

Constructive Cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar

September 13, 2023

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*)

$\implies rel-resource\ A\ B\ res1\ res2$

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

assumes *X res1 res2*

and $\bigwedge res1\ res2. X\ res1\ res2 \implies$

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* $\implies rel-fun\ A\ (rel-spmf\ (rel-prod\ B\ (rel-resource\ A$
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*:

$rel\text{-}resource\ A\ B\ (map\text{-}resource'\ f\ res1)\ res2 = rel\text{-}resource\ A\ (\lambda x. B\ (f\ x))\ res1$
 $res2$
 (is ?lhs = ?rhs)
 <proof>

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

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

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

lemma *rel-resource-pos-distr*:

$rel\text{-}resource\ A\ B\ OO\ rel\text{-}resource\ A'\ B' \leq rel\text{-}resource\ (A\ OO\ A')\ (B\ OO\ B')$
 <proof>

lemma *left-unique-rel-resource*:

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

lemma *right-unique-rel-resource*:

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

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

$\llbracket bi\text{-}total\ A; bi\text{-}unique\ B \rrbracket \implies bi\text{-}unique\ (rel\text{-}resource\ A\ B)$
 <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**
 $rel\text{-}witness\text{-}resource\ A\ A'\ B = corec\text{-}resource\ (\lambda(res1, res2).$
 $map\text{-}spmf\ (map\text{-}prod\ id\ Inr \circ rel\text{-}witness\text{-}prod) \circ$
 $rel\text{-}witness\text{-}spmf\ (rel\text{-}prod\ B\ (rel\text{-}resource\ (A\ OO\ A')\ B)) \circ$
 $rel\text{-}witness\text{-}fun\ A\ A'\ (run\text{-}resource\ res1, run\text{-}resource\ res2))$

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

$run\text{-}resource\ (rel\text{-}witness\text{-}resource\ A\ A'\ B\ (res1, res2)) =$
 $map\text{-}spmf\ (map\text{-}prod\ id\ (rel\text{-}witness\text{-}resource\ A\ A'\ B) \circ rel\text{-}witness\text{-}prod) \circ$
 $rel\text{-}witness\text{-}spmf\ (rel\text{-}prod\ B\ (rel\text{-}resource\ (A\ OO\ A')\ B)) \circ$
 $rel\text{-}witness\text{-}fun\ A\ A'\ (run\text{-}resource\ res1, run\text{-}resource\ res2)$
 <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 ($\lambda b\ (b', c). b = b' \wedge B\ b'\ c$) res*
(rel-witness-resource A A' B (res, res')) (is ?thesis1)

and *rel-witness-resource2: rel-resource A' ($\lambda(b, c'). c = c' \wedge B\ b\ c'$) (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*]:
 $((A \implies \text{rel-spmf} (\text{rel-prod } B (\text{rel-resource } A B))) \implies C) \implies \text{rel-resource } A B \implies C$
case-resource case-resource
 $\langle \text{proof} \rangle$

end

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

1.4 Losslessness

coinductive *lossless-resource* :: ('a, 'b) $\mathcal{I} \Rightarrow$ ('a, 'b) *resource* \Rightarrow *bool*
for \mathcal{I} **where**
lossless-resourceI: *lossless-resource* \mathcal{I} *res* **if**
 $\bigwedge a. a \in \text{outs-}\mathcal{I} \mathcal{I} \implies \text{lossless-spmf} (\text{run-resource } res a)$
 $\bigwedge a b \text{ res}'. \llbracket a \in \text{outs-}\mathcal{I} \mathcal{I}; (b, \text{res}') \in \text{set-spmf} (\text{run-resource } res a) \rrbracket \implies \text{lossless-resource } \mathcal{I} \text{res}'$

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 $\bigwedge \text{res } a. \llbracket X \text{ res}; a \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{lossless-spmf} (\text{run-resource } res a) \wedge (\forall (b, \text{res}') \in \text{set-spmf} (\text{run-resource } res a). X \text{ res}' \vee \text{lossless-resource } \mathcal{I} \text{res}')$
shows *lossless-resource* \mathcal{I} *res*
 $\langle \text{proof} \rangle$

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

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

1.5 Operations

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

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

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

end

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

$((S \text{====>} A \text{====>} rel\text{-spmf} (rel\text{-prod } B \ S)) \text{====>} S \text{====>} rel\text{-resource } A \ B)$ $resource\text{-of}\text{-oracle} \ resource\text{-of}\text{-oracle}$
<proof>

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

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

for $s :: 's$

<proof>

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

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

and $I \ s$

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

<proof>

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

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

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

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

<proof>

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

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

<proof>

end

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

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

<proof>

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{-spmf} \ (map\text{-prod} \ Inl \ (\lambda res1'. \ parallel\text{-resource} \ res1' \$

$res2))$ ($run-resource\ res1\ a$)
 $| Inr\ c \Rightarrow map\text{-}spmf\ (map\text{-}prod\ Inr\ (\lambda res2'.\ parallel\text{-}resource\ res1\ res2'))$
 $(run-resource\ res2\ c)$

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

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*:
 $resource-of-oracle\ (parallel-oracle\ oracle1\ oracle2)\ (s1,\ s2) =$
 $parallel-resource\ (resource-of-oracle\ oracle1\ s1)\ (resource-of-oracle\ oracle2\ s2)$
 $\langle proof \rangle$

lemma *parallel-resource-assoc*: — There's still an ugly map operation in there to rebalance the interface trees, but well...
 $parallel-resource\ (parallel-resource\ res1\ res2)\ res3 =$
 $map-resource\ rsuml\ lsumr\ (parallel-resource\ res1\ (parallel-resource\ res2\ res3))$
 $\langle proof \rangle$

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

1.6 Well-typing

coinductive *WT-resource* :: $('a,\ 'b)\ \mathcal{I} \Rightarrow ('a,\ 'b)\ resource \Rightarrow bool$ ($- \ / \vdash res - \ \checkmark$
 $[100,\ 0]\ 99)$
for \mathcal{I} **where**
 $WT-resourceI: \mathcal{I} \vdash res\ \checkmark$
if $\bigwedge q\ r\ res'. \llbracket q \in outs\text{-}\mathcal{I}\ \mathcal{I}; (r,\ res') \in set\text{-}spmf\ (run-resource\ res\ q) \rrbracket \Longrightarrow r \in$
 $responses\text{-}\mathcal{I}\ \mathcal{I}\ q \wedge \mathcal{I} \vdash res'\ \checkmark$

lemma *WT-resource-coinduct* [*consumes 1, case-names WT-resource, case-conclusion*
 $WT-resource\ response\ WT-resource,$ *coinduct pred: WT-resource*]:
assumes $X\ res$
and $\bigwedge res\ q\ r\ res'. \llbracket X\ res; q \in outs\text{-}\mathcal{I}\ \mathcal{I}; (r,\ res') \in set\text{-}spmf\ (run-resource\ res$
 $q) \rrbracket$
 $\Longrightarrow r \in responses\text{-}\mathcal{I}\ \mathcal{I}\ q \wedge (X\ res' \vee \mathcal{I} \vdash res'\ \checkmark)$
shows $\mathcal{I} \vdash res\ \checkmark$
 $\langle proof \rangle$

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 simp = outsp-converter.simps[to-set]
end

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

lemma resultsp-converter-results-converter-eq [pred-set-conv]:
  Converter.resultsp-converter  $\mathcal{I} \mathcal{I}' x = (\lambda conv. x \in results-converter \mathcal{I} \mathcal{I}' conv)$ 
  ⟨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 simp = resultsp-converter.simps[to-set]
end

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

```

2.4 Relator

```

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

```

```

lemma rel-converter-coinduct [consumes 1, case-names rel-converter, coinduct pred:
rel-converter]:
  assumes X conv1 conv2
  and  $\bigwedge conv1 conv2. X conv1 conv2 \implies$ 
    rel-fun A (rel-gpv'' (rel-prod B ( $\lambda conv1 conv2. X conv1 conv2 \vee rel-converter$ 
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'* $(\lambda(b, c') \ c. c = c' \wedge B \ b \ c') \ (\lambda(c,$
 $d') \ d. d = d' \wedge C \ c \ d') \ R' \ (\text{rel-witness-converter } A \ A' \ B \ C \ R \ R' \ (\text{conv}, \text{conv}'))$
 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 $\text{rel-converter } (A \text{ OO } A') (B \text{ OO } B') (C \text{ OO } C') (R \text{ OO } R') \leq \text{rel-converter}$
 $A B C R \text{ OO } \text{rel-converter } A' B' C' R'$
 ⟨*proof*⟩

lemma *left-total-rel-converter*:

[[*left-unique* A ; *right-total* A ; *left-total* B ; *left-total* C ; *left-unique* R ; *right-total* R]]
 $\implies \text{left-total } (\text{rel-converter } A B C R)$
 ⟨*proof*⟩

lemma *right-total-rel-converter*:

[[*right-unique* A ; *left-total* A ; *right-total* B ; *right-total* C ; *right-unique* R ; *left-total* R]]
 $\implies \text{right-total } (\text{rel-converter } A B C R)$
 ⟨*proof*⟩

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

[[*bi-total* A ; *bi-unique* A ; *bi-total* B ; *bi-total* C ; *bi-total* R ; *bi-unique* R]]
 $\implies \text{bi-total } (\text{rel-converter } A B C R)$
 ⟨*proof*⟩

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**

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

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

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

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

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

⟨*proof*⟩

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)

⟨*proof*⟩

lemma *rel-converter-Grp*:

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

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

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

(*map-converter* $f g h k$)

(**is** ?lhs = ?rhs)

including *lifting-syntax*

<proof>

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*
<proof>

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
<proof>

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

$((S \text{====>} A \text{====>} \text{rel-gpv}''(\text{rel-prod } B (\text{rel-sum } (\text{rel-converter } A B C R) S)) C R) \text{====>} S \text{====>} \text{rel-converter } A B C R)$
corec-converter corec-converter
<proof>

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
<proof>

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'
<proof>

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
<proof>

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 $\mathcal{I} \mathcal{I}'$
 ⟨proof⟩

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

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) $\mathcal{I}, \text{map-}\mathcal{I} f' g' \mathcal{I}' \vdash_C$ conv \checkmark
and f : inj f **and** g : surj g
 ⟨proof⟩

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
 ⟨proof⟩

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

lemma *plossless-converter-bot1* [*simp*]: *plossless-converter bot* \mathcal{I} conv
 ⟨proof⟩

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
 ⟨proof⟩

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

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}'
 ⟨proof⟩

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*
 ⟨proof⟩

lemma *map-converter-of-callee*:

map-converter f g h k (*converter-of-callee* *callee* s) =
converter-of-callee (*map-fun* id (*map-fun* f (*map-gpv'* (*map-prod* g id) h k))
callee) s

⟨proof⟩

lemma *WT-converter-of-callee*:

assumes *WT*: $\bigwedge s q. q \in \text{outs-}\mathcal{I} \mathcal{I} \implies \mathcal{I}' \vdash g \text{ callee } s q \checkmark$

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

shows $\mathcal{I}, \mathcal{I}' \vdash_C \text{converter-of-callee } \text{callee } s \checkmark$

⟨proof⟩

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

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

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

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]:

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

primcorec id-converter :: ('a, 'b, 'a, 'b) converter where

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

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

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

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

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

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

lemma conv-callee-plus-id-right: converter-of-callee (plus-intercept callee 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*
 $\langle \text{proof} \rangle$

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

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

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

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

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

lemma *plossless-fail-converterI* [*plossless-intro*]: *plossless-converter bot* \mathcal{I}' *fail-converter*
 $\langle \text{proof} \rangle$

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

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
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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

primcorec *parallel-converter2*
:: ('a, 'b, 'out, 'in) *converter* \Rightarrow ('c, 'd, 'out', 'in') *converter* \Rightarrow ('a + 'c, 'b + 'd,
'out + 'out', 'in + 'in') *converter*
where
run-converter (parallel-converter2 conv1 conv2) = (λ ac. *case ac of*
Inl a \Rightarrow *map-gpv (map-prod Inl (λ conv1'. parallel-converter2 conv1' conv2))*)

id ($left\text{-}gpv$ ($run\text{-}converter$ $conv1$ a))
 $|$ Inr $b \Rightarrow map\text{-}gpv$ ($map\text{-}prod$ Inr ($\lambda conv2'. parallel\text{-}converter2$ $conv1$ $conv2'$))
 id ($right\text{-}gpv$ ($run\text{-}converter$ $conv2$ b))

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

lemma *map-converter-parallel-converter2*:
 $map\text{-}converter$ ($map\text{-}sum$ f f') ($map\text{-}sum$ g g') ($map\text{-}sum$ h h') ($map\text{-}sum$ k k')
 $(parallel\text{-}converter2$ $conv1$ $conv2)$ =
 $parallel\text{-}converter2$ ($map\text{-}converter$ f g h k $conv1$) ($map\text{-}converter$ f' g' h' k'
 $conv2$)
 $\langle proof \rangle$

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

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

lemma *parallel-converter2-map1-out*:
 $parallel\text{-}converter2$ ($map\text{-}converter$ f g h k $conv1$) $conv2$ =
 $map\text{-}converter$ ($map\text{-}sum$ f id) ($map\text{-}sum$ g id) ($map\text{-}sum$ h id) ($map\text{-}sum$ k id)
 $(parallel\text{-}converter2$ $conv1$ $conv2)$
 $\langle proof \rangle$

lemma *parallel-converter2-map2-out*:
 $parallel\text{-}converter2$ $conv1$ ($map\text{-}converter$ f g h k $conv2$) =
 $map\text{-}converter$ ($map\text{-}sum$ id f) ($map\text{-}sum$ id g) ($map\text{-}sum$ id h) ($map\text{-}sum$ id k)
 $(parallel\text{-}converter2$ $conv1$ $conv2)$
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

<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*) = $(\lambda a. \text{if } a \in A \text{ then}$

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

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

$\text{else } \text{Fail}$)

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* [*plossless-intro*]:

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

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

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

shows *lossless-resource* \mathcal{I} (*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) = (\a.
map-gpv (\((b, conv1'), conv2')). (b, comp-converter conv1' conv2')) id (inline
run-converter (run-converter conv1 a) conv2))

lemma *comp-converter-parametric* [transfer-rule]: **includes** *lifting-syntax* **shows**
(rel-converter A B C R ==>> rel-converter C R C' R' ==>> rel-converter A
B C' R')
comp-converter comp-converter
<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* ⇒ *enat*
 ⇒ *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 $\bigwedge x. \text{consider}' (Inl\ x) \Longrightarrow \text{consider } x$
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 $\bigwedge x. \text{consider}' (Inr\ x) \Longrightarrow \text{consider } x$
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**

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

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

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

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

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

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

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*: $\text{res} = \text{res}' \implies A \vdash_R \text{res} \sim \text{res}'$

<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*: $\text{UNIV} \vdash_R \text{res} \sim \text{res}' \iff \text{res} = \text{res}'$

<proof>

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

$\langle \text{proof} \rangle$

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

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

includes *lifting-syntax*

fixes S

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

and $S: S \text{ } s1 \text{ } s2$

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

$\langle \text{proof} \rangle$

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

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

$eq\mathcal{I}\text{-}gpvI$: $eq\mathcal{I}\text{-}gpv A \mathcal{I} gpv gpv'$
if $rel\text{-}spmf (eq\mathcal{I}\text{-}generat A \mathcal{I} (eq\mathcal{I}\text{-}gpv A \mathcal{I})) (the\text{-}gpv gpv) (the\text{-}gpv gpv')$

lemma $eq\mathcal{I}\text{-}gpv\text{-}coinduct$ [*consumes 1, case-names eq \mathcal{I} -gpv, coinduct pred: eq \mathcal{I} -gpv*]:
assumes $X gpv gpv'$
and $\bigwedge gpv gpv'. X gpv gpv'$
 $\implies rel\text{-}spmf (eq\mathcal{I}\text{-}generat A \mathcal{I} (\lambda gpv gpv'. X gpv gpv' \vee eq\mathcal{I}\text{-}gpv A \mathcal{I} gpv gpv')) (the\text{-}gpv gpv) (the\text{-}gpv gpv')$
shows $eq\mathcal{I}\text{-}gpv A \mathcal{I} gpv gpv'$
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

[[*eq- \mathcal{I} -gpv A \mathcal{I} gpv gpv'*; *I s*]] $\implies rel\text{-}spmf (rel\text{-}prod A (eq\text{-}onp I)) (exec\text{-}gpv callee gpv s) (exec\text{-}gpv callee gpv' s)$
 ⟨*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 $rel\text{-}spmf (rel\text{-}sum (rel\text{-}prod A S) (\lambda (q, rpv1, rpv2) (q', rpv1', rpv2'). q = q' \wedge q' \in outs\text{-}\mathcal{I} \mathcal{I}' \wedge (\exists q'' \in outs\text{-}\mathcal{I} \mathcal{I}. (\forall r \in responses\text{-}\mathcal{I} \mathcal{I}' q'. eq\text{-}\mathcal{I}\text{-}gpv (rel\text{-}prod (eq\text{-}onp (\lambda r'. r' \in responses\text{-}\mathcal{I} \mathcal{I}'))$

$\mathcal{I} q'')) S) \mathcal{I}' (rpv1 r) (rpv1' r)) \wedge$
 $(\forall r' \in \text{responses-}\mathcal{I} \mathcal{I} q''. \text{eq-}\mathcal{I}\text{-gpv } A \mathcal{I} (rpv2 r') (rpv2' r'))))$
 $(\text{inline1 callee1 gpv1 s1}) (\text{inline1 callee2 gpv2 s2})$
 $\langle \text{proof} \rangle$

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

assumes $S: S s1 s2$

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

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

$\langle \text{proof} \rangle$

end

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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

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

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

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

$\langle \text{proof} \rangle$

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 $\mathcal{I}, \mathcal{I}' \vdash_C conv \sim conv' q \in outs\text{-}\mathcal{I} \mathcal{I}$
 $\langle proof \rangle$

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

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

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

lemma $eq\text{-}\mathcal{I}\text{-}converter\text{-}mono$:
assumes *: $\mathcal{I}1, \mathcal{I}2 \vdash_C conv \sim conv'$
and le : $\mathcal{I}1' \leq \mathcal{I}1 \mathcal{I}2 \leq \mathcal{I}2'$
shows $\mathcal{I}1', \mathcal{I}2' \vdash_C conv \sim conv'$
 $\langle proof \rangle$

lemma $eq\text{-}\mathcal{I}\text{-}converter\text{-}eq$: $conv1 = conv2$ **if** $\mathcal{I}\text{-}full, \mathcal{I}\text{-}full \vdash_C conv1 \sim conv2$
 $\langle proof \rangle$

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

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

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

lemma $eq\text{-}\mathcal{I}\text{-}comp\text{-}cong$:
 $\llbracket \mathcal{I}1, \mathcal{I}2 \vdash_C conv1 \sim conv1'; \mathcal{I}2, \mathcal{I}3 \vdash_C conv2 \sim conv2' \rrbracket$
 $\implies \mathcal{I}1, \mathcal{I}3 \vdash_C comp\text{-}converter conv1 conv2 \sim comp\text{-}converter conv1' conv2'$
 $\langle proof \rangle$

lemma $comp\text{-}converter\text{-}cong$: $comp\text{-}converter conv1 conv2 = comp\text{-}converter conv1' conv2'$
if $\mathcal{I}\text{-}full, \mathcal{I} \vdash_C conv1 \sim conv1' \mathcal{I}, \mathcal{I}\text{-}full \vdash_C conv2 \sim conv2'$
 $\langle proof \rangle$

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*:

$\llbracket \mathcal{I}1, \mathcal{I}1' \vdash_C \text{conv1} \sim \text{conv1}' ; \mathcal{I}2, \mathcal{I}2' \vdash_C \text{conv2} \sim \text{conv2}' \rrbracket$
 $\implies \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C \text{parallel-converter2 conv1 conv2} \sim \text{parallel-converter2}$
 $\text{conv1}' \text{ conv2}'$
<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 $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv1} \sim \text{conv2}$ **and** $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv1} \checkmark$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv2} \checkmark$
<proof>

lemma *eq-I-converterD-WT*:

assumes $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv1} \sim \text{conv2}$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv1} \checkmark \longleftrightarrow \mathcal{I}, \mathcal{I}' \vdash_C \text{conv2} \checkmark$
<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* = *bind-spmf p* (λ s. *map-spmf fst* (*callee s x*))
| *trace-callee callee p* ((a, b) # *xs*) *x* =
trace-callee callee (*cond-spmf-fst* (*bind-spmf p* (λ s. *callee s a*)) b) *xs x*

definition *trace-callee-eq* :: ('a, 'b, 's1) *callee* \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*. *set xs* \subseteq A \times UNIV \longrightarrow (\forall *x* \in A. *trace-callee callee1 p xs x* = *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 *trace-callee-eq callee1 callee2 A*
(*return-spmf s1*) (*return-spmf s2*)

lemma *trace-callee-eqI*:

assumes \bigwedge *xs x*. [] *set xs* \subseteq A \times UNIV; *x* \in A []
 \implies *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*
shows *trace-callee-eq callee1 callee2 A p q*
 \langle *proof* \rangle

lemma *trace-callee-eqD*:

assumes *trace-callee-eq callee1 callee2 A p q*
and *set xs* \subseteq A \times UNIV *x* \in A
shows *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*
 \langle *proof* \rangle

lemma *cond-spmf-fst-None* [*simp*]: *cond-spmf-fst* (*return-pmf None*) *x* = *return-pmf None*

\langle *proof* \rangle

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

trace-callee callee (*return-pmf None*) *xs x* = *return-pmf None*
 \langle *proof* \rangle

proposition *trace'-eqI-sim*:

fixes *callee1* :: ('a, 'b, 's1) *callee* **and** *callee2* :: ('a, 'b, 's2) *callee*
assumes *start*: S *p q*
and *step*: \bigwedge *p q a*. [] S *p q*; *a* \in A [] \implies
bind-spmf p (λ s. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ s. *map-spmf fst*
(*callee2 s a*))
and *sim*: \bigwedge *p q a res b s'*. [] S *p q*; *a* \in A; *res* \in *set-spmf q*; (*b*, *s'*) \in *set-spmf*

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

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

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

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

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*: $\mathcal{I} \vdash g \text{ d } \checkmark$

and *out*: $\text{outs-gpv } \mathcal{I} \text{ d } \subseteq A$

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 *connect distinguisher res* = *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} (*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* (*absorb gpv conv*) (*bound1 * 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-rasso_C* [*plossless-intro*]: *plossless-converter* ($\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)$)
 $((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3)$ *rasso_C*
 ⟨*proof*⟩

lemma *plossless-lasso_C* [*plossless-intro*]: *plossless-converter* $((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}}$
 $\mathcal{I}3)$ $(\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3))$ *lasso_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-lasso_C* [*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-lasso_C*
 ⟨*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-lasso_C*: $((conv1 \mid= conv2) \mid= conv3) \odot$ *lasso_C* $=$ *lasso_C* \odot
 $(conv1 \mid= conv2 \mid= conv3)$
 ⟨*proof*⟩

lemmas *comp-lasso_C'* $=$ *comp-converter-eqs*[*OF comp-lasso_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-rasso_C*:
 $(conv1 \mid= conv2 \mid= conv3) \odot$ *rasso_C* $=$ *rasso_C* \odot $((conv1 \mid= conv2) \mid= conv3)$
 ⟨*proof*⟩

lemmas *comp-rasso_C'* $=$ *comp-converter-eqs*[*OF comp-rasso_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*: $(conv1 \models conv2) \odot swap_C = swap_C \odot (conv2 \models conv1)$
 ⟨proof⟩

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

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

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

lemma *comp-parallel-wiring*:
 $((C1 \models C2) \models (C3 \models C4)) \odot parallel-wiring = parallel-wiring \odot ((C1 \models C3)$
 $\models (C2 \models C4))$
 ⟨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 $((I1 \oplus_I I2) \oplus_I (I3 \oplus_I I4)) ((I1 \oplus_I I3) \oplus_I (I2 \oplus_I I4))$
parallel-wiring
 ⟨proof⟩

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

lemma *run-converter-rassoel* [*simp*]:
run-converter *rassoel_C* $x = Pause (lsumr \ x) (\lambda x. Done (rsuml \ x, rassoel_C))$
 ⟨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:

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

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

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

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

hide-fact wiring

lemma wiringD :

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

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

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

<proof>

named-theorems wiring-intro introduction rules for wiring

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

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

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

<proof>

lemma $\text{attach-wiring-resource-of-oracle}$:

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

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

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

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

<proof>

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

<proof>

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

<proof>

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

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

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* :: $(\text{'a}, \text{'b}, \text{'c}, \text{'d}) \text{ wiring} \Rightarrow (\text{'c}, \text{'d}, \text{'e}, \text{'f}) \text{ wiring} \Rightarrow (\text{'a}, \text{'b}, \text{'e}, \text{'f}) \text{ wiring}$ (**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*
 :: $(\text{'a}, \text{'b}, \text{'c}, \text{'d}) \text{ wiring} \Rightarrow (\text{'a}', \text{'b}', \text{'c}', \text{'d}') \text{ wiring}$
 $\Rightarrow (\text{'a} + \text{'a}', \text{'b} + \text{'b}', \text{'c} + \text{'c}', \text{'d} + \text{'d}') \text{ wiring}$ (**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 $\text{wiring } (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) (\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2') (\text{conv1 } |_{=} \text{conv2}) (fg \ |_w \ fg')$
 ⟨proof⟩

lemma *apply-parallel2* [*simp*]:
 $\text{apply-wiring } (fg \ |_w \ fg') (\text{res1} \oplus_{\mathcal{O}} \text{res2}) = (\text{apply-wiring } fg \ \text{res1} \oplus_{\mathcal{O}} \text{apply-wiring } fg' \ \text{res2})$
 ⟨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* :: $((\text{'a} + \text{'b}) + \text{'c}, (\text{'d} + \text{'e}) + \text{'f}, \text{'a} + (\text{'b} + \text{'c}), \text{'d} + (\text{'e} + \text{'f})) \text{ wiring}$

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*]:
 $wiring ((\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) \oplus_{\mathcal{I}} \mathcal{I}3) (\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)) lassocr_C lassocr_w$
 $\langle proof \rangle$

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

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

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

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

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

end

7 Security

theory *Constructive-Cryptography* **imports**

Wiring

begin

definition $advantage \mathcal{A} res1 res2 = |spmf (connect \mathcal{A} res1) True - smpf (connect \mathcal{A} res2) True|$

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

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

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

theorem *parallel-constructive-security*:

assumes *constructive-security* $\text{real1} \text{ideal1} \text{sim1} \mathcal{I}\text{-real1} \mathcal{I}\text{-inner1} \mathcal{I}\text{-common1} \text{bound1} \text{lossless1} w1$
assumes *constructive-security* $\text{real2} \text{ideal2} \text{sim2} \mathcal{I}\text{-real2} \mathcal{I}\text{-inner2} \mathcal{I}\text{-common2} \text{bound2} \text{lossless2} w2$

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* $(\lambda \eta. \text{parallel-wiring} \triangleright \text{real1} \eta \parallel \text{real2} \eta) (\lambda \eta. \text{parallel-wiring} \triangleright \text{ideal1} \eta \parallel \text{ideal2} \eta) (\lambda \eta. \text{sim1} \eta \mid = \text{sim2} \eta)$
 $(\lambda \eta. \mathcal{I}\text{-real1} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2} \eta) (\lambda \eta. \mathcal{I}\text{-inner1} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2} \eta) (\lambda \eta. \mathcal{I}\text{-common1} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2} \eta)$
 $(\lambda \eta. \text{min} (\text{bound1} \eta) (\text{bound2} \eta)) (\text{lossless1} \vee \text{lossless2}) (\lambda \eta. w1 \eta \mid_w w2 \eta)$
 (proof)

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* $(\lambda \eta. \text{parallel-wiring} \triangleright \text{res} \eta \parallel \text{real-resource} \eta)$
 $(\lambda \eta. \text{parallel-wiring} \triangleright (\text{res} \eta \parallel \text{ideal-resource} \eta)) (\lambda \eta. \text{parallel-converter2} \text{id-converter} (\text{sim} \eta))$
 $(\lambda \eta. \mathcal{I}\text{-res} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real} \eta) (\lambda \eta. \mathcal{I}\text{-res} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-ideal} \eta) (\lambda \eta. \mathcal{I}\text{-common}' \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \eta) \text{bound} \text{lossless} (\lambda \eta. (id, id) \mid_w w \eta)$
 (proof)

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* $(\lambda \eta. \text{parallel-wiring} \triangleright \text{real-resource} \eta \parallel \text{res} \eta)$
 $(\lambda \eta. \text{parallel-wiring} \triangleright (\text{ideal-resource} \eta \parallel \text{res} \eta)) (\lambda \eta. \text{parallel-converter2} (\text{sim} \eta) \text{id-converter})$
 $(\lambda \eta. \mathcal{I}\text{-real} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-res} \eta) (\lambda \eta. \mathcal{I}\text{-ideal} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-res} \eta) (\lambda \eta. \mathcal{I}\text{-common} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta) \text{bound} \text{lossless} (\lambda \eta. w \eta \mid_w (id, id))$
 (proof)

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

```

 $\eta$ )
shows constructive-security ( $\lambda\eta. \text{parallel-resource1-wiring} \triangleright \text{res } \eta \parallel \text{real-resource}$ 
 $\eta$ )
  ( $\lambda\eta. \text{parallel-resource1-wiring} \triangleright \text{res } \eta \parallel \text{ideal-resource } \eta$ ) sim
   $\mathcal{I}$ -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 insecure-query = 'a option aquery
type-synonym auth-query = unit aquery

consts Forward :: 'a aquery
abbreviation Forward-auth :: auth-query where Forward-auth  $\equiv$  ForwardOrEdit
()
abbreviation Forward-insec :: 'a insecure-query where Forward-insec  $\equiv$  ForwardOrEdit None
abbreviation Edit :: 'a  $\Rightarrow$  'a insecure-query where Edit m  $\equiv$  ForwardOrEdit (Some m)
adhoc-overloading Forward Forward-auth
adhoc-overloading Forward Forward-insec

```

translations

```

(logic) CONST Forward <= (logic) CONST ForwardOrEdit (CONST None)
(logic) CONST Forward <= (logic) CONST ForwardOrEdit (CONST Product-Type.Unity)
(type) auth-query <= (type) unit aquery
(type) 'a insecure-query <= (type) 'a option aquery

```

8.3.1 Generic channel

```

locale channel =
  fixes side-oracle :: ('m cstate, 'a, 'b option) oracle'
begin

fun send-oracle :: ('m cstate, 'm, unit) oracle' where
  send-oracle Void m = return-spmf ((), Store m)
| send-oracle s m = return-spmf ((), s)

fun recv-oracle :: ('m cstate, unit, 'm option) oracle' where
  recv-oracle (Collect m) () = return-spmf (Some m, Fail)
| recv-oracle s () = return-spmf (None, s)

definition res :: ('a + 'm + unit, 'b option + unit + 'm option) resource where
  res  $\equiv$  RES (side-oracle  $\oplus_O$  send-oracle  $\oplus_O$  recv-oracle) Void

end

```

8.3.2 Insecure channel

```

locale insecure-channel
begin

fun insecure-oracle :: ('m cstate, 'm insecure-query, 'm option) oracle' where
  insecure-oracle Void (Edit m') = return-spmf (None, Collect m')
| insecure-oracle (Store m) (Edit m') = return-spmf (None, Collect m')
| insecure-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)* $auth\text{-}query\text{-}of :: auth\text{-}query \Rightarrow auth\text{-}query$
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-channel + KEY: key key-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 $enc :: ('m, unit, unit + 'c, 'k + unit)\ converter\ \mathbf{where}$
 $enc \equiv CNV\ (stateless\text{-}callee\ (\lambda m. do\ \{$
 $\quad k \leftarrow Pause\ (Inl\ ())\ Done;$
 $\quad c \leftarrow lift\text{-}spmf\ (enc\text{-}alg\ (projl\ k)\ m);$
 $\quad (- :: 'k + unit) \leftarrow Pause\ (Inr\ c)\ Done;$
 $\quad Done\ (())$
 $\quad \})\ ()$

definition $dec :: (unit, 'm\ option, unit + unit, 'k + 'c\ option)\ converter\ \mathbf{where}$
 $dec \equiv CNV\ (stateless\text{-}callee\ (\lambda -. Pause\ (Inr\ ()))\ (\lambda c'.$
 $\quad case\ c'\ of\ Inr\ (Some\ c) \Rightarrow (do\ \{$
 $\quad\quad k \leftarrow Pause\ (Inl\ ())\ Done;$
 $\quad\quad Done\ (dec\text{-}alg\ (projl\ k)\ c)\ \})$
 $\quad | - \Rightarrow Done\ None)$
 $\quad \})\ ()$

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

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

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

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

end

8.5 Message authentication converter

locale *macode* =
 INSEC: *insec-channel* + *RO*: *rorc range*
 for *range* :: 'r set +
 fixes *mac-alg* :: 'r \Rightarrow 'm \Rightarrow 'a *spm*f
begin

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

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

definition π^E :: (('a \times 'm) *insec-query*, ('a \times 'm) option, ('a \times 'm) *insec-query*,
('a \times 'm) option) converter (π^E) **where**
 $\pi^E \equiv 1_C$

definition *routing* \equiv ($1_C \mid =$ *lassocr*_C) \odot *swap-lassocr* \odot ($1_C \mid =$ ($1_C \mid =$ *swap-lassocr*)
 \odot *swap-lassocr*) \odot *rasso*_C

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

end

lemma *interface-wiring*:

(*cnv-addr* $\mid =$ *cnv-send* $\mid =$ *cnv-recv*) \triangleright ($1_C \mid =$ *parallel-wiring*) \triangleright *parallel-resource1-wiring*
 \triangleright
(*RES* (*res2-send* \oplus_O *res2-recv*) *res2-s* \parallel *RES* (*res1-addr* \oplus_O *res1-send* \oplus_O *res1-recv*)
res1-s)
=

$cnv-advr \models cnv-send \models cnv-recv \triangleright$
 $RES (\dagger res1-advr \oplus_O (res2-send \dagger \oplus_O \dagger res1-send) \oplus_O res2-recv \dagger \oplus_O \dagger res1-recv)$
 $(res2-s, res1-s)$
 $(is - \triangleright ?L1 \triangleright ?L2 \triangleright ?L3 = - \triangleright ?R)$
 $\langle proof \rangle$

definition id' where $id' = id$

end

9 Security of one-time-pad encryption

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

System-Construction

begin

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

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

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

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

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

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

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

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

context

fixes $\eta :: security$

begin

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

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

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

$\text{return-spmf } ((), s)\}$

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

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

lemma $\text{resources-indistinguishable}$:

shows $(\text{UNIV } \langle + \rangle \text{nlists UNIV } (\text{id}' \eta) \langle + \rangle \text{UNIV}) \vdash_R$
 $\text{RES } (\text{callee-sec-channel sim } \oplus_O \dagger \dagger \text{channel.send-oracle } \oplus_O \dagger \dagger \text{channel.recv-oracle})$
 $(\text{None} :: \text{bool list option}, (), \text{Void})$
 \approx
 $\text{RES } (\dagger \text{auth-channel.auth-oracle } \oplus_O \text{key-channel-send } \oplus_O \text{key-channel-recv})$
 $(\text{None} :: \text{bool list option}, \text{Void})$
(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 *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*
<proof>

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* $\triangleright - = ?R$)
<proof>

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)
<proof>

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))
<proof>

context begin

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

interpretation *S-CHAN*: *sec-channel* *<proof>*

lemma *one-time-pad*:

defines *I-real* $\equiv \lambda\eta. \mathcal{I}\text{-uniform UNIV (insert None (Some ' nlists UNIV } \eta))$
and *I-ideal* $\equiv \lambda\eta. \mathcal{I}\text{-uniform UNIV \{None, Some } \eta\}$
and *I-common* $\equiv \lambda\eta. \mathcal{I}\text{-uniform (nlists UNIV } \eta) \text{ UNIV } \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform UNIV}$
(*insert None (Some ' nlists UNIV } \eta)*)
shows
constructive-security2 CIPHER.res ($\lambda-. \text{S-CHAN.res}$) ($\lambda-. \text{CNV sim None}$)
I-real I-ideal I-common ($\lambda-. \infty$) *False* ($\lambda-. \text{auth-sec-wiring}$)
<proof>

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

context

fixes $\eta :: \text{security}$

begin

private definition *rorc-channel-send* :: $((\text{bool} \times \text{unit}) \times (\text{bool list} \Rightarrow \text{bool list option}) \times (\text{bool list} \times \text{bool list}) \text{ cstate}, \text{bool list}, \text{unit}) \text{ oracle}' \text{ where}$

rorc-channel-send $s \ m \equiv$ (if fst (fst s)
then return-spmf $(((), (True, ())), \text{snd } s)$
else do {
 $(r, s) \leftarrow (\text{rorc.rnd-oracle } (\text{rnd } \eta))^\dagger (\text{snd } s) \ m;$
 $a \leftarrow \text{mac } \eta \ r \ m;$
 $(-, s) \leftarrow \dagger \text{channel.send-oracle } s \ (a, m);$
 return-spmf $(((), (True, ())), s)$
})

private definition *rorc-channel-recv* :: $((\text{bool} \times \text{unit}) \times (\text{bool list} \Rightarrow \text{bool list option}) \times (\text{bool list} \times \text{bool list}) \text{ cstate}, \text{unit}, \text{bool list option}) \text{ oracle}' \text{ where}$

rorc-channel-recv $s \ q \equiv$ do {
 $(m, s) \leftarrow \dagger \dagger \text{channel.recv-oracle } s \ ();$
 (case m of
 None \Rightarrow return-spmf $(None, s)$
 | Some $(a, m) \Rightarrow$ do {
 $(r, s) \leftarrow \dagger (\text{rorc.rnd-oracle } (\text{rnd } \eta))^\dagger s \ m;$
 $a' \leftarrow \text{mac } \eta \ r \ m;$
 return-spmf $(\text{if } a' = a \text{ then Some } m \text{ else None}, s)$
 }
}

private definition *rorc-channel-recv-f* :: $((\text{bool list} \Rightarrow \text{bool list option}) \times (\text{bool list} \times \text{bool list}) \text{ cstate}, \text{unit}, \text{bool list option}) \text{ oracle}' \text{ where}$

rorc-channel-recv-f $s \ q \equiv$ do {
 $(am, (as, ams)) \leftarrow \dagger \text{channel.recv-oracle } s \ ();$
 (case am of
 None \Rightarrow return-spmf $(None, (as, ams))$
 | Some $(a, m) \Rightarrow$ (case as m of
 None \Rightarrow do {
 $a'' :: \text{bool list} \leftarrow \text{spmf-of-set } (\text{nlists UNIV } \eta - \{a\});$
 $a' \leftarrow \text{spmf-of-set } (\text{nlists UNIV } \eta);$
 (if $a' = a$
 then return-spmf $(None, \text{as}(m := \text{Some } a''), ams)$
 else return-spmf $(None, \text{as}(m := \text{Some } a'), ams)$) }
 | Some $a' \Rightarrow$ return-spmf $(\text{if } a' = a \text{ then Some } m \text{ else None}, as, ams))$
 })

private fun *lazy-channel-send* :: $(\text{bool list cstate} \times (\text{bool list} \times \text{bool list}) \text{ option} \times (\text{bool list} \Rightarrow \text{bool list option}), \text{bool list}, \text{unit}) \text{ oracle}' \text{ where}$

lazy-channel-send $(\text{Void}, es) \ m =$ return-spmf $(((), (\text{Store } m, es))$
| *lazy-channel-send* $s \quad m =$ return-spmf $(((), s)$

private fun *lazy-channel-recv* :: $(\text{bool list cstate} \times (\text{bool list} \times \text{bool list}) \text{ option} \times$

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

private fun *lazy-channel-insec* $:: (\text{bool list cstate} \times (\text{bool list} \times \text{bool list}) \text{option} \times$
 $(\text{bool list} \Rightarrow \text{bool list option}),$
 $(\text{bool list} \times \text{bool list}) \text{insec-query}, (\text{bool list} \times \text{bool list}) \text{option}) \text{ oracle}'$ **where**
lazy-channel-insec (*Void*, $-, \text{as}$) $(\text{Edit} (a', m')) = \text{return-spmf} (\text{None}, (\text{Collect}$
 $m', \text{Some} (a', m'), \text{as}))$
| *lazy-channel-insec* (*Store* $m, -, \text{as}$) $(\text{Edit} (a', m')) = \text{return-spmf} (\text{None}, (\text{Collect}$
 $m', \text{Some} (a', m'), \text{as}))$
| *lazy-channel-insec* (*Store* m, es) *Forward* $= \text{return-spmf} (\text{None}, (\text{Collect}$
 $m, es))$
| *lazy-channel-insec* (*Store* m, es) *Drop* $= \text{return-spmf} (\text{None}, (\text{Fail},$
 $es))$
| *lazy-channel-insec* (*Store* $m, \text{None}, \text{as}$) *Look* $= (\text{case as } m \text{ of}$
 None $\Rightarrow \text{do} \{$
 $a \leftarrow \text{spmf-of-set} (\text{rnd } \eta);$
 $\text{return-spmf} (\text{Some} (a, m), \text{Store } m, \text{None}, \text{as } (m := \text{Some } a)) \}$
| *Some* $a \Rightarrow \text{return-spmf} (\text{Some} (a, m), \text{Store } m, \text{None}, \text{as})$
| *lazy-channel-insec* $s \quad \quad \quad - \quad \quad \quad = \text{return-spmf} (\text{None}, s)$

private fun *lazy-channel-recv-f* $:: (\text{bool list cstate} \times (\text{bool list} \times \text{bool list}) \text{option}$
 $\times (\text{bool list} \Rightarrow \text{bool list option}), \text{unit}, \text{bool list option}) \text{ oracle}'$ **where**
lazy-channel-recv-f (*Collect* $m, \text{None}, \text{as}$) $() = \text{return-spmf} (\text{Some } m, (\text{Fail},$
 $\text{None}, \text{as}))$
| *lazy-channel-recv-f* ($ms, \text{Some} (a', m'), \text{as}$) $() = (\text{case as } m' \text{ of}$
 None $\Rightarrow \text{do} \{$
 $a \leftarrow \text{spmf-of-set} (\text{rnd } \eta);$
 $\text{return-spmf} (\text{None}, \text{Fail}, \text{None}, \text{as } (m' := \text{Some } a)) \}$
| *Some* $a \Rightarrow \text{return-spmf} (\text{if } a = a' \text{ then } \text{Some } m' \text{ else } \text{None}, \text{Fail}, \text{None}, \text{as})$
| *lazy-channel-recv-f* $s \quad \quad \quad () = \text{return-spmf} (\text{None}, s)$

private abbreviation *callee-auth-channel* **where**
callee-auth-channel *callee* $\equiv \text{lift-state-oracle extend-state-oracle} (\text{attach-callee callee}$
 $\text{auth-channel.auth-oracle})$

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

private inductive $S :: (\text{bool list cstate} \times (\text{bool list} \times \text{bool list}) \text{ option} \times (\text{bool list} \Rightarrow \text{bool list option})) \text{ spmf}$
 $\Rightarrow ((\text{bool} \times \text{unit}) \times (\text{bool list} \Rightarrow \text{bool list option}) \times (\text{bool list} \times \text{bool list}) \text{ cstate})$
 $\text{spmf} \Rightarrow \text{bool}$ **where**
 S (return-spmf (Void , None , Map.empty))
 $(\text{return-spmf}$ ($(\text{False}$, $()$), Map.empty , Void))
 $| S$ (return-spmf ($\text{Store } m$, None , Map.empty))
 $(\text{map-spmf}$ ($\lambda a. ((\text{True}, ()), [m \mapsto a], \text{Store } (a, m))$) (spmf-of-set ($nlists$ $UNIV$ η)))
if $\text{length } m = id' \eta$
 $| S$ (return-spmf ($\text{Collect } m$, None , Map.empty))
 $(\text{map-spmf}$ ($\lambda a. ((\text{True}, ()), [m \mapsto a], \text{Collect } (a, m))$) (spmf-of-set ($nlists$ $UNIV$ η)))
if $\text{length } m = id' \eta$
 $| S$ (return-spmf ($\text{Store } m$, None , $[m \mapsto a]$))
 $(\text{return-spmf}$ ($(\text{True}, ()), [m \mapsto a], \text{Store } (a, m)$))
if $\text{length } m = id' \eta$ **and** $\text{length } a = id' \eta$
 $| S$ (return-spmf ($\text{Collect } m$, None , $[m \mapsto a]$))
 $(\text{return-spmf}$ ($(\text{True}, ()), [m \mapsto a], \text{Collect } (a, m)$))
if $\text{length } m = id' \eta$ **and** $\text{length } a = id' \eta$
 $| S$ (return-spmf (Fail , None , Map.empty))
 $(\text{map-spmf}$ ($\lambda a. ((\text{True}, ()), [m \mapsto a], \text{Fail})$) (spmf-of-set ($nlists$ $UNIV$ η)))
if $\text{length } m = id' \eta$
 $| S$ (return-spmf (Fail , None , $[m \mapsto a]$))
 $(\text{return-spmf}$ ($(\text{True}, ()), [m \mapsto a], \text{Fail}$))
if $\text{length } m = id' \eta$ **and** $\text{length } a = id' \eta$
 $| S$ (return-spmf ($\text{Collect } m'$, $\text{Some } (a', m')$, Map.empty))
 $(\text{return-spmf}$ ($(\text{False}, ()), \text{Map.empty}$, $\text{Collect } (a', m')$))
if $\text{length } m' = id' \eta$ **and** $\text{length } a' = id' \eta$
 $| S$ (return-spmf ($\text{Collect } m'$, $\text{Some } (a', m')$, $[m \mapsto a]$))
 $(\text{return-spmf}$ ($(\text{True}, ()), [m \mapsto a], \text{Collect } (a', m')$))
if $\text{length } m = id' \eta$ **and** $\text{length } a = id' \eta$ **and** $\text{length } m' = id' \eta$ **and** $\text{length } a' = id' \eta$
 $| S$ (return-spmf ($\text{Collect } m'$, $\text{Some } (a', m')$, Map.empty))
 $(\text{map-spmf}$ ($\lambda x. ((\text{True}, ()), [m \mapsto x], \text{Collect } (a', m'))$) (spmf-of-set ($nlists$ $UNIV$ η)))
if $\text{length } m = id' \eta$ **and** $\text{length } m' = id' \eta$ **and** $\text{length } a' = id' \eta$
 $| S$ (map-spmf ($\lambda x. (\text{Fail}, \text{None}, \text{as}(m' \mapsto x))$) spmf-s)
 $(\text{map-spmf}$ ($\lambda x. ((\text{False}, ()), \text{as}(m' \mapsto x), \text{Fail})$) spmf-s)
if $\text{length } m' = id' \eta$ **and** $\text{lossless-spmf } \text{spmf-s}$
 $| S$ (map-spmf ($\lambda x. (\text{Fail}, \text{None}, \text{as}(m' \mapsto x))$) spmf-s)
 $(\text{map-spmf}$ ($\lambda x. ((\text{True}, ()), \text{as}(m' \mapsto x), \text{Fail})$) spmf-s)
if $\text{length } m' = id' \eta$ **and** $\text{lossless-spmf } \text{spmf-s}$
 $| S$ (return-spmf (Fail , None , $[m' \mapsto a']$))
 $(\text{map-spmf}$ ($\lambda x. ((\text{True}, ()), [m \mapsto x, m' \mapsto a'], \text{Fail})$) (spmf-of-set ($nlists$ $UNIV$ η)))
if $\text{length } m = id' \eta$ **and** $\text{length } m' = id' \eta$ **and** $\text{length } a' = id' \eta$
 $| S$ (map-spmf ($\lambda x. (\text{Fail}, \text{None}, [m' \mapsto x])$) (spmf-of-set ($nlists$ $UNIV$ $\eta \cap \{x. x \neq a'\}$)))

$(\text{map-spmf } (\lambda x. ((\text{True}, ()), [m \mapsto \text{fst } x, m' \mapsto \text{snd } x], \text{Fail})) (\text{spmf-of-set } (nlists \text{ UNIV } \eta \times nlists \text{ UNIV } \eta \cap \{x. \text{snd } x \neq a'\}))$
if $\text{length } m = \text{id}' \eta$ **and** $\text{length } m' = \text{id}' \eta$
 $| S (\text{map-spmf } (\lambda x. (\text{Fail}, \text{None}, \text{as}(m' \mapsto x))) \text{ spmf-s})$
 $(\text{map-spmf } (\lambda p. ((\text{True}, ()), \text{as}(m' \mapsto \text{fst } p, m \mapsto \text{snd } p), \text{Fail})) (\text{mk-lossless } (\text{pair-spmf } \text{ spmf-s } (\text{spmf-of-set } (nlists \text{ UNIV } \eta))))$
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 $(\text{valid-insecQ } \langle + \rangle \text{ nlists UNIV } (\text{id}' \eta) \langle + \rangle \text{ UNIV}) \vdash_R$
 $\text{RES } (\text{lazy-channel-insec } \oplus_O \text{ lazy-channel-send } \oplus_O \text{ lazy-channel-recv}) (\text{Void},$
 $\text{None}, \text{Map.empty})$
 \approx
 $\text{RES } (\dagger \text{insec-channel.insec-oracle } \oplus_O \text{ rorc-channel-send } \oplus_O \text{ rorc-channel-recv})$
 $((\text{False}, ()), \text{Map.empty}, \text{Void})$
 $(\text{is } ?A \vdash_R \text{ RES } (?L1 \oplus_O ?L2 \oplus_O ?L3) ?SL \approx \text{RES } (?R1 \oplus_O ?R2 \oplus_O ?R3)$
 $?SR)$

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

defines $\mathcal{I} \equiv \mathcal{I}\text{-uniform } (\text{Set.Collect } (\text{valid-mac-query } \eta)) (\text{insert None } (\text{Some } ' \text{ nlists UNIV } \eta \times nlists \text{ UNIV } \eta))) \oplus_{\mathcal{I}}$
 $(\mathcal{I}\text{-uniform } (\text{vld } \eta) \text{ UNIV } \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform UNIV } (\text{insert None } (\text{Some } ' \text{ vld } \eta)))$
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** $S' :: (((\text{bool list } \times \text{bool list}) \text{ option } + \text{unit}) \times \text{unit} \times \text{bool list} \text{ cstate}) \text{ spmf} \Rightarrow$

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

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

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 length m = id' η
| S' (return-spmf (Inr (), (), Fail))
  (return-spmf (Fail, None, Map.empty))
| S' (return-spmf (Inr (), (), Fail))
  (return-spmf (Fail, None, [m ↦ x]))
if length m = id' η
| S' (return-spmf (Inr (), (), Void))
  (return-spmf (Collect m', Some (a', m'), Map.empty))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inr (), (), Fail))
  (return-spmf (Collect m', Some (a', m'), Map.empty))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inr (), (), Store m))
  (return-spmf (Collect m', Some (a', m'), Map.empty))
if length m = id' η and length m' = id' η and length a' = id' η
| S' (return-spmf (Inl (Some (a', m')), (), Collect m'))
  (return-spmf (Collect m', Some (a', m'), [m' ↦ a']))
if length m' = id' η and length a' = id' η

| S' (return-spmf (Inl None, (), cstate.Collect m))
  (return-spmf (cstate.Collect m, None, Map.empty))
if length m = id' η
| S' (return-spmf (Inl None, (), cstate.Fail))
  (return-spmf (cstate.Fail, None, Map.empty))

| S' (return-spmf (Inr (), (), Fail))
  (return-spmf (Collect m', Some (a', m'), [m ↦ a]))
if length m = id' η and length m' = id' η and length a' = id' η and m ≠ m'
| S' (return-spmf (Inr (), (), Fail))
  (return-spmf (Collect m', Some (a', m'), [m ↦ a]))
if length m = id' η and length m' = id' η and length a' = id' η and a ≠ a'
| S' (return-spmf (Inl None, (), Collect m'))
  (return-spmf (Collect m', Some (a', m'), [m' ↦ a']))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inr (), (), Collect m'))
  (return-spmf (Collect m', Some (a', m'), [m' ↦ a']))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inr (), (), Void))
  (map-spmf (λa'. (Fail, None, [m' ↦ a'])) (spmf-of-set (nlists UNIV η)))
if length m' = id' η
| S' (return-spmf (Inr (), (), Fail))
  (map-spmf (λa'. (Fail, None, [m' ↦ a'])) (spmf-of-set (nlists UNIV η)))
if length m' = id' η
| S' (return-spmf (Inr (), (), Store m))
  (map-spmf (λa'. (Fail, None, [m' ↦ a'])) (spmf-of-set (nlists UNIV η)))
if length m = id' η and length m' = id' η
| S' (return-spmf (Inr (), (), Fail))
  (map-spmf (λa'. (Fail, None, [m ↦ a, m' ↦ a'])) (spmf-of-set (nlists UNIV
η)))

```

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

private lemma *trace-eq-sim*:

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

lemma *all-together*:

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

end

context begin

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

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

lemma *WT-enm*:

$X \neq \{\}$ \implies \mathcal{I} -uniform (vld η) UNIV, \mathcal{I} -uniform (vld η) $X \oplus_{\mathcal{I}}$ \mathcal{I} -uniform ($X \times$
vld η) UNIV \vdash_C MAC.enm η \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 $\eta \times$ 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 $\eta \times$ 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 η))) (CIPHER.dec η)
⟨proof⟩

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

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

interpretation *key: callee-invariant-on*

CIPHER.KEY.key-oracle $\eta \oplus_O$ CIPHER.KEY.key-oracle η
 λx . case x of None \Rightarrow True | Some $x' \Rightarrow$ length $x' = \eta$
 \mathcal{I} -uniform UNIV (nlists UNIV η) $\oplus_{\mathcal{I}}$ \mathcal{I} -full for η
⟨proof⟩

lemma *WT-enc* [*WT-intro*]: \mathcal{I} -uniform (nlists UNIV η) UNIV,

\mathcal{I} -uniform UNIV (nlists UNIV η) $\oplus_{\mathcal{I}}$ \mathcal{I} -uniform (vld η) UNIV \vdash_C CIPHER.enc
 $\eta \checkmark$
⟨proof⟩

lemma *WT-dec* [*WT-intro*]: \mathcal{I} -uniform UNIV (insert None (Some ‘ nlists UNIV
 η)),

\mathcal{I} -uniform UNIV (nlists UNIV η) $\oplus_{\mathcal{I}}$ \mathcal{I} -uniform UNIV (insert None (Some ‘
vld η)) \vdash_C
CIPHER.dec $\eta \checkmark$
⟨proof⟩

lemma *bound-enc* [*interaction-bound*]: *interaction-any-bounded-converter* (CIPHER.enc
 η) (*enat* 2)

⟨proof⟩

lemma *bound-dec* [*interaction-bound*]: *interaction-any-bounded-converter* (CIPHER.dec
 η) (*enat* 2)

⟨proof⟩

theorem *mac-otp*:

defines \mathcal{I} -real $\equiv \lambda \eta$. \mathcal{I} -uniform { x . valid-mac-query η x } UNIV

and \mathcal{I} -ideal $\equiv \lambda$ -. \mathcal{I} -full

and \mathcal{I} -common $\equiv \lambda \eta$. \mathcal{I} -uniform (vld η) UNIV $\oplus_{\mathcal{I}}$ \mathcal{I} -full

shows

constructive-security

($\lambda \eta$. $1_C \models (CIPHER.enc \eta \models CIPHER.dec \eta) \odot$ parallel-wiring \triangleright

parallel-resource1-wiring \triangleright

CIPHER.KEY.res $\eta \parallel$

($1_C \models MAC.enm \eta \models MAC.dem \eta \triangleright$


```

      1C |= parallel-wiring ▷
      parallel-resource1-wiring ▷ MAC.RO.res η || INSEC.res))
    (λ-. 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.