I}2 \vdash_C \text{conv1} \checkmark$
and $\mathcal{I}1', \mathcal{I}2' \vdash_C \text{conv2} \checkmark$
shows $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}1', \mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C \text{parallel-converter2 conv1 conv2} \checkmark$
 ⟨*proof*⟩

lemma *plossless-parallel-converter2* [*plossless-intro*]:
assumes *plossless-converter* $\mathcal{I}1$ $\mathcal{I}1'$ *conv1*
and *plossless-converter* $\mathcal{I}2$ $\mathcal{I}2'$ *conv2*
shows *plossless-converter* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) ($\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2'$) (*parallel-converter2* *conv1*
conv2)
 ⟨*proof*⟩

lemma *parallel-converter2-map1-out*:
parallel-converter2 (*map-converter* *f* *g* *h* *k* *conv1*) *conv2* =
map-converter (*map-sum* *f* *id*) (*map-sum* *g* *id*) (*map-sum* *h* *id*) (*map-sum* *k* *id*)
 (*parallel-converter2* *conv1* *conv2*)
 ⟨*proof*⟩

lemma *parallel-converter2-map2-out*:
parallel-converter2 *conv1* (*map-converter* *f* *g* *h* *k* *conv2*) =
map-converter (*map-sum* *id* *f*) (*map-sum* *id* *g*) (*map-sum* *id* *h*) (*map-sum* *id* *k*)
 (*parallel-converter2* *conv1* *conv2*)
 ⟨*proof*⟩

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

lemma *left-interface-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 $(rel\text{-converter } A \ B \ C \ R \ ==\Rightarrow rel\text{-converter } A \ B \ (rel\text{-sum } C \ C') \ (rel\text{-sum } R \ R'))$
left-interface *left-interface*
 $\langle proof \rangle$

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

lemma *right-interface-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 $(rel\text{-converter } A \ B \ C' \ R' \ ==\Rightarrow rel\text{-converter } A \ B \ (rel\text{-sum } C \ C') \ (rel\text{-sum } R \ R'))$ *right-interface* *right-interface*
 $\langle proof \rangle$

lemma *parallel-converter2-alt-def*:
 $parallel\text{-converter2 } conv1 \ conv2 = parallel\text{-converter } (left\text{-interface } conv1) \ (right\text{-interface } conv2)$
 $\langle proof \rangle$

lemma *conv-callee-parallel-id-left*: *converter-of-callee* (*parallel-intercept* *id-oracle* *callee*) (s, s') =
 $parallel\text{-converter2 } (id\text{-converter}) \ (converter\text{-of-callee } callee \ s')$
 $\langle proof \rangle$

lemma *conv-callee-parallel-id-right*: *converter-of-callee* (*parallel-intercept* *callee* *id-oracle*) (s, s') =
 $parallel\text{-converter2 } (converter\text{-of-callee } callee \ s) \ (id\text{-converter})$
 $\langle proof \rangle$

lemma *conv-callee-parallel*: *converter-of-callee* (*parallel-intercept* *callee1* *callee2*) (s, s')
 $= parallel\text{-converter2 } (converter\text{-of-callee } callee1 \ s) \ (converter\text{-of-callee } callee2 \ s')$
 $\langle proof \rangle$

lemma *WT-converter-parallel-converter* [*WT-intro*]:
assumes $\mathcal{I}1, \mathcal{I} \vdash_C \ conv1 \ \checkmark$
and $\mathcal{I}2, \mathcal{I} \vdash_C \ conv2 \ \checkmark$
shows $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I} \vdash_C \ parallel\text{-converter } conv1 \ conv2 \ \checkmark$
 $\langle proof \rangle$

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

lemma *WT-converter-of-resource* [*WT-intro*]:
assumes $\mathcal{I} \vdash_{res} \ res \ \checkmark$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \ converter\text{-of-resource } res \ \checkmark$

<proof>

lemma *plossless-converter-of-resource* [*plossless-intro*]:

assumes *lossless-resource* \mathcal{I} *res*

shows *plossless-converter* \mathcal{I} \mathcal{I}' (*converter-of-resource* *res*)

<proof>

lemma *plossless-converter-of-callee*:

assumes $\bigwedge s x. x \in \text{outs-}\mathcal{I} \mathcal{I}1 \implies \text{plossless-gpv } \mathcal{I}2 \text{ (callee } s x) \wedge (\forall (y, s') \in \text{results-gpv } \mathcal{I}2 \text{ (callee } s x). y \in \text{responses-}\mathcal{I} \mathcal{I}1 x)$

shows *plossless-converter* $\mathcal{I}1$ $\mathcal{I}2$ (*converter-of-callee* *callee* *s*)

<proof>

context

fixes $A :: 'a \text{ set}$

and $\mathcal{I} :: ('c, 'd) \mathcal{I}$

begin

primcorec *restrict-converter* :: $('a, 'b, 'c, 'd) \text{ converter} \implies ('a, 'b, 'c, 'd) \text{ converter}$

where

run-converter (*restrict-converter* *cnv*) = $(\lambda a. \text{if } a \in A \text{ then}$

$\text{map-gpv (map-prod id } (\lambda cnv'. \text{restrict-converter } cnv')) \text{ id (restrict-gpv } \mathcal{I}$

$(\text{run-converter } cnv \ a))$

$\text{else Fail})$

end

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

assumes $\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv } \checkmark$

shows $\mathcal{I}, \mathcal{I}' \vdash_C \text{restrict-converter } A \ \mathcal{I}' \ \text{cnv } \checkmark$

<proof>

lemma *pgen-lossless-restrict-gpv* [*simp*]:

$\mathcal{I} \vdash_g \text{gpv } \checkmark \implies \text{pgen-lossless-gpv } b \ \mathcal{I} \text{ (restrict-gpv } \mathcal{I} \ \text{gpv}) = \text{pgen-lossless-gpv } b \ \mathcal{I} \ \text{gpv}$

<proof>

lemma *plossless-restrict-converter* [*simp*]:

assumes *plossless-converter* \mathcal{I} \mathcal{I}' *conv*

and $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \checkmark$

and $\text{outs-}\mathcal{I} \ \mathcal{I} \subseteq A$

shows *plossless-converter* \mathcal{I} \mathcal{I}' (*restrict-converter* $A \ \mathcal{I}' \ \text{conv}$)

<proof>

lemma *plossless-map-converter*:

plossless-converter \mathcal{I} \mathcal{I}' (*map-converter* *f g h k conv*)

if *plossless-converter* (*map- \mathcal{I}* (*inv-into UNIV* *f*) (*inv-into UNIV* *g*) \mathcal{I}) (*map- \mathcal{I}* *h k* \mathcal{I}') *conv inj f*

<proof>

2.8 Attaching converters to resources

primcorec *attach* :: ('a, 'b, 'out, 'in) converter \Rightarrow ('out, 'in) resource \Rightarrow ('a, 'b) resource **where**

run-resource (*attach conv res*) = ($\lambda a.$
 $\text{map-spmf } (\lambda((b, \text{conv}'), \text{res}'). (b, \text{attach conv}' \text{res}')) (\text{exec-gpv run-resource}$
 $(\text{run-converter conv } a) \text{ res})$)

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

(*rel-converter* *A B C R* \implies *rel-resource* *C R* \implies *rel-resource* *A B*) *attach*

<proof>

lemma *attach-map-converter*:

attach (*map-converter* *f g h k conv*) *res* = *map-resource* *f g* (*attach conv* (*map-resource*
h k res))

<proof>

lemma *WT-resource-attach* [*WT-intro*]: $\llbracket \mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \checkmark; \mathcal{I}' \vdash_{\text{res}} \text{res } \checkmark \rrbracket \implies \mathcal{I}$
 $\vdash_{\text{res}} \text{attach conv res } \checkmark$

<proof>

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

assumes *lossless-converter* $\mathcal{I} \mathcal{I}' \text{ conv}$

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

and $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv } \checkmark \mathcal{I}' \vdash_{\text{res}} \text{res } \checkmark$

shows *lossless-resource* $\mathcal{I} (\text{attach conv res})$

<proof>

definition *attach-callee*

:: ('s \Rightarrow 'a \Rightarrow ('b \times 's, 'out, 'in) gpv)

\Rightarrow ('s' \Rightarrow 'out \Rightarrow ('in \times 's') spmf)

\Rightarrow ('s \times 's' \Rightarrow 'a \Rightarrow ('b \times 's \times 's') spmf) **where**

attach-callee callee oracle = ($\lambda(s, s') q. \text{map-spmf } \text{rprodl } (\text{exec-gpv } \text{oracle } (\text{callee } s q) s')$)

lemma *attach-callee-simps* [*simp*]:

attach-callee callee oracle (*s, s'*) *q* = *map-spmf rprodl* (*exec-gpv oracle* (*callee s*
q) *s'*)

<proof>

lemma *attach-CNV-RES*:

attach (*converter-of-callee callee s*) (*resource-of-oracle res s'*) =

resource-of-oracle (*attach-callee callee res*) (*s, s'*)

<proof>

lemma *attach-stateless-callee*:

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

<proof>

lemma *attach-id-converter* [simp]: *attach id-converter res = res*
<proof>

lemma *attach-callee-parallel-intercept*: **includes** *lifting-syntax* **shows**
attach-callee (parallel-intercept callee1 callee2) (plus-oracle oracle1 oracle2) =
(rprodl ----> id ----> map-spmf (map-prod id lprodr)) (plus-oracle (lift-state-oracle
extend-state-oracle (attach-callee callee1 oracle1)) (extend-state-oracle (attach-callee
callee2 oracle2)))
<proof>

lemma *attach-callee-id-oracle* [simp]:
attach-callee id-oracle oracle = extend-state-oracle oracle
<proof>

lemma *attach-parallel2*: *attach (parallel-converter2 conv1 conv2) (parallel-resource*
res1 res2)
= parallel-resource (attach conv1 res1) (attach conv2 res2)
<proof>

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
<proof>

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'))
<proof>

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')
<proof>

lemma *attach-compose*:
attach (comp-converter conv1 conv2) res = attach conv1 (attach conv2 res)

<proof>
including *lifting-syntax*
 <proof>

lemma *comp-converter-assoc*:
 $comp_converter (comp_converter\ conv1\ conv2)\ conv3 = comp_converter\ conv1$
 $(comp_converter\ conv2\ conv3)$
 <proof>
including *lifting-syntax*
 <proof>

lemma *comp-converter-assoc-left*:
assumes $comp_converter\ conv1\ conv2 = conv3$
shows $comp_converter\ conv1 (comp_converter\ conv2\ conv) = comp_converter$
 $conv3\ conv$
 <proof>

lemma *comp-converter-attach-left*:
assumes $comp_converter\ conv1\ conv2 = conv3$
shows $attach\ conv1 (attach\ conv2\ res) = attach\ conv3\ res$
 <proof>

lemmas *comp-converter-egs* =
 $asm_rl[where\ psi=x = y\ for\ x\ y :: (-, -, -, -)\ converter]$
 $comp_converter_assoc_left$
 $comp_converter_attach_left$

lemma *WT-converter-comp* [*WT-intro*]:
 $\llbracket \mathcal{I}, \mathcal{I}' \vdash_C conv \checkmark; \mathcal{I}', \mathcal{I}'' \vdash_C conv' \checkmark \rrbracket \implies \mathcal{I}, \mathcal{I}'' \vdash_C comp_converter\ conv\ conv'$
 \checkmark
 <proof>

lemma *plossless-comp-converter* [*plossless-intro*]:
assumes $plossless_converter\ \mathcal{I}\ \mathcal{I}'\ conv$
and $plossless_converter\ \mathcal{I}'\ \mathcal{I}''\ conv'$
and $\mathcal{I}, \mathcal{I}' \vdash_C conv \checkmark, \mathcal{I}', \mathcal{I}'' \vdash_C conv' \checkmark$
shows $plossless_converter\ \mathcal{I}\ \mathcal{I}'' (comp_converter\ conv\ conv')$
 <proof>

lemma *comp-converter-id-left*: $comp_converter\ id_converter\ conv = conv$
 <proof>

lemma *comp-converter-id-right*: $comp_converter\ conv\ id_converter = conv$
 <proof>

lemma *comp-converter-of-callee*: $comp_converter (converter_of_callee\ callee1\ s1) (converter_of_callee$
 $callee2\ s2)$

= *converter-of-callee* ($\lambda(s1, s2) q. \text{map-gpv } r\text{prodl } id \text{ (inline callee2 (callee1 s1 } q) s2)) (s1, s2)$)
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

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'*)
 ⟨*proof*⟩

lemma *parallel-converter2-comp1-out*:

parallel-converter2 (*comp-converter* *conv* *conv'*) *conv''* = *comp-converter* (*parallel-converter2*
conv *id-converter*) (*parallel-converter2* *conv'* *conv''*)
 ⟨*proof*⟩

lemma *parallel-converter2-comp2-out*:

parallel-converter2 *conv''* (*comp-converter* *conv* *conv'*) = *comp-converter* (*parallel-converter2*
id-converter *conv*) (*parallel-converter2* *conv''* *conv'*)
 ⟨*proof*⟩

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

lemma *interaction-any-bounded-converter-mono*:

assumes *interaction-any-bounded-converter* *conv* *n*
and $n \leq m$
shows *interaction-any-bounded-converter* *conv* *m*
 ⟨*proof*⟩

lemma *interaction-any-bounded-converter-trivial* [*simp*]: *interaction-any-bounded-converter conv* ∞
 ⟨*proof*⟩

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 (solves (clarsimp *simp add: simp*))) | *rule add interaction-bound*)
method *interaction-bound-converter-rec* **uses** *add simp* =
 (*interaction-bound-converter-step add: add simp: simp; (interaction-bound-converter-rec*
add: add simp: simp)?)
method *interaction-bound-converter* **uses** *add simp* =
 (*interaction-bound-converter-start, interaction-bound-converter-rec add: add simp:*
simp)

lemma *interaction-any-bounded-converter-id* [*interaction-bound*]:
interaction-any-bounded-converter id-converter 1
 ⟨*proof*⟩

lemma *raw-converter-invariant-interaction-any-bounded-converter*:
raw-converter-invariant \mathcal{I} -full \mathcal{I} -full run-converter (λ *conv. interaction-any-bounded-converter*
conv n)
 ⟨*proof*⟩

lemma *interaction-bounded-by-left-gpv* [*interaction-bound*]:
assumes *interaction-bounded-by consider gpv n*
and $\bigwedge x. \text{consider}' (Inl\ x) \implies \text{consider } x$
shows *interaction-bounded-by consider' (left-gpv gpv) n*
 ⟨*proof*⟩

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*⟩

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*
 ⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *interaction-any-bounded-converter-of-callee* [*interaction-bound*]:
assumes $\bigwedge s x. \textit{interaction-any-bounded-by} (\textit{conv s x}) n$
shows *interaction-any-bounded-converter* (*converter-of-callee conv s*) *n*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *interaction-any-bounded-converter-converter-of-resource*:
interaction-any-bounded-converter (*converter-of-resource res*) *n*
⟨*proof*⟩

lemma *interaction-any-bounded-converter-converter-of-resource'* [*interaction-bound*]:
interaction-any-bounded-converter (*converter-of-resource res*) *0*
⟨*proof*⟩

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*
⟨*proof*⟩

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
 (- \vdash_R / - \sim / - [100, 99, 99] 99)

for *A* **where**

eq-resource-onI: $A \vdash_R \text{res} \sim \text{res}'$ **if**

$\bigwedge a. a \in A \Longrightarrow \text{rel-spmf} (\text{rel-prod} (=) (\text{eq-resource-on } A)) (\text{run-resource } \text{res } a)$
 ($\text{run-resource } \text{res}' a$)

lemma *eq-resource-on-coinduct* [*consumes 1, case-names eq-resource-on, coinduct*
pred: eq-resource-on]:

assumes $X \text{res } \text{res}'$

and $\bigwedge \text{res } \text{res}' a. \llbracket X \text{res } \text{res}'; a \in A \rrbracket$

$\Longrightarrow \text{rel-spmf} (\text{rel-prod} (=) (\lambda \text{res } \text{res}'. X \text{res } \text{res}' \vee A \vdash_R \text{res} \sim \text{res}'))$

($\text{run-resource } \text{res } a$) ($\text{run-resource } \text{res}' a$)

shows $A \vdash_R \text{res} \sim \text{res}'$

<proof>

lemma *eq-resource-onD*:

assumes $A \vdash_R \text{res} \sim \text{res}' a \in A$

shows $\text{rel-spmf} (\text{rel-prod} (=) (\text{eq-resource-on } A)) (\text{run-resource } \text{res } a) (\text{run-resource } \text{res}' a)$

<proof>

lemma *eq-resource-on-refl* [*simp*]: $A \vdash_R \text{res} \sim \text{res}$

<proof>

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

<proof>

lemma *eq-resource-on-sym*: $A \vdash_R \text{res} \sim \text{res}'$ **if** $A \vdash_R \text{res}' \sim \text{res}$

<proof>

lemma *eq-resource-on-trans* [*trans*]: $A \vdash_R \text{res} \sim \text{res}''$ **if** $A \vdash_R \text{res} \sim \text{res}' A \vdash_R \text{res}' \sim \text{res}''$

<proof>

lemma *eq-resource-on-UNIV-D* [*simp*]: $\text{res} = \text{res}'$ **if** $\text{UNIV} \vdash_R \text{res} \sim \text{res}'$

<proof>

lemma *eq-resource-on-UNIV-iff*: $\text{UNIV} \vdash_R \text{res} \sim \text{res}' \longleftrightarrow \text{res} = \text{res}'$

<proof>

lemma *eq-resource-on-mono*: $\llbracket A' \vdash_R \text{res} \sim \text{res}'; A \subseteq A' \rrbracket \Longrightarrow A \vdash_R \text{res} \sim \text{res}'$

<proof>

lemma *eq-resource-on-empty* [*simp*]: $\{\}$ \vdash_R $res \sim res'$
 ⟨*proof*⟩

lemma *eq-resource-on-resource-of-oracleI*:

includes *lifting-syntax*

fixes S

assumes $sim: (S \implies eq\text{-on } A \implies rel\text{-spmf } (rel\text{-prod } (=) S)) r1 r2$

and $S: S s1 s2$

shows $A \vdash_R resource\text{-of-oracle } r1 s1 \sim resource\text{-of-oracle } r2 s2$

⟨*proof*⟩

lemma *exec-gpv-eq-resource-on*:

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

and $\mathcal{I} \vdash_g gpv \ \checkmark$

and $\mathcal{I} \vdash_{res} res \ \checkmark$

shows $rel\text{-spmf } (rel\text{-prod } (=) (eq\text{-resource-on } (outs\text{-}\mathcal{I} \ \mathcal{I}))) (exec\text{-gpv } run\text{-resource } gpv \ res) (exec\text{-gpv } run\text{-resource } gpv \ res')$

⟨*proof*⟩

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

for $A \ \mathcal{I} \ D$ **where**

$Pure: eq\text{-}\mathcal{I}\text{-generat } A \ \mathcal{I} \ D \ (Pure \ x) \ (Pure \ y) \ \mathbf{if} \ A \ x \ y$

$| IO: eq\text{-}\mathcal{I}\text{-generat } A \ \mathcal{I} \ D \ (IO \ out \ c) \ (IO \ out' \ c') \ \mathbf{if} \ out \in outs\text{-}\mathcal{I} \ \mathcal{I} \ \wedge input. \ input \in responses\text{-}\mathcal{I} \ \mathcal{I} \ out \implies D \ (c \ input) \ (c' \ input)$

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

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

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

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

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

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

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

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

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

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

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

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

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

$\llbracket eq\text{-}\mathcal{I}\text{-generat } A \ \mathcal{I} \ D \ x \ y; \bigwedge x \ y. A \ x \ y \implies A' \ x \ y; \bigwedge x \ y. D \ x \ y \implies D' \ x \ y; \mathcal{I} \leq \mathcal{I}' \rrbracket$

$\implies eq\text{-}\mathcal{I}\text{-generat } A' \ \mathcal{I}' \ D' \ x \ y$

⟨*proof*⟩

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

$A' D \leq D' \mathcal{I} \leq \mathcal{I}'$
 ⟨proof⟩

lemma *eq- \mathcal{I} -generat-mono''* [*mono*]:
 $\llbracket \bigwedge x y. A x y \longrightarrow A' x y; \bigwedge x y. D x y \longrightarrow D' x y \rrbracket$
 $\implies \text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D x y \longrightarrow \text{eq-}\mathcal{I}\text{-generat } A' \mathcal{I}' D' x y$
 ⟨proof⟩

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

lemma *eq- \mathcal{I} -generat-reflI*:
assumes $\bigwedge x. x \in \text{generat-pures generat} \implies A x x$
and $\bigwedge \text{out } c. \text{generat} = \text{IO out } c \implies \text{out} \in \text{outs-}\mathcal{I} \mathcal{I} \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out. } D (c \text{ input}) (c \text{ input}))$
shows $\text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D \text{ generat generat}$
 ⟨proof⟩

lemma *eq- \mathcal{I} -generat-relcompp*:
 $\text{eq-}\mathcal{I}\text{-generat } A \mathcal{I} D \text{ OO eq-}\mathcal{I}\text{-generat } A' \mathcal{I}' D' = \text{eq-}\mathcal{I}\text{-generat } (A \text{ OO } A') \mathcal{I} (D \text{ OO } D')$
 ⟨proof⟩

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

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

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

lemma *eq- \mathcal{I} -generat-into-rel-generat*:
 $\text{eq-}\mathcal{I}\text{-generat } A \mathcal{I}\text{-full } D \text{ generat generat}' \implies \text{rel-generat } A (=) (\text{rel-fun } (=) D) \text{ generat generat}'$
 ⟨proof⟩

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

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

assumes $X \text{ gpv } gpv'$
and $\bigwedge gpv \text{ gpv}'. X \text{ gpv } gpv'$
 $\implies \text{rel-spmf } (eq\text{-I-generat } A \mathcal{I} (\lambda gpv \text{ gpv}'. X \text{ gpv } gpv' \vee eq\text{-I-gpv } A \mathcal{I} \text{ gpv } gpv')) (the\text{-gpv } gpv) (the\text{-gpv } gpv')$
shows $eq\text{-I-gpv } A \mathcal{I} \text{ gpv } gpv'$
 $\langle proof \rangle$

lemma *eq-I-gpvD*:

$eq\text{-I-gpv } A \mathcal{I} \text{ gpv } gpv' \implies \text{rel-spmf } (eq\text{-I-generat } A \mathcal{I} (eq\text{-I-gpv } A \mathcal{I})) (the\text{-gpv } gpv) (the\text{-gpv } gpv')$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *eq-I-gpv-Pause*:

$\llbracket out \in outs\text{-I } \mathcal{I}; \bigwedge input. input \in responses\text{-I } \mathcal{I} \ out \implies eq\text{-I-gpv } A \mathcal{I} (rpv \ input) (rpv' \ input) \rrbracket$
 $\implies eq\text{-I-gpv } A \mathcal{I} (Pause \ out \ rpv) (Pause \ out \ rpv')$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

$\llbracket eq\text{-I-gpv } A \mathcal{I} \text{ gpv } gpv'; \bigwedge x \ y. A \ x \ y \implies A' \ x \ y; \mathcal{I} \leq \mathcal{I}' \rrbracket \implies eq\text{-I-gpv } A' \mathcal{I}' \text{ gpv } gpv'$
 $\langle proof \rangle$

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

$eq\text{-I-gpv } A \mathcal{I} \text{ gpv } gpv' \longrightarrow eq\text{-I-gpv } A' \mathcal{I}' \text{ gpv } gpv'$ **if** $\bigwedge x \ y. A \ x \ y \longrightarrow A' \ x \ y$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *eq-I-gpv-reflI*:

$\llbracket \bigwedge x. x \in results\text{-gpv } \mathcal{I} \text{ gpv} \implies A \ x \ x; \mathcal{I} \vdash g \text{ gpv } \checkmark \rrbracket \implies eq\text{-I-gpv } A \mathcal{I} \text{ gpv } gpv$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

\mathcal{I} (is ?lhs = ?rhs)
 ⟨proof⟩

lemma *eq- \mathcal{I} -gpv-map-gpv1*: $eq\text{-}\mathcal{I}\text{-}gpv\ A\ \mathcal{I}\ (map\text{-}gpv\ f\ id\ gpv)\ gpv' \longleftrightarrow eq\text{-}\mathcal{I}\text{-}gpv\ (\lambda x. A\ (f\ x))\ \mathcal{I}\ gpv\ gpv'$ (is ?lhs \longleftrightarrow ?rhs)
 ⟨proof⟩

lemma *eq- \mathcal{I} -gpv-map-gpv2*: $eq\text{-}\mathcal{I}\text{-}gpv\ A\ \mathcal{I}\ gpv\ (map\text{-}gpv\ f\ id\ gpv') = eq\text{-}\mathcal{I}\text{-}gpv\ (\lambda x\ y. A\ x\ (f\ y))\ \mathcal{I}\ gpv\ gpv'$
 ⟨proof⟩

lemmas *eq- \mathcal{I} -gpv-map-gpv [simp]* = *eq- \mathcal{I} -gpv-map-gpv1[abs-def]* *eq- \mathcal{I} -gpv-map-gpv2*

lemma (in *callee-invariant-on*) *eq- \mathcal{I} -exec-gpv*:
 $\llbracket eq\text{-}\mathcal{I}\text{-}gpv\ A\ \mathcal{I}\ gpv\ gpv';\ I\ s \rrbracket \implies rel\text{-}spmf\ (rel\text{-}prod\ A\ (eq\text{-}onp\ I))\ (exec\text{-}gpv\ callee\ gpv\ s)\ (exec\text{-}gpv\ callee\ gpv'\ s)$
 ⟨proof⟩

lemma *eq- \mathcal{I} -gpv-coinduct-bind [consumes 1, case-names eq- \mathcal{I} -gpv]*:
fixes $gpv :: ('a, 'out, 'in)\ gpv$ **and** $gpv' :: ('a', 'out', 'in)\ gpv$
assumes $X: X\ gpv\ gpv'$
and step: $\bigwedge gpv\ gpv'. X\ gpv\ gpv' \implies rel\text{-}spmf\ (eq\text{-}\mathcal{I}\text{-}generat\ A\ \mathcal{I}\ (\lambda gpv\ gpv'. X\ gpv\ gpv' \vee eq\text{-}\mathcal{I}\text{-}gpv\ A\ \mathcal{I}\ gpv\ gpv' \vee (\exists gpv''\ gpv''' (B :: 'b \Rightarrow 'b' \Rightarrow bool)\ f\ g. gpv = bind\text{-}gpv\ gpv''\ f \wedge gpv' = bind\text{-}gpv\ gpv'''\ g \wedge eq\text{-}\mathcal{I}\text{-}gpv\ B\ \mathcal{I}\ gpv''\ gpv''' \wedge (rel\text{-}fun\ B\ X)\ f\ g)))\ (the\text{-}gpv\ gpv)\ (the\text{-}gpv\ gpv')$
shows $eq\text{-}\mathcal{I}\text{-}gpv\ A\ \mathcal{I}\ gpv\ gpv'$
 ⟨proof⟩

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

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

(inline1 callee1 gpv1 s1) (inline1 callee2 gpv2 s2)
 ⟨proof⟩

lemma eq- \mathcal{I} -gpv-inline:

assumes $S: S\ s1\ s2$

and $gpv: eq\text{-}\mathcal{I}\text{-gpv}\ A\ \mathcal{I}\ gpv1\ gpv2$

shows $eq\text{-}\mathcal{I}\text{-gpv}\ (rel\text{-}prod\ A\ S)\ \mathcal{I}'\ (inline\ callee1\ gpv1\ s1)\ (inline\ callee2\ gpv2\ s2)$

⟨proof⟩

end

lemma eq- \mathcal{I} -gpv-left-gpv-cong:

$eq\text{-}\mathcal{I}\text{-gpv}\ A\ \mathcal{I}\ gpv\ gpv' \implies eq\text{-}\mathcal{I}\text{-gpv}\ A\ (\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}')\ (left\text{-}gpv\ gpv)\ (left\text{-}gpv\ gpv')$

⟨proof⟩

lemma eq- \mathcal{I} -gpv-right-gpv-cong:

$eq\text{-}\mathcal{I}\text{-gpv}\ A\ \mathcal{I}'\ gpv\ gpv' \implies eq\text{-}\mathcal{I}\text{-gpv}\ A\ (\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}')\ (right\text{-}gpv\ gpv)\ (right\text{-}gpv\ gpv')$

⟨proof⟩

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

⟨proof⟩

lemma eq- \mathcal{I} -gpvD-results-gpv2:

assumes $eq\text{-}\mathcal{I}\text{-gpv}\ A\ \mathcal{I}\ gpv\ gpv'\ y \in results\text{-}gpv\ \mathcal{I}\ gpv'$

shows $\exists x \in results\text{-}gpv\ \mathcal{I}\ gpv. A\ x\ y$

⟨proof⟩

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

converter $\Rightarrow ('a, 'b, 'out, 'in)\ converter \Rightarrow bool$

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

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

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

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

$(eq\text{-}\mathcal{I}\text{-converter}\ \mathcal{I}\ \mathcal{I}')\ \mathcal{I}'\ (run\text{-}converter\ conv\ q)\ (run\text{-}converter\ conv'\ q)$

lemma eq- \mathcal{I} -converter-coinduct [consumes 1, case-names eq- \mathcal{I} -converter, coinduct pred: eq- \mathcal{I} -converter]:

assumes $X\ conv\ conv'$

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

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

$conv' \vee \mathcal{I}, \mathcal{I}' \vdash_C\ conv \sim conv')$ \mathcal{I}'

$(run\text{-}converter\ conv\ q)\ (run\text{-}converter\ conv'\ q)$

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

⟨proof⟩

lemma eq- \mathcal{I} -converterD:

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

$(run\text{-}converter\ conv\ q)\ (run\text{-}converter\ conv'\ q)$

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

lemma *eq- \mathcal{I} -converter-refl*: $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}$ if $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark$
 ⟨proof⟩

lemma *eq- \mathcal{I} -converter-sym* [*sym*]: $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}'$ if $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \sim \text{conv}$
 ⟨proof⟩

lemma *eq- \mathcal{I} -converter-trans* [*trans*]:
 $\llbracket \mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}'; \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}' \sim \text{conv}'' \rrbracket \implies \mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}''$
 ⟨proof⟩

lemma *eq- \mathcal{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 $\mathcal{I}1', \mathcal{I}2' \vdash_C \text{conv} \sim \text{conv}'$
 ⟨proof⟩

lemma *eq- \mathcal{I} -converter-eq*: $\text{conv}1 = \text{conv}2$ if \mathcal{I} -full, \mathcal{I} -full $\vdash_C \text{conv}1 \sim \text{conv}2$
 ⟨proof⟩

lemma *eq- \mathcal{I} -attach-on*:
 assumes $\mathcal{I}' \vdash_{\text{res}} \text{res} \checkmark$ \mathcal{I} -uniform $A \text{ UNIV}$, $\mathcal{I}' \vdash_C \text{conv} \sim \text{conv}'$
 shows $A \vdash_R \text{attach conv res} \sim \text{attach conv}' \text{ res}$
 ⟨proof⟩

lemma *eq- \mathcal{I} -attach-on'*:
 assumes $\mathcal{I}' \vdash_{\text{res}} \text{res} \checkmark$ $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \sim \text{conv}'$ $A \subseteq \text{outs-}\mathcal{I} \mathcal{I}$
 shows $A \vdash_R \text{attach conv res} \sim \text{attach conv}' \text{ res}$
 ⟨proof⟩

lemma *eq- \mathcal{I} -attach*:
 $\llbracket \mathcal{I}' \vdash_{\text{res}} \text{res} \checkmark; \mathcal{I}$ -full, $\mathcal{I}' \vdash_C \text{conv} \sim \text{conv}' \rrbracket \implies \text{attach conv res} = \text{attach conv}' \text{ res}$
 ⟨proof⟩

lemma *eq- \mathcal{I} -comp-cong*:
 $\llbracket \mathcal{I}1, \mathcal{I}2 \vdash_C \text{conv}1 \sim \text{conv}1'; \mathcal{I}2, \mathcal{I}3 \vdash_C \text{conv}2 \sim \text{conv}2' \rrbracket$
 $\implies \mathcal{I}1, \mathcal{I}3 \vdash_C \text{comp-converter conv}1 \text{ conv}2 \sim \text{comp-converter conv}1' \text{ conv}2'$
 ⟨proof⟩

lemma *comp-converter-cong*: $\text{comp-converter conv}1 \text{ conv}2 = \text{comp-converter conv}1' \text{ conv}2'$
 if \mathcal{I} -full, $\mathcal{I} \vdash_C \text{conv}1 \sim \text{conv}1'$ \mathcal{I}, \mathcal{I} -full $\vdash_C \text{conv}2 \sim \text{conv}2'$
 ⟨proof⟩

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

<proof>

lemma *parallel-converter2-eq-I-cong*:

$\llbracket \mathcal{I}1, \mathcal{I}1' \vdash_C \text{conv}1 \sim \text{conv}1'; \mathcal{I}2, \mathcal{I}2' \vdash_C \text{conv}2 \sim \text{conv}2' \rrbracket$
 $\implies \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C \text{parallel-converter2 } \text{conv}1 \text{ conv}2 \sim \text{parallel-converter2}$
 $\text{conv}1' \text{ conv}2'$
<proof>

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

<proof>

lemma *eq-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 $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}2 \checkmark$
<proof>

lemma *eq-I-converterD-WT*:

assumes $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}1 \sim \text{conv}2$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv}1 \checkmark \longleftrightarrow \mathcal{I}, \mathcal{I}' \vdash_C \text{conv}2 \checkmark$
<proof>

lemma *eq-I-gpv-Fail [simp]*: $\text{eq-I-gpv } A \ \mathcal{I} \ \text{Fail} \ \text{Fail}$

<proof>

lemma *eq-I-restrict-gpv*:

assumes $\text{eq-I-gpv } A \ \mathcal{I} \ \text{gpv} \ \text{gpv}'$
shows $\text{eq-I-gpv } A \ \mathcal{I} \ (\text{restrict-gpv } \mathcal{I} \ \text{gpv}) \ \text{gpv}'$
<proof>

lemma *eq-I-restrict-converter*:

assumes $\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \checkmark$
and $\text{outs-}\mathcal{I} \ \mathcal{I} \subseteq A$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \text{restrict-converter } A \ \mathcal{I}' \ \text{cnv} \sim \text{cnv}$
<proof>

lemma *eq-I-restrict-gpv-full*:

$\text{eq-I-gpv } A \ \mathcal{I} \text{-full} \ (\text{restrict-gpv } \mathcal{I} \ \text{gpv}) \ (\text{restrict-gpv } \mathcal{I} \ \text{gpv}')$
if $\text{eq-I-gpv } A \ \mathcal{I} \ \text{gpv} \ \text{gpv}'$
<proof>

lemma *eq-I-restrict-converter-cong*:

assumes $\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \sim \text{cnv}'$
and $A \subseteq \text{outs-}\mathcal{I} \ \mathcal{I}$
shows $\text{restrict-converter } A \ \mathcal{I}' \ \text{cnv} = \text{restrict-converter } A \ \mathcal{I}' \ \text{cnv}'$
<proof>

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 [] $x = \text{bind-spmf } p (\lambda s. \text{map-spmf } \text{fst } (\text{callee } s \ x))$
| *trace-callee callee p* ((a, b) # xs) $x =$
trace-callee callee (cond-spmf-fst (bind-spmf p (\lambda s. callee s a)) b) xs x

definition *trace-callee-eq* :: ('a, 'b, 's1) callee \Rightarrow ('a, 'b, 's2) callee \Rightarrow 'a set \Rightarrow 's1
spmf \Rightarrow 's2 spmf \Rightarrow bool **where**

trace-callee-eq callee1 callee2 A p q \longleftrightarrow
($\forall xs. \text{set } xs \subseteq A \times \text{UNIV} \longrightarrow (\forall x \in A. \text{trace-callee } \text{callee1 } p \ x \ x = \text{trace-callee}$
callee2 q xs x))

abbreviation *trace-callee-eq'* :: 'a set \Rightarrow ('a, 'b, 's1) callee \Rightarrow 's1 \Rightarrow ('a, 'b, 's2)
callee \Rightarrow 's2 \Rightarrow bool

($- \vdash_C / (-'((-)')$) $\approx / (-'((-)')$) [90, 0, 0, 0, 0] 91)
where *trace-callee-eq' A callee1 s1 callee2 s2* $\equiv \text{trace-callee-eq } \text{callee1 } \text{callee2 } A$
(*return-spmf s1*) (*return-spmf s2*)

lemma *trace-callee-eqI*:

assumes $\bigwedge xs \ x. \llbracket \text{set } xs \subseteq A \times \text{UNIV}; x \in A \rrbracket$
 $\implies \text{trace-callee } \text{callee1 } p \ x \ x = \text{trace-callee } \text{callee2 } q \ x \ x$
shows *trace-callee-eq callee1 callee2 A p q*
(*proof*)

lemma *trace-callee-eqD*:

assumes *trace-callee-eq callee1 callee2 A p q*
and $\text{set } xs \subseteq A \times \text{UNIV } x \in A$
shows $\text{trace-callee } \text{callee1 } p \ x \ x = \text{trace-callee } \text{callee2 } q \ x \ x$
(*proof*)

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

(*proof*)

lemma *trace-callee-None [simp]*:

$\text{trace-callee } \text{callee } (\text{return-pmf } \text{None}) \ xs \ x = \text{return-pmf } \text{None}$
(*proof*)

proposition *trace'-eqI-sim*:

fixes *callee1* :: ('a, 'b, 's1) callee **and** *callee2* :: ('a, 'b, 's2) callee
assumes *start*: $S \ p \ q$
and *step*: $\bigwedge p \ q \ a. \llbracket S \ p \ q; a \in A \rrbracket \implies$
 $\text{bind-spmf } p (\lambda s. \text{map-spmf } \text{fst } (\text{callee1 } s \ a)) = \text{bind-spmf } q (\lambda s. \text{map-spmf } \text{fst}$
(*callee2 s a*)
and *sim*: $\bigwedge p \ q \ a \ \text{res } b \ s'. \llbracket S \ p \ q; a \in A; \text{res} \in \text{set-spmf } q; (b, s') \in \text{set-spmf}$

$(\text{callee2 } \text{res } a) \]$
 $\implies S (\text{cond-spmf-fst } (\text{bind-spmf } p (\lambda s. \text{callee1 } s \ a)) \ b)$
 $(\text{cond-spmf-fst } (\text{bind-spmf } q (\lambda s. \text{callee2 } s \ a)) \ b)$
shows $\text{trace-callee-eq } \text{callee1 } \text{callee2 } A \ p \ q$
 $\langle \text{proof} \rangle$

fun $\text{trace-callee-aux} :: ('a, 'b, 's) \text{callee} \Rightarrow 's \text{spmf} \Rightarrow ('a \times 'b) \text{list} \Rightarrow 's \text{spmf}$
where
 $\text{trace-callee-aux } \text{callee } p \ [] = p$
 $| \text{trace-callee-aux } \text{callee } p \ ((x, y) \# \text{xs}) = \text{trace-callee-aux } \text{callee} (\text{cond-spmf-fst}$
 $(\text{bind-spmf } p (\lambda \text{res}. \text{callee } \text{res } x)) \ y) \ \text{xs}$

lemma $\text{trace-callee-conv-trace-callee-aux}$:
 $\text{trace-callee } \text{callee } p \ \text{xs } a = \text{bind-spmf} (\text{trace-callee-aux } \text{callee } p \ \text{xs}) (\lambda s. \text{map-spmf}$
 $\text{fst } (\text{callee } s \ a))$
 $\langle \text{proof} \rangle$

lemma $\text{trace-callee-aux-append}$:
 $\text{trace-callee-aux } \text{callee } p \ (\text{xs } @ \ \text{ys}) = \text{trace-callee-aux } \text{callee} (\text{trace-callee-aux } \text{callee}$
 $p \ \text{xs}) \ \text{ys}$
 $\langle \text{proof} \rangle$

inductive $\text{trace-callee-closure} :: ('a, 'b, 's1) \text{callee} \Rightarrow ('a, 'b, 's2) \text{callee} \Rightarrow 'a \text{set}$
 $\Rightarrow 's1 \text{spmf} \Rightarrow 's2 \text{spmf} \Rightarrow 's1 \text{spmf} \Rightarrow 's2 \text{spmf} \Rightarrow \text{bool}$
for $\text{callee1 } \text{callee2 } A \ p \ q$ **where**
 $\text{trace-callee-closure } \text{callee1 } \text{callee2 } A \ p \ q (\text{trace-callee-aux } \text{callee1 } p \ \text{xs}) (\text{trace-callee-aux}$
 $\text{callee2 } q \ \text{xs}) \ \text{if } \text{set } \text{xs} \subseteq A \times \text{UNIV}$

lemma $\text{trace-callee-closure-start}$: $\text{trace-callee-closure } \text{callee1 } \text{callee2 } A \ p \ q \ p \ q$
 $\langle \text{proof} \rangle$

lemma $\text{trace-callee-closure-step}$:
assumes $\text{trace-callee-eq } \text{callee1 } \text{callee2 } A \ p \ q$
and $\text{trace-callee-closure } \text{callee1 } \text{callee2 } A \ p \ q \ p' \ q'$
and $a \in A$
shows $\text{bind-spmf } p' (\lambda s. \text{map-spmf } \text{fst} (\text{callee1 } s \ a)) = \text{bind-spmf } q' (\lambda s. \text{map-spmf}$
 $\text{fst} (\text{callee2 } s \ a))$
 $\langle \text{proof} \rangle$

lemma $\text{trace-callee-closure-sim}$:
assumes $\text{trace-callee-closure } \text{callee1 } \text{callee2 } A \ p \ q \ p' \ q'$
and $a \in A$
shows $\text{trace-callee-closure } \text{callee1 } \text{callee2 } A \ p \ q$
 $(\text{cond-spmf-fst } (\text{bind-spmf } p' (\lambda s. \text{callee1 } s \ a)) \ b)$
 $(\text{cond-spmf-fst } (\text{bind-spmf } q' (\lambda s. \text{callee2 } s \ a)) \ b)$
 $\langle \text{proof} \rangle$

proposition $\text{trace-callee-eq-complete}$:
assumes $\text{trace-callee-eq } \text{callee1 } \text{callee2 } A \ p \ q$

obtains S
where $S p q$
and $\bigwedge p q a. \llbracket S p q; a \in A \rrbracket \implies$
 $bind\text{-}spmf\ p\ (\lambda s. map\text{-}spmf\ fst\ (callee1\ s\ a)) = bind\text{-}spmf\ q\ (\lambda s. map\text{-}spmf\ fst$
 $(callee2\ s\ a))$
and $\bigwedge p q a s b s'. \llbracket S p q; a \in A; s \in set\text{-}spmf\ q; (b, s') \in set\text{-}spmf\ (callee2\ s$
 $a) \rrbracket$
 $\implies S\ (cond\text{-}spmf\text{-}fst\ (bind\text{-}spmf\ p\ (\lambda s. callee1\ s\ a))\ b)$
 $(cond\text{-}spmf\text{-}fst\ (bind\text{-}spmf\ q\ (\lambda s. callee2\ s\ a))\ b)$
 $\langle proof \rangle$

lemma $set\text{-}spmf\text{-}cond\text{-}spmf\text{-}fst$: $set\text{-}spmf\ (cond\text{-}spmf\text{-}fst\ p\ a) = snd\ ' (set\text{-}spmf\ p$
 $\cap \{a\} \times UNIV)$
 $\langle proof \rangle$

lemma $trace\text{-}callee\text{-}eq\text{-}run\text{-}gpv$:
fixes $callee1 :: ('a, 'b, 's1)\ callee$ **and** $callee2 :: ('a, 'b, 's2)\ callee$
assumes $trace\text{-}eq$: $trace\text{-}callee\text{-}eq\ callee1\ callee2\ A\ p\ q$
and $inv1$: $callee\text{-}invariant\text{-}on\ callee1\ I1\ \mathcal{I}$
and $inv2$: $callee\text{-}invariant\text{-}on\ callee2\ I2\ \mathcal{I}$
and WT : $\mathcal{I} \vdash_g\ gpv\ \checkmark$
and out : $outs\text{-}gpv\ \mathcal{I}\ gpv \subseteq A$
and pq : $lossless\text{-}spmf\ p\ lossless\text{-}spmf\ q$
and $I1$: $\forall x \in set\text{-}spmf\ p. I1\ x$
and $I2$: $\forall y \in set\text{-}spmf\ q. I2\ y$
shows $bind\text{-}spmf\ p\ (run\text{-}gpv\ callee1\ gpv) = bind\text{-}spmf\ q\ (run\text{-}gpv\ callee2\ gpv)$
 $\langle proof \rangle$

lemma $trace\text{-}callee\text{-}eq'\text{-}run\text{-}gpv$:
fixes $callee1 :: ('a, 'b, 's1)\ callee$ **and** $callee2 :: ('a, 'b, 's2)\ callee$
assumes $trace\text{-}eq$: $A \vdash_C\ callee1(s1) \approx callee2(s2)$
and $inv1$: $callee\text{-}invariant\text{-}on\ callee1\ I1\ \mathcal{I}$
and $inv2$: $callee\text{-}invariant\text{-}on\ callee2\ I2\ \mathcal{I}$
and WT : $\mathcal{I} \vdash_g\ gpv\ \checkmark$
and out : $outs\text{-}gpv\ \mathcal{I}\ gpv \subseteq A$
and $I1$: $I1\ s1$
and $I2$: $I2\ s2$
shows $run\text{-}gpv\ callee1\ gpv\ s1 = run\text{-}gpv\ callee2\ gpv\ s2$
 $\langle proof \rangle$

abbreviation $trace\text{-}eq :: 'a\ set \Rightarrow ('a, 'b)\ resource\ spmf \Rightarrow ('a, 'b)\ resource\ spmf$
 $\Rightarrow bool$ **where**
 $trace\text{-}eq \equiv trace\text{-}callee\text{-}eq\ run\text{-}resource\ run\text{-}resource$

abbreviation $trace\text{-}eq' :: 'a\ set \Rightarrow ('a, 'b)\ resource \Rightarrow ('a, 'b)\ resource \Rightarrow bool\ ((-)$
 $\vdash_R / (-) / \approx (-) [90, 90, 90] 91)$ **where**
 $A \vdash_R\ res \approx res' \equiv trace\text{-}eq\ A\ (return\text{-}spmf\ res)\ (return\text{-}spmf\ res')$

lemma $trace\text{-}callee\text{-}resource\text{-}of\text{-}oracle2$:

trace-callee run-resource (*map-spmf* (*resource-of-oracle callee*) *p*) *xs x* =
trace-callee callee p xs x
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *trace-eq'-resource-of-oracle* [*simp*]:
 $A \vdash_R \text{resource-of-oracle callee1 } s1 \approx \text{resource-of-oracle callee2 } s2 =$
 $A \vdash_C \text{callee1}(s1) \approx \text{callee2}(s2)$
 ⟨*proof*⟩

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*⟩

lemma *absorb-comp-converter*: *absorb d (comp-converter conv conv')* = *absorb (absorb d conv) conv'*
 ⟨*proof*⟩

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*: *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 *connect d res1* = *connect d res2*

⟨*proof*⟩

lemma *distinguish-trace-eq*:
assumes *distinguish*: $\bigwedge \text{distinguisher}. \mathcal{I} \vdash_g \text{distinguisher} \checkmark \implies \text{connect distinguisher res} = \text{connect distinguisher res}'$
and *WT1*: $\mathcal{I} \vdash_{\text{res}} \text{res1} \checkmark$
and *WT2*: $\mathcal{I} \vdash_{\text{res}} \text{res2} \checkmark$
shows *outs- \mathcal{I}* $\mathcal{I} \vdash_R \text{res} \approx \text{res}'$
<proof>

lemma *connect-eq-resource-cong*:
assumes $\mathcal{I} \vdash_g \text{distinguisher} \checkmark$
and *outs- \mathcal{I}* $\mathcal{I} \vdash_R \text{res} \sim \text{res}'$
and $\mathcal{I} \vdash_{\text{res}} \text{res} \checkmark$
shows $\text{connect distinguisher res} = \text{connect distinguisher res}'$
<proof>

lemma *WT-gpv-absorb [WT-intro]*:
 $\llbracket \mathcal{I}' \vdash_g \text{gpv} \checkmark; \mathcal{I}', \mathcal{I} \vdash_C \text{conv} \checkmark \rrbracket \implies \mathcal{I} \vdash_g \text{absorb gpv conv} \checkmark$
<proof>

lemma *plossless-gpv-absorb [plossless-intro]*:
assumes *gpv*: *plossless-gpv* $\mathcal{I}' \text{ gpv}$
and *conv*: *plossless-converter* $\mathcal{I}' \mathcal{I} \text{ conv}$
and [*WT-intro*]: $\mathcal{I}' \vdash_g \text{gpv} \checkmark \mathcal{I}', \mathcal{I} \vdash_C \text{conv} \checkmark$
shows *plossless-gpv* $\mathcal{I} (\text{absorb gpv conv})$
<proof>

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* $(\text{absorb gpv conv}) (\text{bound1} * \text{bound2})$
<proof>

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** \parallel 50)

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

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

notation *Converter* (§C§)

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

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

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

notation *id-oracle* (1_I)

notation *plus-oracle* (**infixr** \oplus_O 504)

notation *parallel-oracle* (**infixr** \ddagger_O 504)

notation *plus-intercept* (**infixr** \oplus_I 504)

notation *parallel-intercept* (**infixr** \ddagger_I 504)

notation *parallel-resource* (**infixr** \parallel 501)

notation *parallel-converter* (**infixr** $|_\infty$ 501)

notation *parallel-converter2* (**infixr** $|_=$ 501)

notation *comp-converter* (**infixr** \odot 502)

notation *fail-converter* (\perp_C)

notation *id-converter* (1_C)

notation *attach* (**infixr** \triangleright 500)

6.2 Wiring primitives

primrec *swap-sum* :: $'a + 'b \Rightarrow 'b + 'a$ **where**

swap-sum (*Inl* x) = *Inr* x

| *swap-sum* (*Inr* y) = *Inl* y

definition *swap_C* :: $('a + 'b, 'c + 'd, 'b + 'a, 'd + 'c)$ *converter* **where**

swap_C = *map-converter* *swap-sum* *swap-sum* *id* *id* 1_C

definition *rassocl_C* :: $('a + ('b + 'c), 'd + ('e + 'f), ('a + 'b) + 'c, ('d + 'e) + 'f)$ *converter* **where**

rassocl_C = *map-converter* *lsumr* *rsuml* *id* *id* 1_C

definition *lassocr_C* :: $((('a + 'b) + 'c, ('d + 'e) + 'f, 'a + ('b + 'c), 'd + ('e + 'f))$ *converter* **where**

lassocr_C = *map-converter* *rsuml* *lsumr* *id* *id* 1_C

definition *swap-rassocl* **where** *swap-rassocl* \equiv *lassocr_C* \odot (1_C $|_=$ *swap_C*) \odot *rassocl_C*

definition *swap-lassocr* **where** *swap-lassocr* \equiv *rassocl_C* \odot (*swap_C* $|_=$ 1_C) \odot *lassocr_C*

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

parallel-wiring = *lassocr_C* \odot (1_C $|_=$ *swap-lassocr*) \odot *rassocl_C*

lemma *WT-lassocr_C* [*WT-intro*]: $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3, \mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3) \vdash_C$
lassocr_C \checkmark
 ⟨*proof*⟩

lemma *WT-rassocl_C* [*WT-intro*]: $\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3), (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3 \vdash_C$
rassocl_C \checkmark
 ⟨*proof*⟩

lemma *WT-swap_C* [*WT-intro*]: $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}1 \vdash_C$ *swap_C* \checkmark
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemma *WT-swap-rassocl* [*WT-intro*]: $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3, (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}3) \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_C$
swap-rassocl \checkmark
 ⟨*proof*⟩

lemma *WT-parallel-wiring* [*WT-intro*]:
 $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} (\mathcal{I}3 \oplus_{\mathcal{I}} \mathcal{I}4), (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}3) \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}4) \vdash_C$ *parallel-wiring* \checkmark
 ⟨*proof*⟩

lemma *map-swap-sum-plus-oracle: includes lifting-syntax shows*
 $(id \dashrightarrow swap-sum \dashrightarrow map-spmf (map-prod swap-sum id)) (oracle1 \oplus_O$
oracle2) =
 $(oracle2 \oplus_O oracle1)$
 ⟨*proof*⟩

lemma *map- \mathcal{I} -rsuml-lsumr* [*simp*]: $map\text{-}\mathcal{I} \text{ rsuml lsumr } (\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)) =$
 $((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3)$
 ⟨*proof*⟩

lemma *map- \mathcal{I} -lsumr-rsuml* [*simp*]: $map\text{-}\mathcal{I} \text{ lsumr rsuml } ((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3) = (\mathcal{I}1$
 $\oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3))$
 ⟨*proof*⟩

lemma *map- \mathcal{I} -swap-sum* [*simp*]: $map\text{-}\mathcal{I} \text{ swap-sum swap-sum } (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) = \mathcal{I}2 \oplus_{\mathcal{I}}$
 $\mathcal{I}1$
 ⟨*proof*⟩

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
 ⟨*proof*⟩

lemma *plossless-rassocl_C* [*plossless-intro*]: *plossless-converter* ($\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)$)
 $((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3)$ *rassocl_C*
 ⟨*proof*⟩

lemma *plossless-lassocr_C* [*plossless-intro*]: *plossless-converter* $((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3)$
 $(\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3))$ *lassocr_C*
 ⟨*proof*⟩

lemma *plossless-swap_C* [*plossless-intro*]: *plossless-converter* $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2)$ $(\mathcal{I}2 \oplus_{\mathcal{I}}$
 $\mathcal{I}1)$ *swap_C*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *rsuml-lsumr-parallel-converter2*:
map-converter *id id rsuml lsumr* $((conv1 \mid= conv2) \mid= conv3) =$
map-converter rsuml lsumr id id $(conv1 \mid= conv2 \mid= conv3)$
 ⟨*proof*⟩

lemma *comp-lassocr_C*: $((conv1 \mid= conv2) \mid= conv3) \odot$ *lassocr_C* = *lassocr_C* \odot
 $(conv1 \mid= conv2 \mid= conv3)$
 ⟨*proof*⟩

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

lemma *lsumr-rsuml-parallel-converter2*:
map-converter id id lsumr rsuml $(conv1 \mid= (conv2 \mid= conv3)) =$
map-converter lsumr rsuml id id $((conv1 \mid= conv2) \mid= conv3)$
 ⟨*proof*⟩

lemma *comp-rassocl_C*:
 $(conv1 \mid= conv2 \mid= conv3) \odot$ *rassocl_C* = *rassocl_C* $\odot ((conv1 \mid= conv2) \mid= conv3)$
 ⟨*proof*⟩

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

lemma *swap-sum-right-gpv*:
map-gpv' id swap-sum swap-sum $(right-gpv\ gpv) = left-gpv\ gpv$
 ⟨*proof*⟩

lemma *swap-sum-left-gpv*:
map-gpv' id swap-sum swap-sum $(left-gpv\ gpv) = right-gpv\ gpv$
 ⟨*proof*⟩

lemma *swap-sum-parallel-converter2*:
map-converter id id swap-sum swap-sum $(conv1 \mid= conv2) =$

map-converter swap-sum swap-sum id id (conv2 |= conv1)
 ⟨proof⟩

lemma *comp-swap_C*: $(conv1 \mid= conv2) \odot swap_C = swap_C \odot (conv2 \mid= conv1)$
 ⟨proof⟩

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

lemma *comp-swap-lassocr*: $(conv1 \mid= conv2 \mid= conv3) \odot swap-lassocr = swap-lassocr$
 $\odot (conv2 \mid= conv1 \mid= conv3)$
 ⟨proof⟩

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

lemma *comp-parallel-wiring*:
 $((C1 \mid= C2) \mid= (C3 \mid= C4)) \odot parallel-wiring = parallel-wiring \odot ((C1 \mid= C3)$
 $\mid= (C2 \mid= C4))$
 ⟨proof⟩

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

lemma *attach-converter-of-resource-conv-parallel-resource*:
converter-of-resource res $\mid_{\infty} 1_C \triangleright res' = res \parallel res'$
 ⟨proof⟩

lemma *attach-converter-of-resource-conv-parallel-resource2*:
 $1_C \mid_{\infty} converter-of-resource res \triangleright res' = res' \parallel res$
 ⟨proof⟩

lemma *plossless-parallel-wiring* [*plossless-intro*]:
plossless-converter $((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} (\mathcal{I}3 \oplus_{\mathcal{I}} \mathcal{I}4)) ((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}3) \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}4))$
parallel-wiring
 ⟨proof⟩

lemma *run-converter-lassocr* [*simp*]:
run-converter lassocr_C x = *Pause (rsuml x) (λx. Done (lsumr x, lassocr_C))*
 ⟨proof⟩

lemma *run-converter-rassocl* [*simp*]:
run-converter rassocl_C x = *Pause (lsumr x) (λx. Done (rsuml x, rassocl_C))*
 ⟨proof⟩

lemma *run-converter-swap* [*simp*]: *run-converter swap_C x* = *Pause (swap-sum x)*
 $(\lambda x. Done (swap-sum x, swap_C))$
 ⟨proof⟩

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

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

run-converter swap-lassocr x = Pause (lassocr-swap-sum x) (
case lsumr x of Inl - => (λy. case lsumr y of Inl - => Done (lassocr-swap-sum
y, swap-lassocr) | - => Fail)
| Inr - => (λy. case lsumr y of Inl - => Fail | Inr - => Done (lassocr-swap-sum
y, swap-lassocr))))
⟨proof⟩

definition *parallel-sum-wiring* **where** *parallel-sum-wiring = lsumr ∘ map-sum id*
lassocr-swap-sum ∘ rsuml

lemma *run-converter-parallel-wiring*:

run-converter parallel-wiring x = Pause (parallel-sum-wiring x) (
case rsuml x of Inl - => (λy. case rsuml y of Inl - => Done (parallel-sum-wiring
y, parallel-wiring) | - => Fail)
| Inr x => (case lsumr x of Inl - => (λy. case rsuml y of Inl - => Fail
| Inr x => (case lsumr x of Inl - => Done (parallel-sum-wiring y, parallel-wiring) |
Inr - => Fail)))
| Inr - => (λy. case rsuml y of Inl - => Fail
| Inr x => (case lsumr x of Inl - => Fail | Inr - => Done (parallel-sum-wiring y,
parallel-wiring))))))
⟨proof⟩

lemma *bound-lassocr_C* [interaction-bound]: *interaction-any-bounded-converter las-*
socr_C 1
⟨proof⟩

lemma *bound-rassocl_C* [interaction-bound]: *interaction-any-bounded-converter ras-*
socl_C 1
⟨proof⟩

lemma *bound-swap_C* [interaction-bound]: *interaction-any-bounded-converter swap_C*
1
⟨proof⟩

lemma *bound-swap-rassocl* [interaction-bound]: *interaction-any-bounded-converter*
swap-rassocl 1
⟨proof⟩

lemma *bound-swap-lassocr* [interaction-bound]: *interaction-any-bounded-converter*
swap-lassocr 1
⟨proof⟩

lemma *bound-parallel-wiring* [interaction-bound]: *interaction-any-bounded-converter*
parallel-wiring 1
⟨proof⟩

6.3 Characterization of wirings

type-synonym $(\text{'a}, \text{'b}, \text{'c}, \text{'d}) \text{ wiring} = (\text{'a} \Rightarrow \text{'c}) \times (\text{'d} \Rightarrow \text{'b})$

inductive $\text{wiring} :: (\text{'a}, \text{'b}) \mathcal{I} \Rightarrow (\text{'c}, \text{'d}) \mathcal{I} \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d}) \text{ converter} \Rightarrow (\text{'a}, \text{'b}, \text{'c}, \text{'d}) \text{ wiring} \Rightarrow \text{bool}$

for $\mathcal{I} \mathcal{I}' \text{ cnv}$

where

wiring:

wiring $\mathcal{I} \mathcal{I}' \text{ cnv} (f, g)$ **if**

$\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \sim \text{map-converter id id f g } 1_C$

$\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \checkmark$

lemmas $\text{wiringI} = \text{wiring}$

hide-fact *wiring*

lemma *wiringD*:

assumes *wiring* $\mathcal{I} \mathcal{I}' \text{ cnv} (f, g)$

shows *wiringD-eq*: $\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \sim \text{map-converter id id f g } 1_C$

and *wiringD-WT*: $\mathcal{I}, \mathcal{I}' \vdash_C \text{cnv} \checkmark$

<proof>

named-theorems *wiring-intro* introduction rules for *wiring*

definition *apply-wiring* $:: (\text{'a}, \text{'b}, \text{'c}, \text{'d}) \text{ wiring} \Rightarrow (\text{'s}, \text{'c}, \text{'d}) \text{ oracle}' \Rightarrow (\text{'s}, \text{'a}, \text{'b}) \text{ oracle}'$

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

lemma *apply-wiring-simps*: $\text{apply-wiring} (f, g) = \text{map-fun id (map-fun f (map-spmf (map-prod g id)))$

<proof>

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

assumes *wiring*: *wiring* $\mathcal{I}1 \mathcal{I}2 \text{ conv fg}$

and *WT*: $\mathcal{I}2 \vdash_{\text{res}} \text{RES res s} \checkmark$

and *outs*: *outs-I* $\mathcal{I}1 = \text{UNIV}$

shows $\text{conv} \triangleright \text{RES res s} = \text{RES (apply-wiring fg res) s}$

<proof>

lemma *wiring-id-converter* [*simp*, *wiring-intro*]: *wiring* $\mathcal{I} \mathcal{I} 1_C (id, id)$

<proof>

lemma *apply-wiring-id* [*simp*]: *apply-wiring* $(id, id) \text{ res} = \text{res}$

<proof>

definition *attach-wiring* $:: (\text{'a}, \text{'b}, \text{'c}, \text{'d}) \text{ wiring} \Rightarrow (\text{'s} \Rightarrow \text{'c} \Rightarrow (\text{'d} \times \text{'s}, \text{'e}, \text{'f}) \text{ gpv}) \Rightarrow (\text{'s} \Rightarrow \text{'a} \Rightarrow (\text{'b} \times \text{'s}, \text{'e}, \text{'f}) \text{ gpv})$

where *attach-wiring* $= (\lambda(f, g). \text{map-fun id (map-fun f (map-gpv (map-prod g id) id)))$

lemma *attach-wiring-simps*: $\text{attach-wiring } (f, g) = \text{map-fun id } (\text{map-fun } f \text{ (map-gpv } (\text{map-prod } g \text{ id) id}))$
 ⟨proof⟩

lemma *comp-wiring-converter-of-callee*:
assumes *wiring*: $\text{wiring } \mathcal{I}1 \ \mathcal{I}2 \ \text{conv } w$
and *WT*: $\mathcal{I}2, \mathcal{I}3 \vdash_C \text{CNV callee } s \ \checkmark$
shows $\mathcal{I}1, \mathcal{I}3 \vdash_C \text{conv} \odot \text{CNV callee } s \sim \text{CNV (attach-wiring } w \text{ callee) } s$
 ⟨proof⟩

definition *comp-wiring* :: $(\ 'a, \ 'b, \ 'c, \ 'd) \text{ wiring} \Rightarrow (\ 'c, \ 'd, \ 'e, \ 'f) \text{ wiring} \Rightarrow (\ 'a, \ 'b, \ 'e, \ 'f) \text{ wiring}$ (**infixl** \circ_w 55)
where $\text{comp-wiring} = (\lambda(f, g) (f', g'). (f' \circ f, g \circ g'))$

lemma *comp-wiring-simps*: $\text{comp-wiring } (f, g) (f', g') = (f' \circ f, g \circ g')$
 ⟨proof⟩

lemma *wiring-comp-converterI* [*wiring-intro*]:
 $\text{wiring } \mathcal{I} \ \mathcal{I}'' (\text{conv1} \odot \text{conv2}) (fg \circ_w fg')$ **if** $\text{wiring } \mathcal{I} \ \mathcal{I}' \ \text{conv1 } fg \ \text{wiring } \mathcal{I}' \ \mathcal{I}''$
 $\text{conv2 } fg'$
 ⟨proof⟩

definition *parallel2-wiring*
 :: $(\ 'a, \ 'b, \ 'c, \ 'd) \text{ wiring} \Rightarrow (\ 'a', \ 'b', \ 'c', \ 'd') \text{ wiring}$
 $\Rightarrow (\ 'a + \ 'a', \ 'b + \ 'b', \ 'c + \ 'c', \ 'd + \ 'd') \text{ wiring}$ (**infix** $|_w$ 501) **where**
 $\text{parallel2-wiring} = (\lambda(f, g) (f', g'). (\text{map-sum } f \ f', \ \text{map-sum } g \ g'))$

lemma *parallel2-wiring-simps*:
 $\text{parallel2-wiring } (f, g) (f', g') = (\text{map-sum } f \ f', \ \text{map-sum } g \ g')$
 ⟨proof⟩

lemma *wiring-parallel-converter2* [*simp*, *wiring-intro*]:
assumes $\text{wiring } \mathcal{I}1 \ \mathcal{I}1' \ \text{conv1 } fg$
and $\text{wiring } \mathcal{I}2 \ \mathcal{I}2' \ \text{conv2 } fg'$
shows $\text{wiring } (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) (\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2') (\text{conv1} \mid_{=} \text{conv2}) (fg \mid_w fg')$
 ⟨proof⟩

lemma *apply-parallel2* [*simp*]:
 $\text{apply-wiring } (fg \mid_w fg') (\text{res1} \oplus_{\circ} \text{res2}) = (\text{apply-wiring } fg \ \text{res1} \oplus_{\circ} \text{apply-wiring } fg' \ \text{res2})$
 ⟨proof⟩

lemma *apply-comp-wiring* [*simp*]: $\text{apply-wiring } (fg \circ_w fg') \ \text{res} = \text{apply-wiring } fg \ (\text{apply-wiring } fg' \ \text{res})$
 ⟨proof⟩

definition *lassocr_w* :: $((\ 'a + \ 'b) + \ 'c, (\ 'd + \ 'e) + \ 'f, \ 'a + (\ 'b + \ 'c), \ 'd + (\ 'e + \ 'f)) \text{ wiring}$

where $lassocr_w = (rsuml, lsumr)$

definition $rassocl_w :: ('a + ('b + 'c), 'd + ('e + 'f), ('a + 'b) + 'c, ('d + 'e) + 'f)$ wiring

where $rassocl_w = (lsumr, rsuml)$

definition $swap_w :: ('a + 'b, 'c + 'd, 'b + 'a, 'd + 'c)$ wiring **where**
 $swap_w = (swap-sum, swap-sum)$

lemma *wiring-lassocr* [simp, wiring-intro]:
 $wiring ((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3) (\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)) lassocr_C lassocr_w$
(proof)

lemma *wiring-rassocl* [simp, wiring-intro]:
 $wiring (\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)) ((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3) rassocl_C rassocl_w$
(proof)

lemma *wiring-swap* [simp, wiring-intro]: $wiring (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}1) swap_C$
 $swap_w$
(proof)

lemma *apply-lassocr_w* [simp]: $apply-wiring lassocr_w (res1 \oplus_O res2 \oplus_O res3) =$
 $(res1 \oplus_O res2) \oplus_O res3$
(proof)

lemma *apply-rassocl_w* [simp]: $apply-wiring rassocl_w ((res1 \oplus_O res2) \oplus_O res3) =$
 $res1 \oplus_O res2 \oplus_O res3$
(proof)

lemma *apply-swap_w* [simp]: $apply-wiring swap_w (res1 \oplus_O res2) = res2 \oplus_O res1$
(proof)

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 \mathcal{I} -*real* :: $security \Rightarrow ('a, 'b)$ \mathcal{I}
and \mathcal{I} -*ideal* :: $security \Rightarrow ('c, 'd)$ \mathcal{I}

and \mathcal{I} -common :: security \Rightarrow ('e, 'f) \mathcal{I}
and bound :: security \Rightarrow enat
and lossless :: bool
assumes WT-real [WT-intro]: $\bigwedge \eta. \mathcal{I}$ -real $\eta \oplus_{\mathcal{I}} \mathcal{I}$ -common $\eta \vdash_{\text{res}}$ real-resource
 $\eta \checkmark$
and WT-ideal [WT-intro]: $\bigwedge \eta. \mathcal{I}$ -ideal $\eta \oplus_{\mathcal{I}} \mathcal{I}$ -common $\eta \vdash_{\text{res}}$ ideal-resource η
 \checkmark
and WT-sim [WT-intro]: $\bigwedge \eta. \mathcal{I}$ -real η, \mathcal{I} -ideal $\eta \vdash_C$ sim $\eta \checkmark$
and adv: $\bigwedge \mathcal{A} ::$ security \Rightarrow ('a + 'e, 'b + 'f) distinguisher.
 $\llbracket \bigwedge \eta. \mathcal{I}$ -real $\eta \oplus_{\mathcal{I}} \mathcal{I}$ -common $\eta \vdash_g \mathcal{A} \eta \checkmark$;
 $\bigwedge \eta. \text{interaction-bounded-by} (\lambda \cdot \text{True}) (\mathcal{A} \eta) (\text{bound } \eta)$;
 $\bigwedge \eta. \text{lossless} \Rightarrow \text{plossless-gpv} (\mathcal{I}$ -real $\eta \oplus_{\mathcal{I}} \mathcal{I}$ -common $\eta) (\mathcal{A} \eta) \rrbracket$
 $\Rightarrow \text{negligible} (\lambda \eta. \text{advantage} (\mathcal{A} \eta) (\text{sim } \eta \models 1_C \triangleright \text{ideal-resource } \eta) (\text{real-resource } \eta))$

locale constructive-security =
constructive-security-aux real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -ideal \mathcal{I} -common
bound lossless
for real-resource :: security \Rightarrow ('a + 'e, 'b + 'f) resource
and ideal-resource :: security \Rightarrow ('c + 'e, 'd + 'f) resource
and sim :: security \Rightarrow ('a, 'b, 'c, 'd) converter
and \mathcal{I} -real :: security \Rightarrow ('a, 'b) \mathcal{I}
and \mathcal{I} -ideal :: security \Rightarrow ('c, 'd) \mathcal{I}
and \mathcal{I} -common :: security \Rightarrow ('e, 'f) \mathcal{I}
and bound :: security \Rightarrow enat
and lossless :: bool
and w :: security \Rightarrow ('c, 'd, 'a, 'b) wiring
+
assumes correct: $\exists \text{cnv}. \forall \mathcal{D} ::$ security \Rightarrow ('c + 'e, 'd + 'f) distinguisher.
 $(\forall \eta. \mathcal{I}$ -ideal $\eta \oplus_{\mathcal{I}} \mathcal{I}$ -common $\eta \vdash_g \mathcal{D} \eta \checkmark)$
 $\longrightarrow (\forall \eta. \text{interaction-bounded-by} (\lambda \cdot \text{True}) (\mathcal{D} \eta) (\text{bound } \eta))$
 $\longrightarrow (\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv} (\mathcal{I}$ -ideal $\eta \oplus_{\mathcal{I}} \mathcal{I}$ -common $\eta) (\mathcal{D} \eta))$
 $\longrightarrow (\forall \eta. \text{wiring} (\mathcal{I}$ -ideal $\eta) (\mathcal{I}$ -real $\eta) (\text{cnv } \eta) (w \eta) \wedge$
 $\text{negligible} (\lambda \eta. \text{advantage} (\mathcal{D} \eta) (\text{ideal-resource } \eta) (\text{cnv } \eta \models 1_C \triangleright \text{real-resource } \eta))$

locale constructive-security2 =
constructive-security-aux real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -ideal \mathcal{I} -common
bound lossless
for real-resource :: security \Rightarrow ('a + 'e, 'b + 'f) resource
and ideal-resource :: security \Rightarrow ('c + 'e, 'd + 'f) resource
and sim :: security \Rightarrow ('a, 'b, 'c, 'd) converter
and \mathcal{I} -real :: security \Rightarrow ('a, 'b) \mathcal{I}
and \mathcal{I} -ideal :: security \Rightarrow ('c, 'd) \mathcal{I}
and \mathcal{I} -common :: security \Rightarrow ('e, 'f) \mathcal{I}
and bound :: security \Rightarrow enat
and lossless :: bool
and w :: security \Rightarrow ('c, 'd, 'a, 'b) wiring

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

lemma *constructive-security*:

constructive-security real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -ideal \mathcal{I} -common
bound lossless w
 ⟨proof⟩

sublocale *constructive-security real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -ideal \mathcal{I} -common*
bound lossless w
 ⟨proof⟩

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]: $\bigwedge \eta. interaction\text{-any}\text{-bounded}\text{-converter} (sim\text{-outer}$*
 $\eta) (bound\text{-sim } \eta)$
and *bound-le: $\bigwedge \eta. bound\text{-outer } \eta * \max (bound\text{-sim } \eta) 1 \leq bound\text{-inner } \eta$*
and *lossless-sim [plossless-intro]: $\bigwedge \eta. lossless\text{-inner} \implies plossless\text{-converter} (\mathcal{I}\text{-real}$*
 $\eta) (\mathcal{I}\text{-middle } \eta) (sim\text{-outer } \eta)$
shows *constructive-security real ideal ($\lambda \eta. sim\text{-outer } \eta \odot sim\text{-inner } \eta) \mathcal{I}\text{-real}$*
 $\mathcal{I}\text{-inner } \mathcal{I}\text{-common bound-outer} (lossless\text{-outer} \vee lossless\text{-inner}) (\lambda \eta. w1 \ \eta \circ_w w2$
 $\eta)$
 ⟨proof⟩

theorem (*in constructive-security*) *lifting*:

assumes *WT-conv [WT-intro]: $\bigwedge \eta. \mathcal{I}\text{-common}' \ \eta, \mathcal{I}\text{-common } \eta \vdash_C conv \ \eta \checkmark$*
and *bound [interaction-bound]: $\bigwedge \eta. interaction\text{-any}\text{-bounded}\text{-converter} (conv \ \eta)$*
(bound-conv $\eta)$
and *bound-le: $\bigwedge \eta. bound' \ \eta * \max (bound\text{-conv } \eta) 1 \leq bound \ \eta$*
and *lossless [plossless-intro]: $\bigwedge \eta. lossless \implies plossless\text{-converter} (\mathcal{I}\text{-common}'$*
 $\eta) (\mathcal{I}\text{-common } \eta) (conv \ \eta)$
shows *constructive-security*
($\lambda \eta. 1_C \models conv \ \eta \triangleright real\text{-resource } \eta) (\lambda \eta. 1_C \models conv \ \eta \triangleright ideal\text{-resource } \eta) sim$
 $\mathcal{I}\text{-real } \mathcal{I}\text{-ideal } \mathcal{I}\text{-common}' bound' lossless w$
 ⟨proof⟩

theorem *constructive-security-trivial*:

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

shows *constructive-security* $\text{res res } (\lambda\cdot. 1_C) \mathcal{I} \mathcal{I} \mathcal{I}\text{-common bound lossless } (\lambda\cdot. (id, id))$
 $\langle \text{proof} \rangle$

theorem *parallel-constructive-security*:

assumes *constructive-security* $\text{real1 ideal1 sim1 } \mathcal{I}\text{-real1 } \mathcal{I}\text{-inner1 } \mathcal{I}\text{-common1 bound1 lossless1 } w1$

assumes *constructive-security* $\text{real2 ideal2 sim2 } \mathcal{I}\text{-real2 } \mathcal{I}\text{-inner2 } \mathcal{I}\text{-common2 bound2 lossless2 } w2$

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

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

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* $(\lambda \eta. \text{parallel-wiring } \triangleright \text{real1 } \eta \parallel \text{real2 } \eta) (\lambda \eta. \text{parallel-wiring } \triangleright \text{ideal1 } \eta \parallel \text{ideal2 } \eta) (\lambda \eta. \text{sim1 } \eta \mid = \text{sim2 } \eta)$

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

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

$\langle \text{proof} \rangle$

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

assumes *WT-res*: $\bigwedge \eta. \mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta \vdash_{\text{res}} \text{res } \eta \checkmark$

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

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

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

$(\lambda \eta. \mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real } \eta) (\lambda \eta. \mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-ideal } \eta) (\lambda \eta. \mathcal{I}\text{-common}' \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) \text{ bound lossless } (\lambda \eta. (id, id) \mid_w w \eta)$

$\langle \text{proof} \rangle$

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

assumes *WT-res*: $\bigwedge \eta. \mathcal{I}\text{-res } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta \vdash_{\text{res}} \text{res } \eta \checkmark$

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

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

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

$(\lambda \eta. \mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-res } \eta) (\lambda \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-res } \eta) (\lambda \eta. \mathcal{I}\text{-common } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta) \text{ bound lossless } (\lambda \eta. w \eta \mid_w (id, id))$

$\langle \text{proof} \rangle$

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*]: $\bigwedge \eta. \text{lossless} \implies \text{lossless-resource } (\mathcal{I}\text{-res } \eta) (\text{res } \eta)$

```

shows constructive-security ( $\lambda\eta$ . parallel-resource1-wiring  $\triangleright$  res  $\eta$   $\parallel$  real-resource
 $\eta$ )
  ( $\lambda\eta$ . parallel-resource1-wiring  $\triangleright$  res  $\eta$   $\parallel$  ideal-resource  $\eta$ ) sim
   $\mathcal{I}$ -real  $\mathcal{I}$ -ideal ( $\lambda\eta$ .  $\mathcal{I}$ -res  $\eta \oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\eta$ ) bound lossless w
  <proof>

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  $\Rightarrow$  'r option, 'm, 'r) oracle' where
  rnd-oracle f m = (case f m of
    (Some r)  $\Rightarrow$  return-spmf (r, f)
  | None  $\Rightarrow$  do {
    r  $\leftarrow$  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* **where** *Forward-auth* \equiv *ForwardOrEdit* ($\lambda ()$)
abbreviation *Forward-insec* :: $'a$ *insec-query* **where** *Forward-insec* \equiv *ForwardOrEdit* (*None*)
abbreviation *Edit* :: $'a \Rightarrow 'a$ *insec-query* **where** *Edit* $m \equiv$ *ForwardOrEdit* (*Some* m)
adhoc-overloading *Forward* *Forward-auth*
adhoc-overloading *Forward* *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* ($\lambda ()$, *Store* m)
| *send-oracle* s m = *return-spmf* ($\lambda ()$, 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* ⟨*proof*⟩

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* ⟨*proof*⟩

end

fun *insec-query-of* :: *auth-query* ⇒ *'m insec-query where*
insec-query-of Forward = *Forward*
| *insec-query-of Drop* = *Drop*
| *insec-query-of Look* = *Look*

abbreviation (*input*) *auth-response-of* :: (*'mac × 'm*) *option* ⇒ *'m option*
where *auth-response-of* ≡ *map-option snd*

abbreviation *insec-auth-wiring* :: (*auth-query*, *'m option*, (*'mac × 'm*) *insec-query*,
(*'mac × 'm*) *option*) *wiring*
where *insec-auth-wiring* ≡ (*insec-query-of*, *auth-response-of*)

8.3.4 Secure channel

locale *sec-channel*

begin

fun *sec-oracle* :: (*'a list cstate*, *auth-query*, *nat option*) *oracle' where*
sec-oracle (*Store m*) *Forward* = *return-spmf* (*None*, *Collect m*)
| *sec-oracle* (*Store m*) *Drop* = *return-spmf* (*None*, *Fail*)
| *sec-oracle* (*Store m*) *Look* = *return-spmf* (*Some (length m)*, *Store m*)
| *sec-oracle s* - = *return-spmf* (*None*, *s*)

sublocale *channel sec-oracle* ⟨*proof*⟩

end

abbreviation (*input*) *auth-query-of* :: *auth-query* ⇒ *auth-query*

where $auth\text{-}query\text{-}of \equiv id$

abbreviation $(input) sec\text{-}response\text{-}of :: 'a list option \Rightarrow nat option$
where $sec\text{-}response\text{-}of \equiv map\text{-}option length$

abbreviation $auth\text{-}sec\text{-}wiring :: (auth\text{-}query, nat option, auth\text{-}query, 'a list option)$
wiring

where $auth\text{-}sec\text{-}wiring \equiv (auth\text{-}query\text{-}of, sec\text{-}response\text{-}of)$

8.4 Cipher converter

locale $cipher =$

$AUTH: auth\text{-}channel + KEY: key key\text{-}alg$

for $key\text{-}alg :: 'k spmf +$

fixes $enc\text{-}alg :: 'k \Rightarrow 'm \Rightarrow 'c spmf$

and $dec\text{-}alg :: 'k \Rightarrow 'c \Rightarrow 'm option$

begin

definition $enc :: ('m, unit, unit + 'c, 'k + unit) converter$ **where**

$enc \equiv CNV (stateless\text{-}callee (\lambda m. do \{$
 $k \leftarrow Pause (Inl ()) Done;$
 $c \leftarrow lift\text{-}spmf (enc\text{-}alg (projl k) m);$
 $(- :: 'k + unit) \leftarrow Pause (Inr c) Done;$
 $Done (())$
 $\}) ()$

definition $dec :: (unit, 'm option, unit + unit, 'k + 'c option) converter$ **where**

$dec \equiv CNV (stateless\text{-}callee (\lambda -. Pause (Inr ()) (\lambda c'.$
 $case c' of Inr (Some c) \Rightarrow (do \{$
 $k \leftarrow Pause (Inl ()) Done;$
 $Done (dec\text{-}alg (projl k) c) \})$
 $| - \Rightarrow Done None)$
 $\}) ()$

definition $\pi^E :: (auth\text{-}query, 'c option, auth\text{-}query, 'c option) converter$ (π^E)

where

$\pi^E \equiv 1_C$

definition $routing \equiv (1_C |_{=} lassocr_C) \odot swap\text{-}lassocr \odot (1_C |_{=} (1_C |_{=} swap\text{-}lassocr)$
 $\odot swap\text{-}lassocr) \odot rassocl_C$

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

lemma $res\text{-}alt\text{-}def: res = ((1_C |_{=} enc |_{=} dec) \odot (1_C |_{=} parallel\text{-}wiring)) \triangleright parallel\text{-}resource1\text{-}wiring$
 $\triangleright (KEY.res \parallel AUTH.res)$

$\langle proof \rangle$

end

8.5 Message authentication converter

locale *macode* =

INSEC: *insec-channel* + *RO*: *rorc range*

for *range* :: '*r* set +

fixes *mac-alg* :: '*r* ⇒ '*m* ⇒ '*a* *spmf*

begin

definition *enm* :: ('*m*, *unit*, '*m* + ('*a* × '*m*), '*r* + *unit*) *converter* **where**

enm ≡ *CNV* (λbs *m*. *if bs*

then *Done* (*()*, *True*)

else *do* {

r ← *Pause* (*Inl m*) *Done*;

a ← *lift-spmf* (*mac-alg* (*projl r*) *m*);

(- :: '*r* + *unit*) ← *Pause* (*Inr* (*a*, *m*)) *Done*;

Done (*()*, *True*)

}) *False*

definition *dem* :: (*unit*, '*m* *option*, '*m* + *unit*, '*r* + ('*a* × '*m*) *option*) *converter* **where**

dem ≡ *CNV* (*stateless-callee* ($\lambda-$. *Pause* (*Inr* (*()*)) ($\lambda am'$.

case am' of *Inr* (*Some* (*a*, *m*)) ⇒ (*do* {

r ← *Pause* (*Inl m*) *Done*;

a' ← *lift-spmf* (*mac-alg* (*projl r*) *m*);

Done (*if a' = a* then *Some m* else *None*) })

| - ⇒ *Done None*)

)) (*()*)

definition π^E :: (('*a* × '*m*) *insec-query*, ('*a* × '*m*) *option*, ('*a* × '*m*) *insec-query*, ('*a* × '*m*) *option*) *converter* (π^E) **where**

π^E ≡ *1_C*

definition *routing* ≡ (*1_C* |₌ *lassocr_C*) ∘ *swap-lassocr* ∘ (*1_C* |₌ (*1_C* |₌ *swap-lassocr*) ∘ *swap-lassocr*) ∘ *rassocl_C*

definition *res* = (*1_C* |₌ *enm* |₌ *dem*) ▷ (*1_C* |₌ *parallel-wiring*) ▷ *parallel-resource1-wiring* ▷ (*RO.res* || *INSEC.res*)

end

lemma *interface-wiring*:

(*cnv-advr* |₌ *cnv-send* |₌ *cnv-recv*) ▷ (*1_C* |₌ *parallel-wiring*) ▷ *parallel-resource1-wiring*

▷

(*RES* (*res2-send* ⊕_O *res2-recv*) *res2-s* || *RES* (*res1-advr* ⊕_O *res1-send* ⊕_O *res1-recv*) *res1-s*)

=

cnv-advr |₌ *cnv-send* |₌ *cnv-recv* ▷

RES (\dagger *res1-advr* ⊕_O (*res2-send*† ⊕_O \dagger *res1-send*) ⊕_O *res2-recv*† ⊕_O \dagger *res1-recv*) (*res2-s*, *res1-s*)

(is - ▷ ?L1 ▷ ?L2 ▷ ?L3 = - ▷ ?R)
 ⟨proof⟩

definition *id'* where *id' = id*

end

9 Security of one-time-pad encryption

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

System-Construction

begin

definition *key* :: *security* ⇒ *bool list spmf* **where**

key η ≡ *spmf-of-set* (*nlists UNIV* η)

definition *enc* :: *security* ⇒ *bool list* ⇒ *bool list* ⇒ *bool list spmf* **where**

enc η *k m* ≡ *return-spmf* (*k* [⊕] *m*)

definition *dec* :: *security* ⇒ *bool list* ⇒ *bool list* ⇒ *bool list option* **where**

dec η *k c* ≡ *Some* (*k* [⊕] *c*)

definition *sim* :: '*b list option* ⇒ '*a* ⇒ ('*b list option* × '*b list option*, '*a*, *nat option*) *gpv* **where**

sim *c q* ≡ (do {
 lo ← *Pause* *q Done*;
 (case *lo* of
 Some n ⇒ if *c = None*
 then do {
 x ← *lift-spmf* (*spmf-of-set* (*nlists UNIV n*));
 Done (*Some x*, *Some x*)
 else *Done* (*c*, *c*)
 | *None* ⇒ *Done* (*None*, *c*))})

context

fixes η :: *security*

begin

private definition *key-channel-send* :: *bool list option* × *bool list cstate*

⇒ *bool list* ⇒ (*unit* × *bool list option* × *bool list cstate*) *spmf* **where**

key-channel-send *s m* ≡ do {
 (*k*, *s*) ← (*key.key-oracle* (*key* η))† *s* ();
 c ← *enc* η *k m*;
 (-, *s*) ← †*channel.send-oracle* *s c*;
 return-spmf ((), *s*)}

private definition *key-channel-recv* :: *bool list option* × *bool list cstate*

$\Rightarrow 'a \Rightarrow (\text{bool list option} \times \text{bool list option} \times \text{bool list cstate}) \text{ spmf}$ **where**
 $\text{key-channel-recv } s \ m \equiv \text{do} \{$
 $\quad (c, s) \leftarrow \dagger \text{channel.recv-oracle } s \ ();$
 $\quad (\text{case } c \text{ of } \text{None} \Rightarrow \text{return-spmf } (\text{None}, s)$
 $\quad | \text{Some } c' \Rightarrow \text{do} \{$
 $\quad \quad (k, s) \leftarrow (\text{key.key-oracle } (\text{key } \eta)) \dagger s \ ();$
 $\quad \quad \text{return-spmf } (\text{dec } \eta \ k \ c', s)\})\}$

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

$\text{callee-sec-channel } \text{callee} \equiv \text{lift-state-oracle extend-state-oracle } (\text{attach-callee } \text{callee}$
 $\text{sec-channel.sec-oracle})$

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**
 $S \ (\text{return-spmf } (\text{None}, (), \text{Void}))$
 $\quad (\text{return-spmf } (\text{None}, \text{Void}))$
 $| S \ (\text{return-spmf } (\text{None}, (), \text{Store } \text{plain}))$
 $\quad (\text{map-spmf } (\lambda \text{key}. (\text{Some } \text{key}, \text{Store } (\text{key } [\oplus] \text{plain}))) (\text{spmf-of-set } (\text{nlists } \text{UNIV}$
 $\eta)))$
if $\text{length } \text{plain} = \text{id}' \ \eta$
 $| S \ (\text{return-spmf } (\text{None}, (), \text{Collect } \text{plain}))$
 $\quad (\text{map-spmf } (\lambda \text{key}. (\text{Some } \text{key}, \text{Collect } (\text{key } [\oplus] \text{plain}))) (\text{spmf-of-set } (\text{nlists}$
 $\text{UNIV } \eta)))$
if $\text{length } \text{plain} = \text{id}' \ \eta$
 $| S \ (\text{return-spmf } (\text{Some } (\text{key } [\oplus] \text{plain}), (), \text{Store } \text{plain}))$
 $\quad (\text{return-spmf } (\text{Some } \text{key}, \text{Store } (\text{key } [\oplus] \text{plain})))$
if $\text{length } \text{plain} = \text{id}' \ \eta \ \text{length } \text{key} = \text{id}' \ \eta$ **for** key
 $| S \ (\text{return-spmf } (\text{Some } (\text{key } [\oplus] \text{plain}), (), \text{Collect } \text{plain}))$
 $\quad (\text{return-spmf } (\text{Some } \text{key}, \text{Collect } (\text{key } [\oplus] \text{plain})))$
if $\text{length } \text{plain} = \text{id}' \ \eta \ \text{length } \text{key} = \text{id}' \ \eta$ **for** key
 $| S \ (\text{return-spmf } (\text{None}, (), \text{Fail}))$
 $\quad (\text{map-spmf } (\lambda x. (\text{Some } x, \text{Fail})) (\text{spmf-of-set } (\text{nlists } \text{UNIV } \eta)))$
 $| S \ (\text{return-spmf } (\text{Some } (\text{key } [\oplus] \text{plain}), (), \text{Fail}))$
 $\quad (\text{return-spmf } (\text{Some } \text{key}, \text{Fail}))$
if $\text{length } \text{plain} = \text{id}' \ \eta \ \text{length } \text{key} = \text{id}' \ \eta$ **for** $\text{key } \text{plain}$

lemma *resources-indistinguishable*:

shows $(\text{UNIV } \langle + \rangle \ \text{nlists } \text{UNIV } (\text{id}' \ \eta) \ \langle + \rangle \ \text{UNIV}) \vdash_R$
 $\text{RES } (\text{callee-sec-channel } \text{sim} \oplus_O \dagger \dagger \text{channel.send-oracle} \oplus_O \dagger \dagger \text{channel.recv-oracle})$
 $(\text{None} :: \text{bool list option}, (), \text{Void})$
 \approx
 $\text{RES } (\dagger \text{auth-channel.auth-oracle} \oplus_O \text{key-channel-send} \oplus_O \text{key-channel-recv})$
 $(\text{None} :: \text{bool list option}, \text{Void})$
 $(\text{is } ?A \vdash_R \text{RES } (?L1 \oplus_O ?L2 \oplus_O ?L3) ?SL \approx \text{RES } (?R1 \oplus_O ?R2 \oplus_O ?R3)$
 $?SR)$
 $\langle \text{proof} \rangle$

lemma *real-resource-wiring*:

shows *cipher.res* (*key* η) (*enc* η) (*dec* η)
 = *RES* (\dagger *auth-channel.auth-oracle* \oplus_O *key-channel-send* \oplus_O *key-channel-recv*)
 (*None*, *Void*)
including *lifting-syntax*
 \langle *proof* \rangle

lemma *ideal-resource-wiring*:
shows (*CNV callee* *s*) $|=$ $1_C \triangleright$ *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** $?L1 \triangleright - = ?R$)
 \langle *proof* \rangle

end

lemma *eq-I-gpv-Done1*:
eq-I-gpv *A* \mathcal{I} (*Done* *x*) *gpv* \longleftrightarrow *lossless-spmf* (*the-gpv* *gpv*) \wedge ($\forall a \in \text{set-spmf}$
 (*the-gpv* *gpv*). *eq-I-generat* *A* \mathcal{I} (*eq-I-gpv* *A* \mathcal{I}) (*Pure* *x*) *a*)
 \langle *proof* \rangle

lemma *eq-I-gpv-Done2*:
eq-I-gpv *A* \mathcal{I} *gpv* (*Done* *x*) \longleftrightarrow *lossless-spmf* (*the-gpv* *gpv*) \wedge ($\forall a \in \text{set-spmf}$
 (*the-gpv* *gpv*). *eq-I-generat* *A* \mathcal{I} (*eq-I-gpv* *A* \mathcal{I}) *a* (*Pure* *x*))
 \langle *proof* \rangle

context begin

interpretation *CIPHER*: *cipher* *key* η *enc* η *dec* η **for** η \langle *proof* \rangle

interpretation *S-CHAN*: *sec-channel* \langle *proof* \rangle

lemma *one-time-pad*:

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

shows
constructive-security2 *CIPHER.res* ($\lambda-. \text{S-CHAN.res}$) ($\lambda-. \text{CNV sim None}$)
I-real \mathcal{I} -*ideal* \mathcal{I} -*common* ($\lambda-. \infty$) *False* ($\lambda-. \text{auth-sec-wiring}$)
 \langle *proof* \rangle

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**
 $mac \ \eta \ r \ m \equiv return\text{-}spmf \ r$

definition $vld :: security \Rightarrow bool \ list \ set$ **where**
 $vld \ \eta \equiv nlists \ UNIV \ \eta$

fun $valid\text{-}mac\text{-}query :: security \Rightarrow (bool \ list \times bool \ list) \ insec\text{-}query \Rightarrow bool$ **where**
 $valid\text{-}mac\text{-}query \ \eta \ (ForwardOrEdit \ (Some \ (a, \ m))) \longleftrightarrow a \in vld \ \eta \wedge m \in vld \ \eta$
 $| \ valid\text{-}mac\text{-}query \ \eta \ - = True$

fun $sim :: ('b \ list \times 'b \ list) \ option + unit \Rightarrow ('b \ list \times 'b \ list) \ insec\text{-}query$
 $\Rightarrow (('b \ list \times 'b \ list) \ option \times (('b \ list \times 'b \ list) \ option + unit), \ auth\text{-}query, 'b$
 $list \ option) \ gpv$ **where**

$sim \ (Inr \ ()) \quad - \quad = Done \ (None, \ Inr \ ())$
 $| \ sim \ (Inl \ None) \quad (Edit \ (a', \ m')) = do \ { \ - \leftarrow \ Pause \ Drop \ Done; \ Done$
 $(None, \ Inr \ ()) \}$
 $| \ sim \ (Inl \ (Some \ (a, \ m))) \ (Edit \ (a', \ m')) = (if \ a = a' \wedge m = m'$
 $\ then \ do \ { \ - \leftarrow \ Pause \ Forward \ Done; \ Done \ (None, \ Inl \ (Some \ (a, \ m))) \}$
 $\ else \ do \ { \ - \leftarrow \ Pause \ Drop \ Done; \ Done \ (None, \ Inr \ ()) \}$
 $| \ sim \ (Inl \ None) \quad Forward \quad = do \ {$
 $\ \ \ \ \ \ Pause \ Forward \ Done;$
 $\ \ \ \ \ \ Done \ (None, \ Inl \ None) \}$
 $| \ sim \ (Inl \ (Some \ -)) \quad Forward \quad = do \ {$
 $\ \ \ \ \ \ Pause \ Forward \ Done;$
 $\ \ \ \ \ \ Done \ (None, \ Inr \ ()) \}$
 $| \ sim \ (Inl \ None) \quad Drop \quad = do \ {$
 $\ \ \ \ \ \ Pause \ Drop \ Done;$
 $\ \ \ \ \ \ Done \ (None, \ Inl \ None) \}$
 $| \ sim \ (Inl \ (Some \ -)) \quad Drop \quad = do \ {$
 $\ \ \ \ \ \ Pause \ Drop \ Done;$
 $\ \ \ \ \ \ Done \ (None, \ Inr \ ()) \}$
 $| \ sim \ (Inl \ (Some \ (a, \ m))) \ Look \quad = do \ {$
 $\ \ \ \ \ \ lo \leftarrow \ Pause \ Look \ Done;$
 $\ \ \ \ \ \ (case \ lo \ of$
 $\ \ \ \ \ \ \ \ \ \ \ \ Some \ m \Rightarrow Done \ (Some \ (a, \ m), \ Inl \ (Some \ (a, \ m)))$
 $\ \ \ \ \ \ \ \ \ \ \ \ | \ None \ \Rightarrow Done \ (None, \ Inl \ (Some \ (a, \ m)))) \}$
 $| \ sim \ (Inl \ None) \quad Look \quad = do \ {$
 $\ \ \ \ \ \ lo \leftarrow \ Pause \ Look \ Done;$
 $\ \ \ \ \ \ (case \ lo \ of$
 $\ \ \ \ \ \ \ \ \ \ \ \ Some \ m \Rightarrow do \ {$
 $\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ a \leftarrow \ lift\text{-}spmf \ (spmf\text{-}of\text{-}set \ (nlists \ UNIV \ (length \ m)));$
 $\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ Done \ (Some \ (a, \ m), \ Inl \ (Some \ (a, \ m))) \}$
 $\ \ \ \ \ \ \ \ \ \ \ \ | \ None \ \Rightarrow Done \ (None, \ Inl \ None) \}$

context

fixes $\eta :: security$

begin

private definition *rorc-channel-send* :: ((*bool* × *unit*) × (*bool list* ⇒ *bool list option*) × (*bool list* × *bool list*) *cstate*, *bool list*, *unit*) *oracle'* **where**
 rorc-channel-send *s m* ≡ (if *fst* (*fst s*)
 then *return-spmf* ((*True*, ()), *snd s*)
 else do {
 (*r*, *s*) ← (*rorc.rnd-oracle* (*rnd η*))† (*snd s*) *m*;
 a ← *mac η r m*;
 (*·*, *s*) ← †*channel.send-oracle s* (*a*, *m*);
 return-spmf ((*True*, ()), *s*)
 })

private definition *rorc-channel-recv* :: ((*bool* × *unit*) × (*bool list* ⇒ *bool list option*) × (*bool list* × *bool list*) *cstate*, *unit*, *bool list option*) *oracle'* **where**
 rorc-channel-recv s q ≡ do {
 (*m*, *s*) ← ††*channel.recv-oracle s* ();
 (case *m* of
 None ⇒ *return-spmf* (*None*, *s*)
 | *Some* (*a*, *m*) ⇒ do {
 (*r*, *s*) ← †(*rorc.rnd-oracle* (*rnd η*))† *s m*;
 a' ← *mac η r m*;
 return-spmf (if *a' = a* then *Some m* else *None*, *s*)}
 }

private definition *rorc-channel-recv-f* :: ((*bool list* ⇒ *bool list option*) × (*bool list* × *bool list*) *cstate*, *unit*, *bool list option*) *oracle'* **where**
 rorc-channel-recv-f s q ≡ do {
 (*am*, (*as*, *ams*)) ← †*channel.recv-oracle s* ();
 (case *am* of
 None ⇒ *return-spmf* (*None*, (*as*, *ams*))
 | *Some* (*a*, *m*) ⇒ (case *as m* of
 None ⇒ do {
 a'' :: *bool list* ← *spmf-of-set* (*nlists UNIV η - {a}*);
 a' ← *spmf-of-set* (*nlists UNIV η*);
 (if *a' = a*
 then *return-spmf* (*None*, *as(m := Some a')*, *ams*)
 else *return-spmf* (*None*, *as(m := Some a')*, *ams*)) }
 | *Some a'* ⇒ *return-spmf* (if *a' = a* then *Some m* else *None*, *as*, *ams*))})

private fun *lazy-channel-send* :: (*bool list cstate* × (*bool list* × *bool list*) *option* × (*bool list* ⇒ *bool list option*), *bool list*, *unit*) *oracle'* **where**
 lazy-channel-send (*Void*, *es*) *m* = *return-spmf* ((*Store m*, *es*))
| *lazy-channel-send s* *m* = *return-spmf* ((*·*), *s*)

private fun *lazy-channel-recv* :: (*bool list cstate* × (*bool list* × *bool list*) *option* × (*bool list* ⇒ *bool list option*), *unit*, *bool list option*) *oracle'* **where**
 lazy-channel-recv (*Collect m*, *None*, *as*) () = *return-spmf* (*Some m*, (*Fail*, *None*, *as*))

| *lazy-channel-recv* (*ms*, *Some* (*a'*, *m'*), *as*) () = (case *as m'* of
 None \Rightarrow do {
 a \leftarrow *spmf-of-set* (*rnd* η);
 return-spmf (if *a* = *a'* then *Some m'* else *None*, *cstate.Fail*, *None*, *as* (*m' := Some a*)))
 | *Some a* \Rightarrow *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* \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'* **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 \Rightarrow do {
 a \leftarrow *spmf-of-set* (*rnd* η);
 return-spmf (*Some* (*a*, *m*), *Store m*, *None*, *as* (*m := Some a*)))
 | *Some a* \Rightarrow *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* \times (*bool list* \times *bool list*) *option* \times (*bool list* \Rightarrow *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 \Rightarrow do {
 a \leftarrow *spmf-of-set* (*rnd* η);
 return-spmf (*None*, *Fail*, *None*, *as* (*m' := Some a*)))
 | *Some a* \Rightarrow *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* \equiv *lift-state-oracle extend-state-oracle* (*attach-callee callee auth-channel.auth-oracle*)

private abbreviation
valid-insecQ \equiv {(*x* :: (*bool list* \times *bool list*) *insec-query*). case *x* of
ForwardOrEdit (*Some* (*a*, *m*)) \Rightarrow *length a* = *id' η* \wedge *length m* = *id' η*
 | - \Rightarrow *True*}

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

```

spmf ⇒ bool where
  S (return-spmf (Void, None, Map.empty))
    (return-spmf ((False, ()), Map.empty, Void))
| S (return-spmf (Store m, None, Map.empty))
    (map-spmf (λa. ((True, ()), [m ↦ a], Store (a, m))) (spmf-of-set (nlists UNIV
η)))
if length m = id' η
| S (return-spmf (Collect m, None, Map.empty))
    (map-spmf (λa. ((True, ()), [m ↦ a], Collect (a, m))) (spmf-of-set (nlists
UNIV η)))
if length m = id' η
| S (return-spmf (Store m, None, [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Store (a, m)))
if length m = id' η and length a = id' η
| S (return-spmf (Collect m, None, [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Collect (a, m)))
if length m = id' η and length a = id' η
| S (return-spmf (Fail, None, Map.empty))
    (map-spmf (λa. ((True, ()), [m ↦ a], Fail)) (spmf-of-set (nlists UNIV η)))
if length m = id' η
| S (return-spmf (Fail, None, [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Fail))
if length m = id' η and length a = id' η
| S (return-spmf (Collect m', Some (a', m'), Map.empty))
    (return-spmf ((False, ()), Map.empty, Collect (a', m')))
if length m' = id' η and length a' = id' η
| S (return-spmf (Collect m', Some (a', m'), [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Collect (a', m')))
if length m = id' η and length a = id' η and length m' = id' η and length a' =
id' η
| S (return-spmf (Collect m', Some (a', m'), Map.empty))
    (map-spmf (λx. ((True, ()), [m ↦ x], Collect (a', m'))) (spmf-of-set (nlists
UNIV η)))
if length m = id' η and length m' = id' η and length a' = id' η
| S (map-spmf (λx. (Fail, None, as(m' ↦ x))) spmf-s)
    (map-spmf (λx. ((False, ()), as(m' ↦ x), Fail)) spmf-s)
if length m' = id' η and lossless-spmf spmf-s
| S (map-spmf (λx. (Fail, None, as(m' ↦ x))) spmf-s)
    (map-spmf (λx. ((True, ()), as(m' ↦ x), Fail)) spmf-s)
if length m' = id' η and lossless-spmf spmf-s
| S (return-spmf (Fail, None, [m' ↦ a']))
    (map-spmf (λx. ((True, ()), [m ↦ x, m' ↦ a'], Fail)) (spmf-of-set (nlists UNIV
η)))
if length m = id' η and length m' = id' η and length a' = id' η
| S (map-spmf (λx. (Fail, None, [m' ↦ x])) (spmf-of-set (nlists UNIV η ∩ {x. x
≠ a'})))
    (map-spmf (λx. ((True, ()), [m ↦ fst x, m' ↦ snd x], Fail)) (spmf-of-set (nlists
UNIV η × nlists UNIV η ∩ {x. snd x ≠ a'})))
if length m = id' η and length m' = id' η

```

| S ($\text{map-spmf } (\lambda x. (\text{Fail}, \text{None}, \text{as}(m' \mapsto x))) \text{ spmf-s}$)
 $(\text{map-spmf } (\lambda p. ((\text{True}, ()), \text{as}(m' \mapsto \text{fst } p, m \mapsto \text{snd } p), \text{Fail})) (\text{mk-lossless}$
 $(\text{pair-spmf } \text{spmf-s } (\text{spmf-of-set } (\text{nlists } \text{UNIV } \eta))))$)
if $\text{length } m = \text{id}' \eta$ **and** $\text{length } m' = \text{id}' \eta$ **and** $\text{lossless-spmf } \text{spmf-s}$

private lemma *trace-eq-lazy*:

assumes $\eta > 0$

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

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

\approx

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

$(\text{is } ?A \vdash_R \text{RES } (?L1 \oplus_O ?L2 \oplus_O ?L3) ?SL \approx \text{RES } (?R1 \oplus_O ?R2 \oplus_O ?R3)$
 $?SR)$

<proof> **lemma** *game-difference*:

defines $\mathcal{I} \equiv \mathcal{I}\text{-uniform } (\text{Set.Collect } (\text{valid-mac-query } \eta)) (\text{insert } \text{None } (\text{Some } \langle$
 $\text{nlists } \text{UNIV } \eta \times \text{nlists } \text{UNIV } \eta \rangle)) \oplus_{\mathcal{I}}$

$(\mathcal{I}\text{-uniform } (\text{vld } \eta) \text{UNIV } \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform } \text{UNIV } (\text{insert } \text{None } (\text{Some } \langle \text{vld } \eta \rangle)))$

assumes *bound: interaction-bounded-by'* $(\lambda \cdot. \text{True}) \mathcal{A} q$

and *lossless: plossless-gpv* $\mathcal{I} \mathcal{A}$

and *WT*: $\mathcal{I} \vdash_g \mathcal{A} \checkmark$

shows

$|\text{spmf } (\text{connect } \mathcal{A} (\text{RES } (\text{lazy-channel-insec } \oplus_O \text{ lazy-channel-send } \oplus_O \text{ lazy-channel-recv-f})$
 $(\text{Void}, \text{None}, \text{Map.empty}))) \text{ True} -$

$\text{spmf } (\text{connect } \mathcal{A} (\text{RES } (\text{lazy-channel-insec } \oplus_O \text{ lazy-channel-send } \oplus_O \text{ lazy-channel-recv})$
 $(\text{Void}, \text{None}, \text{Map.empty}))) \text{ True}$

$\leq q / \text{real } (2^{\wedge} \eta) (\text{is } ?LHS \leq -)$

<proof> **inductive** $S' :: (((\text{bool list} \times \text{bool list}) \text{option} + \text{unit}) \times \text{unit} \times \text{bool list}$
 $\text{cstate}) \text{ spmf} \Rightarrow$

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

$S' (\text{return-spmf } (\text{Inl } \text{None}, ()), \text{Void}))$

$(\text{return-spmf } (\text{Void}, \text{None}, \text{Map.empty}))$

| $S' (\text{return-spmf } (\text{Inl } \text{None}, ()), \text{Store } m))$

$(\text{return-spmf } (\text{Store } m, \text{None}, \text{Map.empty}))$

if $\text{length } m = \text{id}' \eta$

| $S' (\text{return-spmf } (\text{Inr } ()), ()), \text{Collect } m))$

$(\text{return-spmf } (\text{Collect } m, \text{None}, \text{Map.empty}))$

if $\text{length } m = \text{id}' \eta$

| $S' (\text{return-spmf } (\text{Inl } (\text{Some } (a, m)), ()), \text{Store } m))$

$(\text{return-spmf } (\text{Store } m, \text{None}, [m \mapsto a]))$

if $\text{length } m = \text{id}' \eta$

| $S' (\text{return-spmf } (\text{Inr } ()), ()), \text{Collect } m))$

$(\text{return-spmf } (\text{Collect } m, \text{None}, [m \mapsto a]))$

if $\text{length } m = \text{id}' \eta$

| $S' (\text{return-spmf } (\text{Inr } ()), ()), \text{Fail}))$

$(\text{return-spmf } (\text{Fail}, \text{None}, \text{Map.empty}))$

```

| 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 $\text{length } m' = \text{id}' \eta$ **and** $\text{length } a' = \text{id}' \eta$
 $| S' (\text{return-spmf } (\text{Inl } \text{None}, (), \text{Fail}))$
 $(\text{return-spmf } (\text{Fail}, \text{None}, [m' \mapsto a']))$
if $\text{length } m' = \text{id}' \eta$ **and** $\text{length } a' = \text{id}' \eta$

private lemma *trace-eq-sim*:

shows $(\text{valid-insecQ } \langle + \rangle \text{ nlists } \text{UNIV } (\text{id}' \eta) \langle + \rangle \text{UNIV}) \vdash_R$
 $\text{RES } (\text{callee-auth-channel } \text{sim} \oplus_O \dagger\dagger \text{channel.send-oracle} \oplus_O \dagger\dagger \text{channel.recv-oracle})$
 $(\text{Inl } \text{None}, (), \text{Void})$
 \approx
 $\text{RES } (\text{lazy-channel-insec} \oplus_O \text{lazy-channel-send} \oplus_O \text{lazy-channel-recv-f}) (\text{Void},$
 $\text{None}, \text{Map.empty})$
(is $?A \vdash_R \text{RES } (?L1 \oplus_O ?L2 \oplus_O ?L3) ?SL \approx \text{RES } (?R1 \oplus_O ?R2 \oplus_O ?R3)$
 $?SR)$
 $\langle \text{proof} \rangle$ **lemma** *real-resource-wiring*: $\text{macode.res } (\text{rnd } \eta) (\text{mac } \eta) =$
 $\text{RES } (\dagger\dagger \text{insec-channel.insec-oracle} \oplus_O \text{rorc-channel-send} \oplus_O \text{rorc-channel-recv})$
 $((\text{False}, ()), \text{Map.empty}, \text{Void})$
(is $?L = ?R)$ **including** *lifting-syntax*
 $\langle \text{proof} \rangle$ **lemma** *ideal-resource-wiring*: $(\text{CNV } \text{callee } s) \models_{1_C} \triangleright \text{channel.res } \text{auth-channel.auth-oracle}$
 $=$
 $\text{RES } (\text{callee-auth-channel } \text{callee} \oplus_O \dagger\dagger \text{channel.send-oracle} \oplus_O \dagger\dagger \text{channel.recv-oracle})$
 $(s, (), \text{Void})$ **(is** $?L1 \triangleright - = ?R)$
 $\langle \text{proof} \rangle$

lemma *all-together*:

defines $\mathcal{I} \equiv \mathcal{I}\text{-uniform } (\text{Set.Collect } (\text{valid-mac-query } \eta)) (\text{insert } \text{None } (\text{Some } ' \text{vld } \eta))$
 $(\text{nlists } \text{UNIV } \eta \times \text{nlists } \text{UNIV } \eta)) \oplus_{\mathcal{I}}$
 $(\mathcal{I}\text{-uniform } (\text{vld } \eta) \text{UNIV} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform } \text{UNIV } (\text{insert } \text{None } (\text{Some } ' \text{vld } \eta)))$
assumes $\eta > 0$
and *interaction-bounded-by'* $(\lambda \cdot \text{True}) (\mathcal{A} \eta) q$
and *lossless*: $\text{plossless-gpv } \mathcal{I} (\mathcal{A} \eta)$
and *WT*: $\mathcal{I} \vdash_g \mathcal{A} \eta \checkmark$
shows
 $| \text{spm}f (\text{connect } (\mathcal{A} \eta) (\text{CNV } \text{sim } (\text{Inl } \text{None})) \models_{1_C} \triangleright \text{channel.res } \text{auth-channel.auth-oracle})$
 $\text{True} -$
 $\text{spm}f (\text{connect } (\mathcal{A} \eta) (\text{macode.res } (\text{rnd } \eta) (\text{mac } \eta))) \text{True} | \leq q / \text{real } (2 \wedge$
 $\eta)$
 $\langle \text{proof} \rangle$

end

context begin

interpretation *MAC*: $\text{macode } \text{rnd } \eta \text{ mac } \eta$ **for** η $\langle \text{proof} \rangle$

interpretation *A-CHAN*: auth-channel $\langle \text{proof} \rangle$

lemma *WT-enm*:

$X \neq \{\}$ $\implies \mathcal{I}\text{-uniform } (\text{vld } \eta) \text{UNIV}, \mathcal{I}\text{-uniform } (\text{vld } \eta) X \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform } (X \times$
 $\text{vld } \eta) \text{UNIV} \vdash_C \text{MAC.enm } \eta \checkmark$

<proof>

lemma *WT-dem*: \mathcal{I} -uniform UNIV (insert None (Some ‘ vld η)), \mathcal{I} -full $\oplus_{\mathcal{I}}$
 \mathcal{I} -uniform UNIV (insert None (Some ‘ (nlists UNIV η \times nlists UNIV η))) \vdash_C
MAC.dem η \checkmark
<proof>

lemma *valid-insec-query-of [simp]*: valid-mac-query η (insec-query-of x)
<proof>

lemma *secure-mac*:

defines \mathcal{I} -real $\equiv \lambda\eta. \mathcal{I}$ -uniform $\{x. \text{valid-mac-query } \eta \ x\}$ (insert None (Some ‘
(nlists UNIV η \times nlists UNIV η)))
and \mathcal{I} -ideal $\equiv \lambda\eta. \mathcal{I}$ -uniform UNIV (insert None (Some ‘ nlists UNIV η))
and \mathcal{I} -common $\equiv \lambda\eta. \mathcal{I}$ -uniform (vld η) UNIV $\oplus_{\mathcal{I}}$ \mathcal{I} -uniform UNIV (insert
None (Some ‘ vld η))
shows
constructive-security MAC.res ($\lambda.$ A-CHAN.res) ($\lambda.$ CNV sim (Inl None))
 \mathcal{I} -real \mathcal{I} -ideal \mathcal{I} -common ($\lambda.$ enat q) True ($\lambda.$ insec-auth-wiring)
<proof>

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 *<proof>*

interpretation *MAC*: macode rnd η mac η **for** η *<proof>*

interpretation *AUTH*: auth-channel *<proof>*

interpretation *CIPHER*: cipher key η enc η dec η **for** η *<proof>*

interpretation *SEC*: sec-channel *<proof>*

lemma *plossless-enc [plossless-intro]*:

plossless-converter (\mathcal{I} -uniform (nlists UNIV η) UNIV) (\mathcal{I} -uniform UNIV (nlists
UNIV η) $\oplus_{\mathcal{I}}$ \mathcal{I} -uniform (nlists UNIV η) UNIV) (CIPHER.enc η)
<proof>

lemma *plossless-dec [plossless-intro]*:

plossless-converter (\mathcal{I} -uniform UNIV (insert None (Some ‘ nlists UNIV η)))
(\mathcal{I} -uniform UNIV (nlists UNIV η) $\oplus_{\mathcal{I}}$ \mathcal{I} -uniform UNIV (insert None (Some ‘

$nlists\ UNIV\ \eta)))\ (CIPHER.dec\ \eta)$
 $\langle proof \rangle$

lemma *callee-invariant-on-key-oracle*:

callee-invariant-on
 $(CIPHER.KEY.key-oracle\ \eta\ \oplus_O\ CIPHER.KEY.key-oracle\ \eta)$
 $(\lambda x. case\ x\ of\ None\ \Rightarrow\ True\ |\ Some\ x'\ \Rightarrow\ length\ x' = \eta)$
 $(\mathcal{I}\text{-uniform}\ UNIV\ (nlists\ UNIV\ \eta)\ \oplus_{\mathcal{I}}\ \mathcal{I}\text{-full})$
 $\langle proof \rangle$

interpretation *key: callee-invariant-on*

$CIPHER.KEY.key-oracle\ \eta\ \oplus_O\ CIPHER.KEY.key-oracle\ \eta$
 $\lambda x. case\ x\ of\ None\ \Rightarrow\ True\ |\ Some\ x'\ \Rightarrow\ length\ x' = \eta$
 $\mathcal{I}\text{-uniform}\ UNIV\ (nlists\ UNIV\ \eta)\ \oplus_{\mathcal{I}}\ \mathcal{I}\text{-full}\ \mathbf{for}\ \eta$
 $\langle proof \rangle$

lemma *WT-enc* [*WT-intro*]: $\mathcal{I}\text{-uniform}\ (nlists\ UNIV\ \eta)\ UNIV,$
 $\mathcal{I}\text{-uniform}\ UNIV\ (nlists\ UNIV\ \eta)\ \oplus_{\mathcal{I}}\ \mathcal{I}\text{-uniform}\ (vld\ \eta)\ UNIV\ \vdash_C\ CIPHER.enc$
 $\eta\ \checkmark$
 $\langle proof \rangle$

lemma *WT-dec* [*WT-intro*]: $\mathcal{I}\text{-uniform}\ UNIV\ (insert\ None\ (Some\ 'nlists\ UNIV\ \eta)),$
 $\mathcal{I}\text{-uniform}\ UNIV\ (nlists\ UNIV\ \eta)\ \oplus_{\mathcal{I}}\ \mathcal{I}\text{-uniform}\ UNIV\ (insert\ None\ (Some\ 'vld\ \eta))\ \vdash_C$
 $CIPHER.dec\ \eta\ \checkmark$
 $\langle proof \rangle$

lemma *bound-enc* [*interaction-bound*]: *interaction-any-bounded-converter* $(CIPHER.enc\ \eta)$ *(enat 2)*
 $\langle proof \rangle$

lemma *bound-dec* [*interaction-bound*]: *interaction-any-bounded-converter* $(CIPHER.dec\ \eta)$ *(enat 2)*
 $\langle proof \rangle$

theorem *mac-otp*:

defines $\mathcal{I}\text{-real} \equiv \lambda\eta. \mathcal{I}\text{-uniform}\ \{x. valid\text{-mac}\text{-query}\ \eta\ x\}\ UNIV$
and $\mathcal{I}\text{-ideal} \equiv \lambda\cdot. \mathcal{I}\text{-full}$
and $\mathcal{I}\text{-common} \equiv \lambda\eta. \mathcal{I}\text{-uniform}\ (vld\ \eta)\ UNIV\ \oplus_{\mathcal{I}}\ \mathcal{I}\text{-full}$

shows

constructive-security
 $(\lambda\eta. 1_C\ |=\ (CIPHER.enc\ \eta\ |=\ CIPHER.dec\ \eta)\ \odot\ parallel\text{-wiring}\ \triangleright$
 $parallel\text{-resource1}\text{-wiring}\ \triangleright$
 $CIPHER.KEY.res\ \eta\ ||$
 $(1_C\ |=\ MAC.enm\ \eta\ |=\ MAC.dem\ \eta\ \triangleright$
 $1_C\ |=\ parallel\text{-wiring}\ \triangleright$
 $parallel\text{-resource1}\text{-wiring}\ \triangleright\ MAC.RO.res\ \eta\ ||\ INSEC.res))$
 $(\lambda\cdot. SEC.res)$


```

  (λη. CNV Message-Authentication-Code.sim (Inl None) ⊙ CNV One-Time-Pad.sim
  None)
  (λη. I-uniform (Set.Collect (valid-mac-query η)) (insert None (Some ‘ (nlists
  UNIV η × nlists UNIV η))))
  (λη. I-uniform UNIV {None, Some η})
  (λη. I-uniform (nlists UNIV η) UNIV ⊕I I-uniform UNIV (insert None
  (Some ‘ nlists UNIV η)))
  (λ-. enat q) True (λη. (id, map-option length) ◦w (insec-query-of, map-option
  snd))
  ⟨proof⟩

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.