Constructive Cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar

April 24, 2025

Abstract

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

Contents

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*

⟨*proof*⟩

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

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*)
$\Longrightarrow$ *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* $\Longrightarrow$
rel-fun A (*rel-spmf* (*rel-prod* B (λ*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*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *rel-resourceD*:
rel-resource A B *res1 res2* $\Longrightarrow$ *rel-fun* A (*rel-spmf* (*rel-prod* B (*rel-resource* A B))) (*run-resource res1*) (*run-resource res2*)
⟨*proof*⟩

lemma *rel-resource-eq1*: *rel-resource* $(=) =$ *rel-resource'*
⟨*proof*⟩

lemma *rel-resource-eq*: *rel-resource* $(=)$ $(=)$ $=$ $(=)$
⟨*proof*⟩

lemma *rel-resource-mono*:
assumes $A' \leq A$ $B \leq B'$
shows *rel-resource* A $B \leq$ *rel-resource* A' B'
⟨*proof*⟩

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

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** $?lhs = ?rhs$)
$\langle proof \rangle$

lemma *rel-resource-map-resource'2*:
$rel\text{-}resource\ A\ B\ res1\ (map\text{-}resource'\ f\ res2) = rel\text{-}resource\ A\ (\lambda x\ y.\ B\ x\ (f\ y))\ res1\ res2$
$\langle proof \rangle$

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

lemma *rel-resource-pos-distr*:
$rel\text{-}resource\ A\ B\ OO\ rel\text{-}resource\ A'\ B' \le rel\text{-}resource\ (A\ OO\ A')\ (B\ OO\ B')$
$\langle proof \rangle$

lemma *left-unique-rel-resource*:
$[\![\ left\text{-}total\ A;\ left\text{-}unique\ B\ ]\!] \Longrightarrow left\text{-}unique\ (rel\text{-}resource\ A\ B)$
$\langle proof \rangle$

lemma *right-unique-rel-resource*:
$[\![\ right\text{-}total\ A;\ right\text{-}unique\ B\ ]\!] \Longrightarrow right\text{-}unique\ (rel\text{-}resource\ A\ B)$
$\langle proof \rangle$

lemma *bi-unique-rel-resource* [*transfer-rule*]:
$[\![\ bi\text{-}total\ A;\ bi\text{-}unique\ B\ ]\!] \Longrightarrow bi\text{-}unique\ (rel\text{-}resource\ A\ B)$
$\langle proof \rangle$

definition $rel\text{-}witness\text{-}resource :: ('a \Rightarrow 'e \Rightarrow bool) \Rightarrow ('e \Rightarrow 'c \Rightarrow bool) \Rightarrow ('b \Rightarrow 'd \Rightarrow bool) \Rightarrow ('a, 'b)\ resource \times ('c, 'd)\ resource \Rightarrow ('e, 'b \times 'd)\ resource$ **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*]:
$run\text{-}resource\ (rel\text{-}witness\text{-}resource\ A\ A'\ B\ (res1, res2)) =$
$map\text{-}spmf\ (map\text{-}prod\ id\ (rel\text{-}witness\text{-}resource\ A\ A'\ B) \circ rel\text{-}witness\text{-}prod) \circ$
$rel\text{-}witness\text{-}spmf\ (rel\text{-}prod\ B\ (rel\text{-}resource\ (A\ OO\ A')\ B)) \circ$
$rel\text{-}witness\text{-}fun\ A\ A'\ (run\text{-}resource\ res1,\ run\text{-}resource\ res2)$
$\langle proof \rangle$

lemma assumes $rel\text{-}resource\ (A\ OO\ A')\ B\ res\ res'$
and A: $left\text{-}unique\ A\ right\text{-}total\ A$
and A': $right\text{-}unique\ A'\ left\text{-}total\ A'$
shows *rel-witness-resource1*: $rel\text{-}resource\ A\ (\lambda b\ (b', c).\ b = b' \land B\ b'\ c)\ res\ (rel\text{-}witness\text{-}resource\ A\ A'\ B\ (res, res'))$ (**is** $?thesis1$)
and *rel-witness-resource2*: $rel\text{-}resource\ A'\ (\lambda(b, c')\ c.\ c = c' \land B\ b\ c')\ (rel\text{-}witness\text{-}resource$

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′*) ≤ *rel-resource A B OO rel-resource A′ B′*
⟨*proof*⟩

lemma *left-total-rel-resource*:
  ⟦ *left-unique A*; *right-total A*; *left-total B* ⟧ ⟹ *left-total* (*rel-resource A B*)
  ⟨*proof*⟩

lemma *right-total-rel-resource*:
  ⟦ *right-unique A*; *left-total A*; *right-total B* ⟧ ⟹ *right-total* (*rel-resource A B*)
  ⟨*proof*⟩

lemma *bi-total-rel-resource* [*transfer-rule*]:
  ⟦ *bi-total A*; *bi-unique A*; *bi-total B* ⟧ ⟹ *bi-total* (*rel-resource A B*)
  ⟨*proof*⟩

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

lemma *Resource-parametric* [*transfer-rule*]:
  ((*A* ===> *rel-spmf* (*rel-prod B* (*rel-resource A B*))) ===> *rel-resource A B*) *Resource Resource*
  ⟨*proof*⟩

lemma *run-resource-parametric* [*transfer-rule*]:
  (*rel-resource A B* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-resource A B*))) *run-resource run-resource*
  ⟨*proof*⟩

lemma *corec-resource-parametric* [*transfer-rule*]:
  ((*S* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-sum* (*rel-resource A B*) *S*))) ===> *S* ===> *rel-resource A B*)
    corec-resource corec-resource
⟨*proof*⟩

lemma *map-resource-parametric* [*transfer-rule*]:
  ((*A′* ===> *A*) ===> (*B* ===> *B′*) ===> *rel-resource A B* ===> *rel-resource A′ B′*) *map-resource map-resource*
  ⟨*proof*⟩

lemma *map-resource′-parametric* [*transfer-rule*]:
  ((*B* ===> *B′*) ===> *rel-resource* (=) *B* ===> *rel-resource* (=) *B′*) *map-resource′ map-resource′*
  ⟨*proof*⟩

lemma *case-resource-parametric* [*transfer-rule*]:
$(((A ===> \textit{rel-spmf} \ (\textit{rel-prod} \ B \ (\textit{rel-resource} \ A \ B))) ===> C) ===> \textit{rel-resource} \ A \ B ===> C)$
case-resource case-resource
⟨*proof*⟩

end

lemma *rel-resource-Grp*:
rel-resource (*conversep* (*BNF-Def.Grp UNIV f*)) (*BNF-Def.Grp UNIV g*) = *BNF-Def.Grp UNIV* (*map-resource f g*)
⟨*proof*⟩

1.4 Losslessness

coinductive *lossless-resource* :: $('a, 'b) \ \mathcal{I} \Rightarrow ('a, 'b) \ \textit{resource} \Rightarrow \textit{bool}$
for $\mathcal{I}$ **where**
lossless-resourceI: *lossless-resource* $\mathcal{I}$ *res* **if**
$\bigwedge a. \ a \in \textit{outs-}\mathcal{I} \ \mathcal{I} \Longrightarrow \textit{lossless-spmf} \ (\textit{run-resource res a})$
$\bigwedge a \ b \ res'. \ [\![\ a \in \textit{outs-}\mathcal{I} \ \mathcal{I}; \ (b, \ res') \in \textit{set-spmf} \ (\textit{run-resource res a}) \ ]\!] \Longrightarrow \textit{lossless-resource} \ \mathcal{I} \ res'$

lemma *lossless-resource-coinduct* [*consumes 1*, *case-names lossless-resource*, *case-conclusion lossless-resource lossless step*, *coinduct pred*: *lossless-resource*]:
assumes *X res*
and $\bigwedge res \ a. \ [\![\ X \ res; \ a \in \textit{outs-}\mathcal{I} \ \mathcal{I} \ ]\!] \Longrightarrow \textit{lossless-spmf} \ (\textit{run-resource res a}) \ \wedge$
$(\forall (b, \ res') \in \textit{set-spmf} \ (\textit{run-resource res a}). \ X \ res' \vee \textit{lossless-resource} \ \mathcal{I} \ res')$
shows *lossless-resource* $\mathcal{I}$ *res*
⟨*proof*⟩

lemma *lossless-resourceD*:
$[\![\ \textit{lossless-resource} \ \mathcal{I} \ res; \ a \in \textit{outs-}\mathcal{I} \ \mathcal{I} \ ]\!]$
$\Longrightarrow \textit{lossless-spmf} \ (\textit{run-resource res a}) \wedge (\forall (x, \ res') \in \textit{set-spmf} \ (\textit{run-resource res a}). \ \textit{lossless-resource} \ \mathcal{I} \ res')$
⟨*proof*⟩

lemma *lossless-resource-mono*:
assumes *lossless-resource* $\mathcal{I}'$ *res*
and *le*: $\textit{outs-}\mathcal{I} \ \mathcal{I} \subseteq \textit{outs-}\mathcal{I} \ \mathcal{I}'$
shows *lossless-resource* $\mathcal{I}$ *res*
⟨*proof*⟩

lemma *lossless-resource-mono'*:
$[\![\ \textit{lossless-resource} \ \mathcal{I}' \ res; \ \mathcal{I} \le \mathcal{I}' \ ]\!] \Longrightarrow \textit{lossless-resource} \ \mathcal{I} \ res$
⟨*proof*⟩

1.5 Operations

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

primcorec *resource-of-oracle* :: $'s \Rightarrow ('a, 'b)$ *resource* **where**
run-resource (*resource-of-oracle s*) = (λa. *map-spmf* (*map-prod id resource-of-oracle*) (*oracle s a*))

end

lemma *resource-of-oracle-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *A* ===> *rel-spmf* (*rel-prod B S*)) ===> *S* ===> *rel-resource A B*) *resource-of-oracle resource-of-oracle*
⟨*proof*⟩

lemma *map-resource-resource-of-oracle*:
map-resource f g (*resource-of-oracle oracle s*) = *resource-of-oracle* (*map-fun id* (*map-fun f* (*map-spmf* (*map-prod g id*))) *oracle*) *s*
for *s* :: $'s$
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *lossless-resource-of-oracle*:
assumes $*$: $\bigwedge s\ x.\ [\![\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow$ *lossless-spmf* (*callee s x*)
and *I s*
shows *lossless-resource* $\mathcal{I}$ (*resource-of-oracle callee s*)
⟨*proof*⟩

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

lemma *resource-of-oracle-rprodl*: **includes** *lifting-syntax* **shows**
resource-of-oracle ((*rprodl* ---> *id* ---> *map-spmf* (*map-prod id lprodr*)) *oracle*) ((*s1*, *s2*), *s3*) =
resource-of-oracle oracle (*s1*, *s2*, *s3*)
⟨*proof*⟩

lemma *resource-of-oracle-extend-state-oracle* [*simp*]:
resource-of-oracle (*extend-state-oracle oracle*) (s', *s*) = *resource-of-oracle oracle s*
⟨*proof*⟩

end

lemma *exec-gpv-resource-of-oracle*:
exec-gpv run-resource gpv (*resource-of-oracle oracle s*) = *map-spmf* (*map-prod id* (*resource-of-oracle oracle*)) (*exec-gpv oracle gpv s*)
⟨*proof*⟩

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

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

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

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

lemma *resource-of-parallel-oracle*:
resource-of-oracle (*parallel-oracle oracle1 oracle2*) (*s1*, *s2*) =
parallel-resource (*resource-of-oracle oracle1 s1*) (*resource-of-oracle oracle2 s2*)
⟨*proof*⟩

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

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

1.6 Well-typing

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

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

lemma *WT-resourceD*:

assumes $\mathcal{I} \vdash res\ res\ \surd\ q \in outs\text{-}\mathcal{I}\ \mathcal{I}\ (r,\ res') \in set\text{-}spmf\ (run\text{-}resource\ res\ q)$
shows $r \in responses\text{-}\mathcal{I}\ \mathcal{I}\ q \wedge \mathcal{I} \vdash res\ res'\ \surd$
⟨*proof*⟩

lemma *WT-resource-of-oracle* [*simp*]:
assumes $\bigwedge s.\ \mathcal{I} \vdash c\ oracle\ s\ \surd$
shows $\mathcal{I} \vdash res\ resource\text{-}of\text{-}oracle\ oracle\ s\ \surd$
⟨*proof*⟩

lemma *WT-resource-bot* [*simp*]: $bot \vdash res\ res\ \surd$
⟨*proof*⟩

lemma *WT-resource-full*: $\mathcal{I}\text{-}full \vdash res\ res\ \surd$
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *WT-resource-of-oracle*:
$I\ s \Longrightarrow \mathcal{I} \vdash res\ resource\text{-}of\text{-}oracle\ callee\ s\ \surd$
⟨*proof*⟩

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

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

lemma *WT-parallel-resource* [*WT-intro*]:
assumes $\mathcal{I}1 \vdash res\ res1\ \surd$
and $\mathcal{I}2 \vdash res\ res2\ \surd$
shows $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash res\ parallel\text{-}resource\ res1\ res2\ \surd$
⟨*proof*⟩

lemma *callee-invariant-run-resource*: $callee\text{-}invariant\text{-}on\ run\text{-}resource\ (\lambda res.\ \mathcal{I}$
$\vdash res\ res\ \surd)\ \mathcal{I}$
⟨*proof*⟩

lemma *callee-invariant-run-lossless-resource*:
$callee\text{-}invariant\text{-}on\ run\text{-}resource\ (\lambda res.\ lossless\text{-}resource\ \mathcal{I}\ res \wedge \mathcal{I} \vdash res\ res\ \surd)\ \mathcal{I}$
⟨*proof*⟩

interpretation *run-lossless-resource*:
$callee\text{-}invariant\text{-}on\ run\text{-}resource\ \lambda res.\ lossless\text{-}resource\ \mathcal{I}\ res \wedge \mathcal{I} \vdash res\ res\ \surd\ \mathcal{I}$
for $\mathcal{I}$
⟨*proof*⟩

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

2 Converters

2.1 Type definition

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

lemma *case-converter-conv-run-converter*: *case-converter f conv* = *f* (*run-converter conv*)
  ⟨*proof*⟩

2.2 Functor

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

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

lemma *map-converter-sel* [*simp*]:
  run-converter (*map-converter conv*) *a'* = *map-gpv'* (*map-prod b map-converter*) *out inn* (*run-converter conv* (*a a'*))
  ⟨*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-converter a b f g (*map-converter a′ b′ f′ g′ conv*) = *map-converter* (*a′* ∘ *a*) (*b* ∘ *b′*) (*f* ∘ *f′*) (*g′* ∘ *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* ⇒ (′*a*, ′*b*, ′*out*, ′*in*) *converter* ⇒ *bool* **for** *out* **where**
Out: *outsp-converter out conv* **if** *out* ∈ *outs-gpv* $\mathcal{I}'$ (*run-converter conv a*) *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$
| *Cont*: *outsp-converter out conv*
if (*b*, *conv′*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) *outsp-converter out conv′ a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$

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

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

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

end

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

context begin
⟨*ML*⟩

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

outsp-converter.cases[*to-set*]
and *simps* = *outsp-converter.simps*[*to-set*]
end

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

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

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

2.4 Relator

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

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

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

lemma *rel-converterD*:
rel-converter A B C R *conv1* *conv2*
$\Longrightarrow$ *rel-fun* A (*rel-gpv''* (*rel-prod* B (*rel-converter* A B C R)) C R) (*run-converter* *conv1*) (*run-converter* *conv2*)
⟨*proof*⟩

lemma *rel-converter-eq14*: *rel-converter* (=) B C (=) = *rel-converter'* B C (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *rel-converter-eq* [*relator-eq*]: *rel-converter* (=) (=) (=) (=) = (=)
⟨*proof*⟩

lemma *rel-converter-mono* [*relator-mono*]:
assumes $A' \leq A$ $B \leq B'$ $C \leq C'$ $R' \leq R$
shows *rel-converter* A B C $R \leq$ *rel-converter* A' B' C' R'
⟨*proof*⟩

lemma *rel-converter-conversep*: *rel-converter* A^{-1-1} B^{-1-1} C^{-1-1} R^{-1-1} = (*rel-converter* A B C R)$^{-1-1}$
⟨*proof*⟩

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

lemma *rel-converter-map-converter'2*:
rel-converter A B C R *conv1* (*map-converter'* f g *conv2*) = *rel-converter* A (λx $y.$ B x (f y)) (λx $y.$ C x (g y)) R *conv1* *conv2*
⟨*proof*⟩

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

lemma *rel-converter-pos-distr* [*relator-distr*]:
rel-converter A B C R *OO* *rel-converter* A' B' C' $R' \leq$ *rel-converter* (A *OO* A') (B *OO* B') (C *OO* C') (R *OO* R')
⟨*proof*⟩

lemma *left-unique-rel-converter*:
⟦ *left-total* A; *left-unique* B; *left-unique* C; *left-total* R ⟧ $\Longrightarrow$ *left-unique* (*rel-converter* A B C R)
⟨*proof*⟩

lemma *right-unique-rel-converter*:
 $[\![$ *right-total* A; *right-unique* B; *right-unique* C; *right-total* R $]\!] \Longrightarrow$ *right-unique* (*rel-converter* A B C R)
 ⟨*proof*⟩

lemma *bi-unique-rel-converter* [*transfer-rule*]:
 $[\![$ *bi-total* A; *bi-unique* B; *bi-unique* C; *bi-total* R $]\!] \Longrightarrow$ *bi-unique* (*rel-converter* A B C R)
 ⟨*proof*⟩

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

lemma *rel-witness-converter-sel* [*simp*]:
 run-converter (*rel-witness-converter* A A' B C R R' (*conv1*, *conv2*)) =
 map-gpv (*map-prod id* (*rel-witness-converter* A A' B C R R') ∘ *rel-witness-prod*) *id* ∘
 rel-witness-gpv (*rel-prod* B (*rel-converter* (A *OO* A') B C (R *OO* R'))) C R R' ∘
 rel-witness-fun A A' (*run-converter conv1*, *run-converter conv2*)
 ⟨*proof*⟩

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 *conv* (*rel-witness-converter* A A' B C R R' (*conv*, *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' (*rel-witness-converter* A A' B C R R' (*conv*, *conv'*)) *conv'* (**is** *?thesis2*)
⟨*proof*⟩

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

and R': *right-unique* R' *left-total* R'
shows *rel-converter* $(A\ OO\ A')$ $(B\ OO\ B')$ $(C\ OO\ C')$ $(R\ OO\ R') \leq$ *rel-converter* $A\ B\ C\ R$ *OO rel-converter* $A'\ B'\ C'\ R'$
⟨*proof*⟩

lemma *left-total-rel-converter*:
$\llbracket$ *left-unique* A; *right-total* A; *left-total* B; *left-total* C; *left-unique* R; *right-total* R $\rrbracket$
$\Longrightarrow$ *left-total* (*rel-converter* $A\ B\ C\ R$)
⟨*proof*⟩

lemma *right-total-rel-converter*:
$\llbracket$ *right-unique* A; *left-total* A; *right-total* B; *right-total* C; *right-unique* R; *left-total* R $\rrbracket$
$\Longrightarrow$ *right-total* (*rel-converter* $A\ B\ C\ R$)
⟨*proof*⟩

lemma *bi-total-rel-converter* [*transfer-rule*]:
$\llbracket$ *bi-total* A; *bi-unique* A; *bi-total* B; *bi-total* C; *bi-total* R; *bi-unique* R $\rrbracket$
$\Longrightarrow$ *bi-total* (*rel-converter* $A\ B\ C\ R$)
⟨*proof*⟩

inductive *pred-converter* :: $'a$ *set* $\Rightarrow$ $('b \Rightarrow bool) \Rightarrow ('out \Rightarrow bool) \Rightarrow$ $'in$ *set* $\Rightarrow$ $('a, 'b, 'out, 'in)$ *converter* $\Rightarrow$ *bool*
for $A\ B\ C\ R$ *conv* **where**
pred-converter $A\ B\ C\ R$ *conv* **if**
$\forall\, x \in$ *results-converter* ($\mathcal{I}$-*uniform* A *UNIV*) ($\mathcal{I}$-*uniform UNIV* R) *conv*. $B\ x$
$\forall$ *out* $\in$ *outs-converter* ($\mathcal{I}$-*uniform* A *UNIV*) ($\mathcal{I}$-*uniform UNIV* R) *conv*. C *out*

lemma *pred-gpv'-mono-weak*:
pred-gpv' $A\ C\ R \leq$ *pred-gpv'* $A'\ C'\ R$ **if** $A \leq A'$ $C \leq C'$
⟨*proof*⟩

lemma *Domainp-rel-converter-le*:
Domainp (*rel-converter* $A\ B\ C\ R$) $\leq$ *pred-converter* (*Collect* (*Domainp* A)) (*Domainp* B) (*Domainp* C) (*Collect* (*Domainp* R))
(**is** *?lhs* $\leq$ *?rhs*)
⟨*proof*⟩

lemma *rel-converter-Grp*:
rel-converter (*BNF-Def.Grp UNIV* $f)^{-1-1}$ (*BNF-Def.Grp* $B\ g$) (*BNF-Def.Grp* $C\ h$) (*BNF-Def.Grp UNIV* $k)^{-1-1}$ =
BNF-Def.Grp {*conv*. *results-converter* ($\mathcal{I}$-*uniform* (*range* f) *UNIV*) ($\mathcal{I}$-*uniform UNIV* (*range* k)) *conv* $\subseteq B\ \wedge$
outs-converter ($\mathcal{I}$-*uniform* (*range* f) *UNIV*) ($\mathcal{I}$-*uniform UNIV* (*range* k)) *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* ===> *rel-gpv′′* (*rel-prod B* (*rel-converter A B C R*)) *C R*) ===> *rel-converter A B C R*) *Converter Converter*
⟨*proof*⟩

lemma *run-converter-parametric* [*transfer-rule*]:
(*rel-converter A B C R* ===> *A* ===> *rel-gpv′′* (*rel-prod B* (*rel-converter A B C R*)) *C R*)
run-converter run-converter
⟨*proof*⟩

lemma *corec-converter-parametric* [*transfer-rule*]:
((*S* ===> *A* ===> *rel-gpv′′* (*rel-prod B* (*rel-sum* (*rel-converter A B C R*) *S*)) *C R*) ===> *S* ===> *rel-converter A B C R*)
corec-converter corec-converter
⟨*proof*⟩

lemma *map-converter-parametric* [*transfer-rule*]:
((*A′* ===> *A*) ===> (*B* ===> *B′*) ===> (*C* ===> *C′*) ===> (*R′* ===> *R*) ===> *rel-converter A B C R* ===> *rel-converter A′ B′ C′ R′*)
map-converter map-converter
⟨*proof*⟩

lemma *map-converter′-parametric* [*transfer-rule*]:
((*B* ===> *B′*) ===> (*C* ===> *C′*) ===> *rel-converter* (=) *B C* (=) ===> *rel-converter* (=) *B′ C′* (=))
map-converter′ map-converter′
⟨*proof*⟩

lemma *case-converter-parametric* [*transfer-rule*]:
(((*A* ===> *rel-gpv′′* (*rel-prod B* (*rel-converter A B C R*)) *C R*) ===> *X*) ===> *rel-converter A B C R* ===> *X*)
case-converter case-converter
⟨*proof*⟩

end

2.5 Well-typing

coinductive *WT-converter* :: (*′a*, *′b*) $\mathcal{I}$ ⇒ (*′out*, *′in*) $\mathcal{I}$ ⇒ (*′a*, *′b*, *′out*, *′in*) *converter* ⇒ *bool*
(‹-,/ - $\vdash_C$/ - √› [*100*, *0*, *0*] *99*)

for $\mathcal{I}$ $\mathcal{I}'$ **where**
WT-converterI: $\mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv* $\surd$ **if**
$\bigwedge q.\ q \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow \mathcal{I}' \vdash g$ *run-converter conv q* $\surd$
$\bigwedge q\ r\ conv'.$ $[\![$ $q \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; $(r,\ conv') \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*) $]\!]$
$\Longrightarrow r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ $q \wedge \mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv'* $\surd$

lemma *WT-converter-coinduct*[*consumes 1*, *case-names WT-converter*, *case-conclusion WT-converter WT-gpv results-gpv*, *coinduct pred*: *WT-converter*]:
assumes *X conv*
and $\bigwedge$*conv q r conv'*. $[\![$ *X conv*; $q \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!]$
$\Longrightarrow \mathcal{I}' \vdash g$ *run-converter conv q* $\surd \wedge$
$((r,\ conv') \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*) $\longrightarrow r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ $q \wedge$ (*X conv'* $\vee$ $\mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv'* $\surd$))
shows $\mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv* $\surd$
⟨*proof*⟩

lemma *WT-converterD*:
assumes $\mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv* $\surd$ $q \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$
shows *WT-converterD-WT*: $\mathcal{I}' \vdash g$ *run-converter conv q* $\surd$
and *WT-converterD-results*: $(r,\ conv') \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*) $\Longrightarrow r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ $q \wedge \mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv'* $\surd$
⟨*proof*⟩

lemma *WT-converterD'*:
assumes $\mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv* $\surd$ $q \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$
shows $\mathcal{I}' \vdash g$ *run-converter conv q* $\surd \wedge (\forall\ (r,\ conv') \in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv q*). $r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ $q \wedge \mathcal{I}$, $\mathcal{I}' \vdash_C$ *conv'* $\surd$)
⟨*proof*⟩

lemma *WT-converter-bot1* [*simp*]: *bot*, $\mathcal{I} \vdash_C$ *conv* $\surd$
⟨*proof*⟩

lemma *WT-converter-mono*:
$[\![$ $\mathcal{I}1$,$\mathcal{I}2 \vdash_C$ *conv* $\surd$; $\mathcal{I}1' \leq \mathcal{I}1$; $\mathcal{I}2 \leq \mathcal{I}2'$ $]\!]$ $\Longrightarrow \mathcal{I}1'$,$\mathcal{I}2' \vdash_C$ *conv* $\surd$
⟨*proof*⟩

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

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

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

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* $\surd$
  ⟨*proof*⟩

lemma *WT-converter-map-converter* [*WT-intro*]:
  $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *map-converter f g f' g' conv* $\surd$ **if**
  $*$: *map-*$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$, *map-*$\mathcal{I}$ *f' g'* $\mathcal{I}'$ $\vdash_C$ *conv* $\surd$
  and *f*: *inj f* **and** *g*: *surj g*
  ⟨*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* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*)
    $\bigwedge$*a b conv'*. ⟦ *a* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; (*b*, *conv'*) $\in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) ⟧ $\Longrightarrow$ *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv'*

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

lemma *plossless-converterD*:
  ⟦ *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*; *a* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
  $\Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*) $\wedge$
    ($\forall$ (*b*, *conv'*) $\in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*). *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv'*)
  ⟨*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*: *outs-*$\mathcal{I}$ $\mathcal{I}1'$ $\subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}1$ $\mathcal{I}2 \leq \mathcal{I}2'$
    and *WT*: $\mathcal{I}1$, $\mathcal{I}2$ $\vdash_C$ *conv* $\surd$
  shows *plossless-converter* $\mathcal{I}1'$ $\mathcal{I}2'$ *conv*
  ⟨*proof*⟩

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

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

end

lemma *converter-of-callee-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *A* ===> *rel-gpv''* (*rel-prod B S*) *C R*) ===> *S* ===> *rel-converter A B C R*)
converter-of-callee converter-of-callee
⟨*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* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\Longrightarrow$ $\mathcal{I}'$ $\vdash$*g callee s q* $\surd$
and *res*: $\bigwedge$*s q r s'*. ⟦ *q* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; (*r*, *s'*) $\in$ *results-gpv* $\mathcal{I}'$ (*callee s q*) ⟧ $\Longrightarrow$ *r* $\in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q*
shows $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *converter-of-callee callee s* $\surd$
⟨*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*

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

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

lemma *parallel-converter-assoc*:
parallel-converter (*parallel-converter conv1 conv2*) *conv3* =
map-converter rsuml lsumr id id (*parallel-converter conv1* (*parallel-converter conv2 conv3*))
⟨*proof*⟩

lemma *plossless-parallel-converter* [*plossless-intro*]:
$[\![$ *plossless-converter* $\mathcal{I}1$ $\mathcal{I}$ *conv1*; *plossless-converter* $\mathcal{I}2$ $\mathcal{I}$ *conv2*; $\mathcal{I}1, \mathcal{I} \vdash_C$ *conv1* $\surd$; $\mathcal{I}2, \mathcal{I} \vdash_C$ *conv2* $\surd$ $]\!]$
$\Longrightarrow$ *plossless-converter* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) $\mathcal{I}$ (*parallel-converter conv1 conv2*)
⟨*proof*⟩

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

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

lemma *converter-of-callee-id-oracle* [*simp*]:
converter-of-callee id-oracle s = *id-converter*
⟨*proof*⟩

lemma *conv-callee-plus-id-left*: *converter-of-callee* (*plus-intercept id-oracle callee*) *s* =
parallel-converter id-converter (*converter-of-callee callee s*)
⟨*proof*⟩

lemma *conv-callee-plus-id-right*: *converter-of-callee* (*plus-intercept callee id-oracle*) *s* =
parallel-converter (*converter-of-callee callee s*) *id-converter*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *map-converter-id-move-left*:
map-converter f g f' g' id-converter = *map-converter* (*f'* ∘ *f*) (*g* ∘ *g'*) *id id id-converter*
⟨*proof*⟩

lemma *map-converter-id-move-right*:
map-converter f g f' g' id-converter = *map-converter id id* (*f'* ∘ *f*) (*g* ∘ *g'*) *id-converter*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

⟨*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 $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}1 \Longrightarrow$ *plossless-gpv* $\mathcal{I}2$ (*callee s x*) $\wedge$ ($\forall$ (y, s')$\in$*results-gpv* $\mathcal{I}2$ (*callee s x*). $y \in$ *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 :: $'a$ *set*
and $\mathcal{I}$ :: ($'c$, $'d$) $\mathcal{I}$
begin

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

end

lemma *WT-restrict-converter* [*WT-intro*]:
assumes $\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *cnv* $\surd$
shows $\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *restrict-converter* A $\mathcal{I}'$ *cnv* $\surd$
⟨*proof*⟩

lemma *pgen-lossless-restrict-gpv* [*simp*]:
$\mathcal{I} \vdash g$ *gpv* $\surd \Longrightarrow$ *pgen-lossless-gpv* b $\mathcal{I}$ (*restrict-gpv* $\mathcal{I}$ *gpv*) = *pgen-lossless-gpv* b $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *plossless-restrict-converter* [*simp*]:
assumes *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*
and $\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *conv* $\surd$
and *outs-*$\mathcal{I}$ $\mathcal{I} \subseteq A$
shows *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ (*restrict-converter* A $\mathcal{I}'$ *conv*)
⟨*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* ⇒ ('*out*, '*in*) *resource* ⇒ ('*a*, '*b*) *resource* **where**
run-resource (*attach conv res*) = (λ*a*.
map-spmf (λ((*b*, *conv'*), *res'*). (*b*, *attach conv' res'*)) (*exec-gpv run-resource* (*run-converter conv a*) *res*))

lemma *attach-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-resource C R* ===> *rel-resource A B*) *attach attach*
⟨*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*]: ⟦ $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* √; $\mathcal{I}'$ ⊢*res res* √ ⟧ ⟹ $\mathcal{I}$ ⊢*res attach conv res* √
⟨*proof*⟩

lemma *lossless-attach* [*plossless-intro*]:
assumes *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*
and *lossless-resource* $\mathcal{I}'$ *res*
and $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* √ $\mathcal{I}'$ ⊢*res res* √
shows *lossless-resource* $\mathcal{I}$ (*attach conv res*)
⟨*proof*⟩

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

lemma *attach-callee-simps* [*simp*]:
attach-callee callee oracle (*s*, *s'*) *q* = *map-spmf rprodl* (*exec-gpv oracle* (*callee s q*) *s'*)
⟨*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* (λ*s q*. *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* ⇒ ($'out$, $'in$, $'out'$, $'in'$) *converter* ⇒ ($'a$, $'b$, $'out'$, $'in'$) *converter* **where**
run-converter (*comp-converter conv1 conv2*) = (λ*a*.
map-gpv (λ((*b*, *conv1'*), *conv2'*). (*b*, *comp-converter conv1' conv2'*)) *id* (*inline run-converter* (*run-converter conv1 a*) *conv2*))

lemma *comp-converter-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-converter C R C' R'* ===> *rel-converter A B C' R'*)
comp-converter comp-converter
⟨*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-eqs* =
asm-rl[**where** *psi*=*x* = *y* **for** *x y* :: (-, -, -, -) *converter*]
comp-converter-assoc-left
comp-converter-attach-left

lemma *WT-converter-comp* [*WT-intro*]:
$[\![\ \mathcal{I}, \mathcal{I}' \vdash_C conv \surd; \mathcal{I}', \mathcal{I}'' \vdash_C conv' \surd\ ]\!] \Longrightarrow \mathcal{I}, \mathcal{I}'' \vdash_C$ *comp-converter conv conv'* $\surd$
⟨*proof*⟩

lemma *plossless-comp-converter* [*plossless-intro*]:
assumes *plossless-converter* $\mathcal{I}\ \mathcal{I}'$ *conv*
and *plossless-converter* $\mathcal{I}'\ \mathcal{I}''$ *conv'*
and $\mathcal{I}, \mathcal{I}' \vdash_C conv \surd\ \mathcal{I}', \mathcal{I}'' \vdash_C conv' \surd$
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-coverter-of-callee*: *comp-converter* (*converter-of-callee callee1 s1*) (*converter-of-callee callee2 s2*)

= *converter-of-callee* (λ(*s1*, *s2*) *q*. *map-gpv rprodl id* (*inline callee2* (*callee1 s1 q*) *s2*)) (*s1*, *s2*)
⟨*proof*⟩

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

lemma *comp-converter-parallel2*: *comp-converter* (*parallel-converter2 conv1l conv1r*) (*parallel-converter2 conv2l conv2r*) =
parallel-converter2 (*comp-converter conv1l conv2l*) (*comp-converter conv1r conv2r*)
⟨*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**
⋀*a*. *interaction-any-bounded-by* (*run-converter conv a*) *n*
⋀*a b conv′*. (*b*, *conv′*) ∈ *results′-gpv* (*run-converter conv a*) ⟹ *interaction-any-bounded-converter conv′ n*

lemma *interaction-any-bounded-converterD*:
assumes *interaction-any-bounded-converter conv n*
shows *interaction-any-bounded-by* (*run-converter conv a*) *n* ∧ (∀ (*b*, *conv′*)∈*results′-gpv* (*run-converter conv a*). *interaction-any-bounded-converter conv′ n*)
⟨*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* - - - ⇒ *fail* | *interaction-any-bounded-converter* - - ⇒ *fail* | - ⇒ ‹*solves* ‹*clarsimp simp add*: *simp*››) | *rule add interaction-bound*)
method *interaction-bound-converter-rec* **uses** *add simp* =
(*interaction-bound-converter-step add*: *add simp*: *simp*; (*interaction-bound-converter-rec add*: *add simp*: *simp*)?)
method *interaction-bound-converter* **uses** *add simp* =
(*interaction-bound-converter-start*, *interaction-bound-converter-rec add*: *add simp*: *simp*)

lemma *interaction-any-bounded-converter-id* [*interaction-bound*]:
interaction-any-bounded-converter id-converter 1
⟨*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 $\bigwedge x$. *consider'* (*Inl x*) $\Longrightarrow$ *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 $\bigwedge x$. *consider'* (*Inr x*) $\Longrightarrow$ *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 ⋀*s x*. *interaction-any-bounded-by* (*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* $\mathcal{I}$ *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: $A \vdash_R$ *res* $\sim$ *res′* **if**
$\bigwedge a.\ a \in A \Longrightarrow$ *rel-spmf* (*rel-prod* (=) (*eq-resource-on* A)) (*run-resource res a*) (*run-resource res′ a*)

lemma *eq-resource-on-coinduct* [*consumes 1*, *case-names eq-resource-on*, *coinduct pred*: *eq-resource-on*]:
assumes X *res res′*
and $\bigwedge$*res res′ a*. ⟦ X *res res′*; $a \in A$ ⟧
$\Longrightarrow$ *rel-spmf* (*rel-prod* (=) (λ*res res′*. X *res res′* $\vee$ $A \vdash_R$ *res* $\sim$ *res′*)) (*run-resource res a*) (*run-resource res′ a*)
shows $A \vdash_R$ *res* $\sim$ *res′*
⟨*proof*⟩

lemma *eq-resource-onD*:
assumes $A \vdash_R$ *res* $\sim$ *res′* $a \in A$
shows *rel-spmf* (*rel-prod* (=) (*eq-resource-on* A)) (*run-resource res a*) (*run-resource res′ a*)
⟨*proof*⟩

lemma *eq-resource-on-refl* [*simp*]: $A \vdash_R$ *res* $\sim$ *res*
⟨*proof*⟩

lemma *eq-resource-on-reflI*: *res* = *res′* $\Longrightarrow$ $A \vdash_R$ *res* $\sim$ *res′*
⟨*proof*⟩

lemma *eq-resource-on-sym*: $A \vdash_R$ *res* $\sim$ *res′* **if** $A \vdash_R$ *res′* $\sim$ *res*
⟨*proof*⟩

lemma *eq-resource-on-trans* [*trans*]: $A \vdash_R$ *res* $\sim$ *res′′* **if** $A \vdash_R$ *res* $\sim$ *res′* $A \vdash_R$ *res′* $\sim$ *res′′*
⟨*proof*⟩

lemma *eq-resource-on-UNIV-D* [*simp*]: *res* = *res′* **if** *UNIV* $\vdash_R$ *res* $\sim$ *res′*
⟨*proof*⟩

lemma *eq-resource-on-UNIV-iff*: *UNIV* $\vdash_R$ *res* $\sim$ *res′* $\longleftrightarrow$ *res* = *res′*
⟨*proof*⟩

lemma *eq-resource-on-mono*: ⟦ $A' \vdash_R$ *res* $\sim$ *res′*; $A \subseteq A'$ ⟧ $\Longrightarrow$ $A \vdash_R$ *res* $\sim$ *res′*

⟨*proof*⟩

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

lemma *eq-resource-on-resource-of-oracleI*:
includes *lifting-syntax*
fixes *S*
assumes *sim*: (*S* ===> *eq-on A* ===> *rel-spmf* (*rel-prod* (=) *S*)) *r1 r2*
and *S*: *S s1 s2*
shows $A \vdash_R$ *resource-of-oracle r1 s1* $\sim$ *resource-of-oracle r2 s2*
⟨*proof*⟩

lemma *exec-gpv-eq-resource-on*:
assumes *outs-*$\mathcal{I}$ $\mathcal{I} \vdash_R$ *res* $\sim$ *res'*
and $\mathcal{I} \vdash g$ *gpv* $\surd$
and $\mathcal{I} \vdash res$ *res* $\surd$
shows *rel-spmf* (*rel-prod* (=) (*eq-resource-on* (*outs-*$\mathcal{I}$ $\mathcal{I}$))) (*exec-gpv run-resource gpv res*) (*exec-gpv run-resource gpv res'*)
⟨*proof*⟩

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

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

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

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

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

lemma *eq-*$\mathcal{I}$*-generat-mono'*:
⟦ *eq-*$\mathcal{I}$*-generat A* $\mathcal{I}$ *D x y*; $\bigwedge$*x y*. *A x y* ⟹ *A' x y*; $\bigwedge$*x y*. *D x y* ⟹ *D' x y*; $\mathcal{I} \leq \mathcal{I}'$ ⟧
⟹ *eq-*$\mathcal{I}$*-generat A'* $\mathcal{I}'$ *D' x y*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

coinductive *eq-$\mathcal{I}$-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**

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

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

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

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

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

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

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

lemma $eq\text{-}\mathcal{I}\text{-}gpv\text{-}mono'$:
$[\![\ eq\text{-}\mathcal{I}\text{-}gpv\ A\ \mathcal{I}\ gpv\ gpv';\ \bigwedge x\ y.\ A\ x\ y \Longrightarrow A'\ x\ y;\ \mathcal{I} \le \mathcal{I}'\ ]\!] \Longrightarrow eq\text{-}\mathcal{I}\text{-}gpv\ A'\ \mathcal{I}'\ gpv\ gpv'$
$\langle proof \rangle$

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

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

lemma $eq\text{-}\mathcal{I}\text{-}gpv\text{-}reflI$:
$[\![\ \bigwedge x.\ x \in results\text{-}gpv\ \mathcal{I}\ gpv \Longrightarrow A\ x\ x;\ \mathcal{I} \vdash g\ gpv\ \surd\ ]\!] \Longrightarrow eq\text{-}\mathcal{I}\text{-}gpv\ A\ \mathcal{I}\ gpv\ gpv$
$\langle proof \rangle$

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

lemma *eq-$\mathcal{I}$-gpv-relcompp*: *eq-$\mathcal{I}$-gpv* $(A\ OO\ A')$ $\mathcal{I}$ = *eq-$\mathcal{I}$-gpv* A $\mathcal{I}$ *OO* *eq-$\mathcal{I}$-gpv* A' $\mathcal{I}$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

lemma *eq-$\mathcal{I}$-gpv-map-gpv2*: *eq-$\mathcal{I}$-gpv* A $\mathcal{I}$ *gpv* (*map-gpv* f *id* *gpv'*) = *eq-$\mathcal{I}$-gpv* (λx $y.$ 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 ⟧ ⟹ *rel-spmf* (*rel-prod* A (*eq-onp* I)) (*exec-gpv* *callee* *gpv* s) (*exec-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*: ⋀*gpv* *gpv'*. X *gpv* *gpv'*
⟹ *rel-spmf* (*eq-$\mathcal{I}$-generat* A $\mathcal{I}$ (λ*gpv* *gpv'*. X *gpv* *gpv'* ∨ *eq-$\mathcal{I}$-gpv* A $\mathcal{I}$ *gpv* *gpv'* ∨
(∃ *gpv''* *gpv'''* (B :: '*b* ⇒ '*b'* ⇒ *bool*) f g. *gpv* = *bind-gpv* *gpv''* f ∧ *gpv'* = *bind-gpv* *gpv'''* g ∧ *eq-$\mathcal{I}$-gpv* B $\mathcal{I}$ *gpv''* *gpv'''* ∧ (*rel-fun* B X) f g))) (*the-gpv* *gpv*) (*the-gpv* *gpv'*)
shows *eq-$\mathcal{I}$-gpv* A $\mathcal{I}$ *gpv* *gpv'*
⟨*proof*⟩

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

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

$\mathcal{I}$ *q''*)) *S*) $\mathcal{I}'$ (*rpv1 r*) (*rpv1' r*)) ∧
(∀ *r'* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q''*. *eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ (*rpv2 r'*) (*rpv2' r'*)))))
(*inline1 callee1 gpv1 s1*) (*inline1 callee2 gpv2 s2*)
⟨*proof*⟩

lemma *eq-*$\mathcal{I}$*-gpv-inline*:
assumes *S*: *S s1 s2*
and *gpv*: *eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv1 gpv2*
shows *eq-*$\mathcal{I}$*-gpv* (*rel-prod A S*) $\mathcal{I}'$ (*inline callee1 gpv1 s1*) (*inline callee2 gpv2 s2*)
⟨*proof*⟩

end

lemma *eq-*$\mathcal{I}$*-gpv-left-gpv-cong*:
eq-$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv gpv'* ⟹ *eq-*$\mathcal{I}$*-gpv A* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*left-gpv gpv*) (*left-gpv gpv'*)
⟨*proof*⟩

lemma *eq-*$\mathcal{I}$*-gpv-right-gpv-cong*:
eq-$\mathcal{I}$*-gpv A* $\mathcal{I}'$ *gpv gpv'* ⟹ *eq-*$\mathcal{I}$*-gpv A* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*right-gpv gpv*) (*right-gpv gpv'*)
⟨*proof*⟩

lemma *eq-*$\mathcal{I}$*-gpvD-WT1*: ⟦ *eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv gpv'*; $\mathcal{I}$ ⊢*g gpv* √ ⟧ ⟹ $\mathcal{I}$ ⊢*g gpv'* √
⟨*proof*⟩

lemma *eq-*$\mathcal{I}$*-gpvD-results-gpv2*:
assumes *eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv gpv' y* ∈ *results-gpv* $\mathcal{I}$ *gpv'*
shows ∃ *x* ∈ *results-gpv* $\mathcal{I}$ *gpv*. *A x y*
⟨*proof*⟩

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

lemma *eq-*$\mathcal{I}$*-converter-coinduct* [*consumes 1*, *case-names eq-*$\mathcal{I}$*-converter*, *coinduct pred*: *eq-*$\mathcal{I}$*-converter*]:
assumes *X conv conv'*
and ⋀*conv conv' q*. ⟦ *X conv conv'*; *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ *eq-*$\mathcal{I}$*-gpv* (*rel-prod* (*eq-onp* (λ*r*. *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q*)) (λ*conv conv'*. *X conv conv'* ∨ $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* ∼ *conv'*)) $\mathcal{I}'$
(*run-converter conv q*) (*run-converter conv' q*)
shows $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* ∼ *conv'*
⟨*proof*⟩

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

eq-$\mathcal{I}$-gpv (*rel-prod* (*eq-onp* ($\lambda r.$ $r \in$ *responses-$\mathcal{I}$* $\mathcal{I}$ q)) (*eq-$\mathcal{I}$-converter* $\mathcal{I}$ $\mathcal{I}'$)) $\mathcal{I}'$ (*run-converter conv q*) (*run-converter conv' q*)
if $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv* $\sim$ *conv'* $q \in$ *outs-$\mathcal{I}$* $\mathcal{I}$
⟨*proof*⟩

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

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

lemma *eq-$\mathcal{I}$-converter-trans* [*trans*]:
⟦ $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv* $\sim$ *conv'*; $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv'* $\sim$ *conv''* ⟧ $\Longrightarrow$ $\mathcal{I}, \mathcal{I}' \vdash_C$ *conv* $\sim$ *conv''*
⟨*proof*⟩

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

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

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

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

lemma *eq-$\mathcal{I}$-attach*:
⟦ $\mathcal{I}' \vdash$*res res* $\surd$; *$\mathcal{I}$-full*, $\mathcal{I}' \vdash_C$ *conv* $\sim$ *conv'* ⟧ $\Longrightarrow$ *attach conv res* = *attach conv' res*
⟨*proof*⟩

lemma *eq-$\mathcal{I}$-comp-cong*:
⟦ $\mathcal{I}1, \mathcal{I}2 \vdash_C$ *conv1* $\sim$ *conv1'*; $\mathcal{I}2, \mathcal{I}3 \vdash_C$ *conv2* $\sim$ *conv2'* ⟧
$\Longrightarrow$ $\mathcal{I}1, \mathcal{I}3 \vdash_C$ *comp-converter conv1 conv2* $\sim$ *comp-converter conv1' conv2'*
⟨*proof*⟩

lemma *comp-converter-cong*: *comp-converter conv1 conv2* = *comp-converter conv1' conv2'*
if *$\mathcal{I}$-full*, $\mathcal{I} \vdash_C$ *conv1* $\sim$ *conv1'* $\mathcal{I}$, *$\mathcal{I}$-full* $\vdash_C$ *conv2* $\sim$ *conv2'*
⟨*proof*⟩

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

lemma *parallel-converter2-eq-$\mathcal{I}$-cong*:
$[\![\mathcal{I}1, \mathcal{I}1' \vdash_C$ *conv1* $\sim$ *conv1'*; $\mathcal{I}2, \mathcal{I}2' \vdash_C$ *conv2* $\sim$ *conv2'* $]\!]$
$\Longrightarrow \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C$ *parallel-converter2 conv1 conv2* $\sim$ *parallel-converter2 conv1' conv2'*
⟨*proof*⟩

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

lemma *eq-$\mathcal{I}$-converterD-WT1*:
assumes $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\sim$ *conv2* **and** $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\surd$
shows $\mathcal{I},\mathcal{I}' \vdash_C$ *conv2* $\surd$
⟨*proof*⟩

lemma *eq-$\mathcal{I}$-converterD-WT*:
assumes $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\sim$ *conv2*
shows $\mathcal{I},\mathcal{I}' \vdash_C$ *conv1* $\surd \longleftrightarrow \mathcal{I},\mathcal{I}' \vdash_C$ *conv2* $\surd$
⟨*proof*⟩

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

lemma *eq-$\mathcal{I}$-restrict-gpv*:
assumes *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv'*
shows *eq-$\mathcal{I}$-gpv A $\mathcal{I}$* (*restrict-gpv $\mathcal{I}$ gpv*) *gpv'*
⟨*proof*⟩

lemma *eq-$\mathcal{I}$-restrict-converter*:
assumes $\mathcal{I},\mathcal{I}' \vdash_C$ *cnv* $\surd$
and *outs-$\mathcal{I}$* $\mathcal{I} \subseteq A$
shows $\mathcal{I},\mathcal{I}' \vdash_C$ *restrict-converter A $\mathcal{I}'$ cnv* $\sim$ *cnv*
⟨*proof*⟩

lemma *eq-$\mathcal{I}$-restrict-gpv-full*:
eq-$\mathcal{I}$-gpv A $\mathcal{I}$-full (*restrict-gpv $\mathcal{I}$ gpv*) (*restrict-gpv $\mathcal{I}$ gpv'*)
if *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv'*
⟨*proof*⟩

lemma *eq-$\mathcal{I}$-restrict-converter-cong*:
assumes $\mathcal{I},\mathcal{I}' \vdash_C$ *cnv* $\sim$ *cnv'*
and $A \subseteq$ *outs-$\mathcal{I}$ $\mathcal{I}$*
shows *restrict-converter A $\mathcal{I}'$ cnv* = *restrict-converter A $\mathcal{I}'$ cnv'*
⟨*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 ⟷
  (∀*xs. set xs* ⊆ *A* × *UNIV* ⟶ (∀*x* ∈ *A. trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*))

abbreviation *trace-callee-eq'* :: *'a set* $\Rightarrow$ $('a, 'b, 's1)$ *callee* $\Rightarrow$ *'s1* $\Rightarrow$ $('a, 'b, 's2)$ *callee* $\Rightarrow$ *'s2* $\Rightarrow$ *bool*
  (‹- $\vdash_C$/ (-'((-)')) ≈/ (-'((-)'))› [*90*, *0*, *0*, *0*, *0*] *91*)
  where *trace-callee-eq' A callee1 s1 callee2 s2* ≡ *trace-callee-eq callee1 callee2 A* (*return-spmf s1*) (*return-spmf s2*)

lemma *trace-callee-eqI*:
  assumes ⋀*xs x*. ⟦ *set xs* ⊆ *A* × *UNIV*; *x* ∈ *A* ⟧
    ⟹ *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*
  shows *trace-callee-eq callee1 callee2 A p q*
  ⟨*proof*⟩

lemma *trace-callee-eqD*:
  assumes *trace-callee-eq callee1 callee2 A p q*
    and *set xs* ⊆ *A* × *UNIV x* ∈ *A*
  shows *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*
  ⟨*proof*⟩

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

lemma *trace-callee-None* [*simp*]:
  trace-callee callee (*return-pmf None*) *xs x* = *return-pmf None*
  ⟨*proof*⟩

proposition *trace'-eqI-sim*:
  fixes *callee1* :: $('a, 'b, 's1)$ *callee* **and** *callee2* :: $('a, 'b, 's2)$ *callee*
  assumes *start*: *S p q*
    and *step*: ⋀*p q a*. ⟦ *S p q*; *a* ∈ *A* ⟧ ⟹
      bind-spmf p (λ*s. map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ*s. map-spmf fst* (*callee2 s a*))
    and *sim*: ⋀*p q a res b s'*. ⟦ *S p q*; *a* ∈ *A*; *res* ∈ *set-spmf q*; (*b*, *s'*) ∈ *set-spmf*

(*callee2 res a*) ⟧
  $\Longrightarrow$ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s. callee1 s a*)) *b*)
    (*cond-spmf-fst* (*bind-spmf q* (λ*s. callee2 s a*)) *b*)
 shows *trace-callee-eq callee1 callee2 A p q*
⟨*proof*⟩

fun *trace-callee-aux* :: (′*a*, ′*b*, ′*s*) *callee* ⇒ ′*s spmf* ⇒ (′*a* × ′*b*) *list* ⇒ ′*s spmf*
where
 trace-callee-aux callee p [] = *p*
| *trace-callee-aux callee p* ((*x*, *y*) # *xs*) = *trace-callee-aux callee* (*cond-spmf-fst* (*bind-spmf p* (λ*res. callee res x*)) *y*) *xs*

lemma *trace-callee-conv-trace-callee-aux*:
 trace-callee callee p xs a = *bind-spmf* (*trace-callee-aux callee p xs*) (λ*s. map-spmf fst* (*callee s a*))
 ⟨*proof*⟩

lemma *trace-callee-aux-append*:
 trace-callee-aux callee p (*xs* @ *ys*) = *trace-callee-aux callee* (*trace-callee-aux callee p xs*) *ys*
 ⟨*proof*⟩

inductive *trace-callee-closure* :: (′*a*, ′*b*, ′*s1*) *callee* ⇒ (′*a*, ′*b*, ′*s2*) *callee* ⇒ ′*a set* ⇒ ′*s1 spmf* ⇒ ′*s2 spmf* ⇒ ′*s1 spmf* ⇒ ′*s2 spmf* ⇒ *bool*
 for *callee1 callee2 A p q* **where**
 trace-callee-closure callee1 callee2 A p q (*trace-callee-aux callee1 p xs*) (*trace-callee-aux callee2 q xs*) **if** *set xs* ⊆ *A* × *UNIV*

lemma *trace-callee-closure-start*: *trace-callee-closure callee1 callee2 A p q p q*
 ⟨*proof*⟩

lemma *trace-callee-closure-step*:
 assumes *trace-callee-eq callee1 callee2 A p q*
  and *trace-callee-closure callee1 callee2 A p q p′ q′*
  and *a* ∈ *A*
 shows *bind-spmf p′* (λ*s. map-spmf fst* (*callee1 s a*)) = *bind-spmf q′* (λ*s. map-spmf fst* (*callee2 s a*))
⟨*proof*⟩

lemma *trace-callee-closure-sim*:
 assumes *trace-callee-closure callee1 callee2 A p q p′ q′*
  and *a* ∈ *A*
 shows *trace-callee-closure callee1 callee2 A p q*
  (*cond-spmf-fst* (*bind-spmf p′* (λ*s. callee1 s a*)) *b*)
  (*cond-spmf-fst* (*bind-spmf q′* (λ*s. callee2 s a*)) *b*)
 ⟨*proof*⟩

proposition *trace-callee-eq-complete*:
 assumes *trace-callee-eq callee1 callee2 A p q*

obtains S
where $S\ p\ q$
and $\bigwedge p\ q\ a.\ [\![\ S\ p\ q;\ a \in A\ ]\!] \Longrightarrow$
bind-spmf p $(\lambda s.$ *map-spmf fst* $($*callee1* $s\ a)) =$ *bind-spmf* q $(\lambda s.$ *map-spmf fst* $($*callee2* $s\ a))$
and $\bigwedge p\ q\ a\ s\ b\ s'.\ [\![\ S\ p\ q;\ a \in A;\ s \in$ *set-spmf* q; $(b, s') \in$ *set-spmf* $($*callee2* $s\ a)\ ]\!]$
$\Longrightarrow S$ $($*cond-spmf-fst* $($*bind-spmf* p $(\lambda s.$ *callee1* $s\ a))\ b)$
$($*cond-spmf-fst* $($*bind-spmf* q $(\lambda s.$ *callee2* $s\ a))\ b)$
$\langle proof \rangle$

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

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

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

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

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

lemma *trace-callee-resource-of-oracle2*:

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

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*]:
$A \vdash_R$ *resource-of-oracle callee1 s1* ≈ *resource-of-oracle callee2 s2* =
$A \vdash_C$ *callee1*(*s1*) ≈ *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* ⇒ (*′a*, *′b*) *resource* ⇒ *bool spmf* **where**
connect d res = *run-gpv run-resource d res*

definition *absorb* :: (*′a*, *′b*) *distinguisher* ⇒ (*′a*, *′b*, *′out*, *′in*) *converter* ⇒ (*′out*, *′in*) *distinguisher* **where**
absorb d conv = *map-gpv fst id* (*inline run-converter d conv*)

lemma *distinguish-attach*: *connect d* (*attach conv res*) = *connect* (*absorb d conv*) *res*
⟨*proof*⟩

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*: $A \vdash_R$ *res1* ≈ *res2*
and *WT*: $\mathcal{I} \vdash g$ *d* √
and *out*: *outs-gpv* $\mathcal{I}$ *d* ⊆ *A*
and *WT1*: $\mathcal{I} \vdash$*res res1* √
and *WT2*: $\mathcal{I} \vdash$*res res2* √
shows *connect d res1* = *connect d res2*
⟨*proof*⟩

lemma *distinguish-trace-eq*:
  assumes *distinguish*: $\bigwedge$*distinguisher*. $\mathcal{I} \vdash g$ *distinguisher* $\surd \Longrightarrow$ *connect distinguisher res* = *connect distinguisher res'*
    and *WT1*: $\mathcal{I} \vdash res$ *res1* $\surd$
    and *WT2*: $\mathcal{I} \vdash res$ *res2* $\surd$
  shows *outs-$\mathcal{I}$* $\mathcal{I} \vdash_R$ *res* $\approx$ *res'*
⟨*proof*⟩

lemma *connect-eq-resource-cong*:
  assumes $\mathcal{I} \vdash g$ *distinguisher* $\surd$
    and *outs-$\mathcal{I}$* $\mathcal{I} \vdash_R$ *res* $\sim$ *res'*
    and $\mathcal{I} \vdash res$ *res* $\surd$
  shows *connect distinguisher res* = *connect distinguisher res'*
  ⟨*proof*⟩

lemma *WT-gpv-absorb* [*WT-intro*]:
  $[\![\mathcal{I}' \vdash g$ *gpv* $\surd$; $\mathcal{I}', \mathcal{I} \vdash_C$ *conv* $\surd]\!] \Longrightarrow \mathcal{I} \vdash g$ *absorb gpv conv* $\surd$
  ⟨*proof*⟩

lemma *plossless-gpv-absorb* [*plossless-intro*]:
  assumes *gpv*: *plossless-gpv* $\mathcal{I}'$ *gpv*
    and *conv*: *plossless-converter* $\mathcal{I}'$ $\mathcal{I}$ *conv*
    and [*WT-intro*]: $\mathcal{I}' \vdash g$ *gpv* $\surd$ $\mathcal{I}', \mathcal{I} \vdash_C$ *conv* $\surd$
  shows *plossless-gpv* $\mathcal{I}$ (*absorb gpv conv*)
  ⟨*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** ‹∥› *50*)
no-notation *plus-oracle* (**infix** ‹$\oplus_O$› *500*)

notation *Resource* (‹§*R*§›)

notation *Converter* (‹§*C*§›)

alias *RES* = *resource-of-oracle*
alias *CNV* = *converter-of-callee*

alias *id-intercept* = *id-oracle*
notation *id-oracle* (‹1_I›)

notation *plus-oracle* (**infixr** ‹$\oplus_O$› *504*)
notation *parallel-oracle* (**infixr** ‹$\ddagger_O$› *504*)

notation *plus-intercept* (**infixr** ‹$\oplus_I$› *504*)
notation *parallel-intercept* (**infixr** ‹$\ddagger_I$› *504*)

notation *parallel-resource* (**infixr** ‹$\parallel$› *501*)

notation *parallel-converter* (**infixr** ‹$|_\propto$› *501*)
notation *parallel-converter2* (**infixr** ‹$|_=$› *501*)
notation *comp-converter* (**infixr** ‹$\odot$› *502*)

notation *fail-converter* (‹$\bot_C$›)
notation *id-converter* (‹1_C›)

notation *attach* (**infixr** ‹$\rhd$› *500*)

6.2 Wiring primitives

primrec *swap-sum* :: $'a + 'b \Rightarrow 'b + 'a$ **where**
  swap-sum (*Inl x*) = *Inr x*
| *swap-sum* (*Inr y*) = *Inl y*

definition $swap_C$ :: $('a + 'b, 'c + 'd, 'b + 'a, 'd + 'c)$ *converter* **where**
  $swap_C$ = *map-converter swap-sum swap-sum id id* 1_C

definition $rassocl_C$ :: $('a + ('b + 'c), 'd + ('e + 'f), ('a + 'b) + 'c, ('d + 'e) + 'f)$ *converter* **where**
  $rassocl_C$ = *map-converter lsumr rsuml id id* 1_C
definition $lassocr_C$ :: $(('a + 'b) + 'c, ('d + 'e) + 'f, 'a + ('b + 'c), 'd + ('e + 'f))$ *converter* **where**
  $lassocr_C$ = *map-converter rsuml lsumr id id* 1_C

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

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

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

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

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

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

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

definition *parallel-resource1-wiring* :: $('a + ('b + 'c), 'd + ('e + 'f), 'b + ('a + 'c), 'e + ('d + 'f))$ *converter* **where**
parallel-resource1-wiring = *swap-lassocr*

lemma *WT-parallel-resource1-wiring* [*WT-intro*]: $\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3)$, $\mathcal{I}2 \oplus_{\mathcal{I}} (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}3) \vdash_C$ *parallel-resource1-wiring* $\surd$
⟨*proof*⟩

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

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

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

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

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

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

lemmas *comp-lassocr*$_C{}'$ = *comp-converter-eqs*[*OF comp-lassocr*$_C$]

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

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

lemmas *comp-rassocl*$_C{}'$ = *comp-converter-eqs*[*OF comp-rassocl*$_C$]

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

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

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

map-converter swap-sum swap-sum id id $(conv2 \mid_= conv1)$
⟨*proof*⟩

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

lemmas *comp-swap*$_C{}'$ = *comp-converter-eqs*[*OF comp-swap*$_C$]

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

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

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

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

lemma *attach-converter-of-resource-conv-parallel-resource*:
converter-of-resource res $\mid_\propto 1_C \rhd res' = res \parallel res'$
⟨*proof*⟩

lemma *attach-converter-of-resource-conv-parallel-resource2*:
$1_C \mid_\propto$ *converter-of-resource res* $\rhd res' = res' \parallel res$
⟨*proof*⟩

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

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

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

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

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

lemma *run-converter-swap-lassocr* [*simp*]:
run-converter swap-lassocr x = *Pause* (*lassocr-swap-sum x*) (
case lsumr x of Inl - ⇒ (λ*y. case lsumr y of Inl -* ⇒ *Done* (*lassocr-swap-sum y*, *swap-lassocr*) | *-* ⇒ *Fail*)
| *Inr -* ⇒ (λ*y. case lsumr y of Inl -* ⇒ *Fail* | *Inr -* ⇒ *Done* (*lassocr-swap-sum y*, *swap-lassocr*)))
⟨*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 - ⇒ (λ*y. case rsuml y of Inl -* ⇒ *Done* (*parallel-sum-wiring y*, *parallel-wiring*) | *-* ⇒ *Fail*)
| *Inr x* ⇒ (*case lsumr x of Inl -* ⇒ (λ*y. case rsuml y of Inl -* ⇒ *Fail* | *Inr x* ⇒ (*case lsumr x of Inl -* ⇒ *Done* (*parallel-sum-wiring y*, *parallel-wiring*) | *Inr -* ⇒ *Fail*))
| *Inr -* ⇒ (λ*y. case rsuml y of Inl -* ⇒ *Fail* | *Inr x* ⇒ (*case lsumr x of Inl -* ⇒ *Fail* | *Inr -* ⇒ *Done* (*parallel-sum-wiring y*, *parallel-wiring*)))))
⟨*proof*⟩

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

lemma *bound-rassocl*$_C$ [*interaction-bound*]: *interaction-any-bounded-converter rassocl*$_C$ *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 $('a, 'b, 'c, 'd)$ *wiring* $= ('a \Rightarrow 'c) \times ('d \Rightarrow 'b)$

inductive *wiring* :: $('a, 'b)\ \mathcal{I} \Rightarrow ('c, 'd)\ \mathcal{I} \Rightarrow ('a, 'b, 'c, 'd)$ *converter* $\Rightarrow ('a, 'b, 'c, 'd)$ *wiring* $\Rightarrow$ *bool*
for $\mathcal{I}\ \mathcal{I}'$ *cnv*
where
wiring:
wiring $\mathcal{I}\ \mathcal{I}'$ *cnv* (f, g) **if**
$\mathcal{I},\mathcal{I}' \vdash_C$ *cnv* $\sim$ *map-converter id id f g* 1_C
$\mathcal{I},\mathcal{I}' \vdash_C$ *cnv* $\surd$

lemmas *wiringI* = *wiring*
hide-fact *wiring*

lemma *wiringD*:
assumes *wiring* $\mathcal{I}\ \mathcal{I}'$ *cnv* (f, g)
shows *wiringD-eq*: $\mathcal{I},\mathcal{I}' \vdash_C$ *cnv* $\sim$ *map-converter id id f g* 1_C
and *wiringD-WT*: $\mathcal{I},\mathcal{I}' \vdash_C$ *cnv* $\surd$
⟨*proof*⟩

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

definition *apply-wiring* :: $('a, 'b, 'c, 'd)$ *wiring* $\Rightarrow ('s, 'c, 'd)$ *oracle'* $\Rightarrow ('s, 'a, 'b)$ *oracle'*
where *apply-wiring* = $(\lambda(f, g).$ *map-fun id* (*map-fun f* (*map-spmf* (*map-prod g id*))))

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

lemma *attach-wiring-resource-of-oracle*:
assumes *wiring*: *wiring* $\mathcal{I}1\ \mathcal{I}2$ *conv fg*
and *WT*: $\mathcal{I}2 \vdash$*res RES res s* $\surd$
and *outs*: *outs*-$\mathcal{I}\ \mathcal{I}1$ = *UNIV*
shows *conv* $\rhd$ *RES res s* = *RES* (*apply-wiring fg res*) *s*
⟨*proof*⟩

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

lemma *apply-wiring-id* [*simp*]: *apply-wiring* (*id*, *id*) *res* = *res*
⟨*proof*⟩

definition *attach-wiring* :: $('a, 'b, 'c, 'd)$ *wiring* $\Rightarrow ('s \Rightarrow 'c \Rightarrow ('d \times 's, 'e, 'f)$ *gpv*$) \Rightarrow ('s \Rightarrow 'a \Rightarrow ('b \times 's, 'e, 'f)$ *gpv*$)$
where *attach-wiring* = $(\lambda(f, g).$ *map-fun id* (*map-fun f* (*map-gpv* (*map-prod g id*) *id*)))

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

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

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

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

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

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

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

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

lemma *apply-parallel2* [*simp*]:
apply-wiring (*fg* $|_w$ *fg'*) (*res1* $\oplus_O$ *res2*) = (*apply-wiring fg res1* $\oplus_O$ *apply-wiring fg' res2*)
⟨*proof*⟩

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

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

where $lassocr_w = (rsuml,\ lsumr)$

definition $rassocl_w$ :: $('a + ('b + 'c),\ 'd + ('e + 'f),\ ('a + 'b) + 'c,\ ('d + 'e) + 'f)$ *wiring*
where $rassocl_w = (lsumr,\ rsuml)$

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

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

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

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

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

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

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

end

7 Security

theory *Constructive-Cryptography* **imports**
Wiring
begin

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

locale *constructive-security-aux* =
fixes *real-resource* :: *security* ⇒ $('a + 'e,\ 'b + 'f)$ *resource*
and *ideal-resource* :: *security* ⇒ $('c + 'e,\ 'd + 'f)$ *resource*
and *sim* :: *security* ⇒ $('a,\ 'b,\ 'c,\ 'd)$ *converter*
and $\mathcal{I}$-*real* :: *security* ⇒ $('a,\ 'b)\ \mathcal{I}$
and $\mathcal{I}$-*ideal* :: *security* ⇒ $('c,\ 'd)\ \mathcal{I}$

and $\mathcal{I}$*-common* :: *security* $\Rightarrow$ $('e, 'f)$ $\mathcal{I}$
and *bound* :: *security* $\Rightarrow$ *enat*
and *lossless* :: *bool*
assumes *WT-real* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res real-resource* η $\surd$
and *WT-ideal* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res ideal-resource* η $\surd$
and *WT-sim* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$*-real* η, $\mathcal{I}$*-ideal* η $\vdash_C$ *sim* η $\surd$
and *adv*: $\bigwedge\mathcal{A}$ :: *security* $\Rightarrow$ $('a + 'e, 'b + 'f)$ *distinguisher*.
$\llbracket$ $\bigwedge\eta$. $\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*g* $\mathcal{A}$ η $\surd$;
$\bigwedge\eta$. *interaction-bounded-by* (λ-. *True*) ($\mathcal{A}$ η) (*bound* η);
$\bigwedge\eta$. *lossless* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-real* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{A}$ η) $\rrbracket$
$\Longrightarrow$ *negligible* ($\lambda\eta$. *advantage* ($\mathcal{A}$ η) (*sim* η $|_=$ 1_C $\rhd$ *ideal-resource* η) (*real-resource* η))

locale *constructive-security* =
constructive-security-aux real-resource ideal-resource sim $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common bound lossless*
for *real-resource* :: *security* $\Rightarrow$ $('a + 'e, 'b + 'f)$ *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*: $\exists$ *cnv*. $\forall\mathcal{D}$ :: *security* $\Rightarrow$ $('c + 'e, 'd + 'f)$ *distinguisher*.
($\forall\eta$. $\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*g* $\mathcal{D}$ η $\surd$)
$\longrightarrow$ ($\forall\eta$. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound* η))
$\longrightarrow$ ($\forall\eta$. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η))
$\longrightarrow$ ($\forall\eta$. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η)) $\wedge$
negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))

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

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

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

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

theorem (**in** *constructive-security*) *lifting*:
assumes *WT-conv* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$*-common'* η, $\mathcal{I}$*-common* $\eta \vdash_C$ *conv* η $\surd$
and *bound* [*interaction-bound*]: $\bigwedge\eta$. *interaction-any-bounded-converter* (*conv* η) (*bound-conv* η)
and *bound-le*: $\bigwedge\eta$. *bound'* η $*$ *max* (*bound-conv* η) *1* $\leq$ *bound* η
and *lossless* [*plossless-intro*]: $\bigwedge\eta$. *lossless* $\Longrightarrow$ *plossless-converter* ($\mathcal{I}$*-common'* η) ($\mathcal{I}$*-common* η) (*conv* η)
shows *constructive-security*
($\lambda\eta$. 1_C $|_=$ *conv* $\eta \rhd$ *real-resource* η) ($\lambda\eta$. 1_C $|_=$ *conv* $\eta \rhd$ *ideal-resource* η) *sim*
$\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common' bound' lossless w*
⟨*proof*⟩

theorem *constructive-security-trivial*:
fixes *res*

assumes [*WT-intro*]: $\bigwedge\eta.\ \mathcal{I}\ \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-}common\ \eta \vdash res\ res\ \eta \surd$
shows *constructive-security res res* (λ-. 1_C) $\mathcal{I}$ $\mathcal{I}$ $\mathcal{I}$*-common bound lossless* (λ-. (*id*, *id*))
⟨*proof*⟩

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

and *lossless-real1* [*plossless-intro*]: $\bigwedge\eta.$ *lossless2* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$*-real1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common1* η) (*real1* η)
and *lossless-sim2* [*plossless-intro*]: $\bigwedge\eta.$ *lossless1* $\Longrightarrow$ *plossless-converter* ($\mathcal{I}$*-real2* η) ($\mathcal{I}$*-inner2* η) (*sim2* η)
and *lossless-ideal2* [*plossless-intro*]: $\bigwedge\eta.$ *lossless1* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$*-inner2* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) (*ideal2* η)
shows *constructive-security* ($\lambda\eta.$ *parallel-wiring* $\rhd$ *real1* $\eta \parallel$ *real2* η) ($\lambda\eta.$ *parallel-wiring* $\rhd$ *ideal1* $\eta \parallel$ *ideal2* η) ($\lambda\eta.$ *sim1* $\eta\ |_=$ *sim2* η)
($\lambda\eta.$ $\mathcal{I}$*-real1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) ($\lambda\eta.$ $\mathcal{I}$*-inner1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) ($\lambda\eta.$ $\mathcal{I}$*-common1* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η)
($\lambda\eta.$ *min* (*bound1* η) (*bound2* η)) (*lossless1* $\vee$ *lossless2*) ($\lambda\eta.$ *w1* $\eta\ |_w$ *w2* η)
⟨*proof*⟩

theorem (**in** *constructive-security*) *parallel-realisation1*:
assumes *WT-res*: $\bigwedge\eta.$ $\mathcal{I}$*-res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common′* $\eta \vdash res\ res\ \eta \surd$
and *lossless-res*: $\bigwedge\eta.$ *lossless* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$*-res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common′* η) (*res* η)
shows *constructive-security* ($\lambda\eta.$ *parallel-wiring* $\rhd$ *res* $\eta \parallel$ *real-resource* η)
($\lambda\eta.$ *parallel-wiring* $\rhd$ (*res* $\eta \parallel$ *ideal-resource* η)) ($\lambda\eta.$ *parallel-converter2 id-converter* (*sim* η))
($\lambda\eta.$ $\mathcal{I}$*-res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-real* η) ($\lambda\eta.$ $\mathcal{I}$*-res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-ideal* η) ($\lambda\eta.$ $\mathcal{I}$*-common′* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) *bound lossless* ($\lambda\eta.$ (*id*, *id*) $|_w$ *w* η)
⟨*proof*⟩

theorem (**in** *constructive-security*) *parallel-realisation2*:
assumes *WT-res*: $\bigwedge\eta.$ $\mathcal{I}$*-res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common′* $\eta \vdash res\ res\ \eta \surd$
and *lossless-res*: $\bigwedge\eta.$ *lossless* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$*-res* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common′* η) (*res* η)
shows *constructive-security* ($\lambda\eta.$ *parallel-wiring* $\rhd$ *real-resource* $\eta \parallel$ *res* η)
($\lambda\eta.$ *parallel-wiring* $\rhd$ (*ideal-resource* $\eta \parallel$ *res* η)) ($\lambda\eta.$ *parallel-converter2* (*sim* η) *id-converter*)
($\lambda\eta.$ $\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-res* η) ($\lambda\eta.$ $\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-res* η) ($\lambda\eta.$ $\mathcal{I}$*-common* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common′* η) *bound lossless* ($\lambda\eta.$ *w* $\eta\ |_w$ (*id*, *id*))
⟨*proof*⟩

theorem (**in** *constructive-security*) *parallel-resource1*:
assumes *WT-res* [*WT-intro*]: $\bigwedge\eta.$ $\mathcal{I}$*-res* $\eta \vdash res\ res\ \eta \surd$
and *lossless-res* [*plossless-intro*]: $\bigwedge\eta.$ *lossless* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$*-res* η) (*res*

η)
shows *constructive-security* ($\lambda\eta$. *parallel-resource1-wiring* $\rhd$ *res* η $\|$ *real-resource* η)
($\lambda\eta$. *parallel-resource1-wiring* $\rhd$ *res* η $\|$ *ideal-resource* η) *sim*
$\mathcal{I}$*-real* $\mathcal{I}$*-ideal* ($\lambda\eta$. $\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) *bound lossless w*
⟨*proof*⟩

end

8 Examples

theory *System-Construction* **imports**
../../*Constructive-Cryptography*
begin

8.1 Random oracle resource

locale *rorc* =
fixes *range* :: *'r set*
begin

fun *rnd-oracle* :: (*'m* ⇒ *'r option*, *'m*, *'r*) *oracle'* **where**
rnd-oracle f m = (*case f m of*
(*Some r*) ⇒ *return-spmf* (*r*, *f*)
| *None* ⇒ *do* {
r ← *spmf-of-set* (*range*);
return-spmf (*r*, *f*(*m* := *Some r*))})

definition *res* = *RES* (*rnd-oracle* $\oplus_O$ *rnd-oracle*) *Map.empty*

end

8.2 Key resource

locale *key* =
fixes *key-gen* :: *'k spmf*
begin

fun *key-oracle* :: (*'k option*, *unit*, *'k*) *oracle'* **where**
key-oracle None () = *do* { *k* ← *key-gen*; *return-spmf* (*k*, *Some k*)}
| *key-oracle* (*Some x*) () = *return-spmf* (*x*, *Some x*)

definition *res* = *RES* (*key-oracle* $\oplus_O$ *key-oracle*) *None*

end

8.3 Channel resource

datatype *'a cstate* = *Void* | *Fail* | *Store 'a* | *Collect 'a*

datatype *'a aquery* = *Look* | *ForwardOrEdit* (*forward-or-edit*: *'a*) | *Drop*
type-synonym *'a insec-query* = *'a option aquery*
type-synonym *auth-query* = *unit aquery*

consts *Forward* :: *'a aquery*
abbreviation *Forward-auth* :: *auth-query* **where** *Forward-auth* ≡ *ForwardOrEdit* ()
abbreviation *Forward-insec* :: *'a insec-query* **where** *Forward-insec* ≡ *ForwardOrEdit None*
abbreviation *Edit* :: *'a* ⇒ *'a insec-query* **where** *Edit m* ≡ *ForwardOrEdit* (*Some m*)
adhoc-overloading *Forward* ⇌ *Forward-auth*
adhoc-overloading *Forward* ⇌ *Forward-insec*

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

8.3.1 Generic channel

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

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

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

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

end

8.3.2 Insecure channel

locale *insec-channel*
begin

fun *insec-oracle* :: (*'m cstate*, *'m insec-query*, *'m option*) *oracle'* **where**
insec-oracle Void (*Edit m'*) = *return-spmf* (*None*, *Collect m'*)
| *insec-oracle* (*Store m*) (*Edit m'*) = *return-spmf* (*None*, *Collect m'*)
| *insec-oracle* (*Store m*) *Forward* = *return-spmf* (*None*, *Collect m*)

| *insec-oracle* (*Store m*) *Drop* = *return-spmf* (*None*, *Fail*)
| *insec-oracle* (*Store m*) *Look* = *return-spmf* (*Some m*, *Store m*)
| *insec-oracle s* - = *return-spmf* (*None*, *s*)

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

end

8.3.3 Authenticated channel

locale *auth-channel*
begin

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

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

end

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

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

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

8.3.4 Secure channel

locale *sec-channel*
begin

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

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

end

abbreviation (*input*) *auth-query-of* :: *auth-query* ⇒ *auth-query*
where *auth-query-of* ≡ *id*

abbreviation (*input*) *sec-response-of* :: *'a list option* ⇒ *nat option*
where *sec-response-of* ≡ *map-option length*

abbreviation *auth-sec-wiring* :: (*auth-query*, *nat option*, *auth-query*, *'a list option*) *wiring*
where *auth-sec-wiring* ≡ (*auth-query-of*, *sec-response-of*)

8.4 Cipher converter

locale *cipher* =
AUTH: *auth-channel* + *KEY*: *key key-alg*
for *key-alg* :: *'k spmf* +
fixes *enc-alg* :: *'k* ⇒ *'m* ⇒ *'c spmf*
and *dec-alg* :: *'k* ⇒ *'c* ⇒ *'m option*
begin

definition *enc* :: (*'m*, *unit*, *unit* + *'c*, *'k* + *unit*) *converter* **where**
enc ≡ *CNV* (*stateless-callee* (λ*m*. *do* {
k ← *Pause* (*Inl* ()) *Done*;
c ← *lift-spmf* (*enc-alg* (*projl k*) *m*);
(- :: *'k* + *unit*) ← *Pause* (*Inr c*) *Done*;
Done (())
})) ()

definition *dec* :: (*unit*, *'m option*, *unit* + *unit*, *'k* + *'c option*) *converter* **where**
dec ≡ *CNV* (*stateless-callee* (λ-. *Pause* (*Inr* ()) (λ*c'*.
case c' of Inr (*Some c*) ⇒ (*do* {
k ← *Pause* (*Inl* ()) *Done*;
Done (*dec-alg* (*projl k*) *c*) })
| - ⇒ *Done None*)
)) ()

definition πE :: (*auth-query*, *'c option*, *auth-query*, *'c option*) *converter* (‹π^E›)
where
$\pi^E \equiv 1_C$

definition *routing* ≡ (1_C $|_=$ *lassocr*$_C$) ⊙ *swap-lassocr* ⊙ (1_C $|_=$ (1_C $|_=$ *swap-lassocr*) ⊙ *swap-lassocr*) ⊙ *rassocl*$_C$

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

lemma *res-alt-def*: *res* = ((1_C $|_=$ *enc* $|_=$ *dec*) ⊙ (1_C $|_=$ *parallel-wiring*)) ▷ *parallel-resource1-wiring* ▷ (*KEY*.*res* ∥ *AUTH*.*res*)
⟨*proof*⟩

end

8.5 Message authentication converter

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

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

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

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

definition *routing* ≡ (1_C $|_=$ $lassocr_C$) ⊙ *swap-lassocr* ⊙ (1_C $|_=$ (1_C $|_=$ *swap-lassocr*)
⊙ *swap-lassocr*) ⊙ $rassocl_C$

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

end

lemma *interface-wiring*:
(*cnv-advr* $|_=$ *cnv-send* $|_=$ *cnv-recv*) ▷ (1_C $|_=$ *parallel-wiring*) ▷ *parallel-resource1-wiring*
▷
(*RES* (*res2-send* $\oplus_O$ *res2-recv*) *res2-s* ∥ *RES* (*res1-advr* $\oplus_O$ *res1-send* $\oplus_O$ *res1-recv*)
res1-s)
=

cnv-advr $|_=$ *cnv-send* $|_=$ *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*)
⟨*proof*⟩

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

end

9 Security of one-time-pad encryption

theory *One-Time-Pad* **imports**
System-Construction
begin

definition *key* :: *security* ⇒ *bool list spmf* **where**
key η ≡ *spmf-of-set* (*nlists UNIV* η)

definition *enc* :: *security* ⇒ *bool list* ⇒ *bool list* ⇒ *bool list spmf* **where**
enc η *k m* ≡ *return-spmf* (*k* [⊕] *m*)

definition *dec* :: *security* ⇒ *bool list* ⇒ *bool list* ⇒ *bool list option* **where**
dec η *k c* ≡ *Some* (*k* [⊕] *c*)

definition *sim* :: *'b list option* ⇒ *'a* ⇒ (*'b list option* × *'b list option*, *'a*, *nat option*) *gpv* **where**
sim c q ≡ (*do* {
  lo ← *Pause q Done*;
  (*case lo of*
    Some n ⇒ *if c* = *None*
      then do {
        x ← *lift-spmf* (*spmf-of-set* (*nlists UNIV n*));
        Done (*Some x*, *Some x*)}
      else Done (*c*, *c*)
  | *None* ⇒ *Done* (*None*, *c*))})

context
fixes η :: *security*
begin

private definition *key-channel-send* :: *bool list option* × *bool list cstate*
⇒ *bool list* ⇒ (*unit* × *bool list option* × *bool list cstate*) *spmf* **where**
key-channel-send s m ≡ *do* {
  (*k*, *s*) ← (*key.key-oracle* (*key* η))$\dagger$ *s* ();
  c ← *enc* η *k m*;
  (-, *s*) ← $\dagger$*channel.send-oracle s c*;

return-spmf ((), *s*)}

private definition *key-channel-recv* :: *bool list option* × *bool list cstate*
⇒ *'a* ⇒ (*bool list option* × *bool list option* × *bool list cstate*) *spmf* **where**
key-channel-recv s m ≡*do* {
(*c*, *s*) ← †*channel.recv-oracle s* ();
(*case c of None* ⇒ *return-spmf* (*None*, *s*)
| *Some c'* ⇒ *do* {
(*k*, *s*) ← (*key.key-oracle* (*key* η))† *s* ();
return-spmf (*dec* η *k c'*, *s*)})}

private abbreviation *callee-sec-channel* **where**
callee-sec-channel callee ≡ *lift-state-oracle extend-state-oracle* (*attach-callee callee sec-channel.sec-oracle*)

private inductive *S* :: (*bool list option* × *unit* × *bool list cstate*) *spmf* ⇒
(*bool list option* × *bool list cstate*) *spmf* ⇒ *bool* **where**
S (*return-spmf* (*None*, (), *Void*))
(*return-spmf* (*None*, *Void*))
| *S* (*return-spmf* (*None*, (), *Store plain*))
(*map-spmf* (λ*key*. (*Some key*, *Store* (*key* [⊕] *plain*))) (*spmf-of-set* (*nlists UNIV* η)))
if *length plain* = *id'* η
| *S* (*return-spmf* (*None*, (), *Collect plain*))
(*map-spmf* (λ*key*. (*Some key*, *Collect* (*key* [⊕] *plain*))) (*spmf-of-set* (*nlists UNIV* η)))
if *length plain* = *id'* η
| *S* (*return-spmf* (*Some* (*key* [⊕] *plain*), (), *Store plain*))
(*return-spmf* (*Some key*, *Store* (*key* [⊕] *plain*)))
if *length plain* = *id'* η *length key* = *id'* η **for** *key*
| *S* (*return-spmf* (*Some* (*key* [⊕] *plain*), (), *Collect plain*))
(*return-spmf* (*Some key*, *Collect* (*key* [⊕] *plain*)))
if *length plain* = *id'* η *length key* = *id'* η **for** *key*
| *S* (*return-spmf* (*None*, (), *Fail*))
(*map-spmf* (λ*x*. (*Some x*, *Fail*)) (*spmf-of-set* (*nlists UNIV* η)))
| *S* (*return-spmf* (*Some* (*key* [⊕] *plain*), (), *Fail*))
(*return-spmf* (*Some key*, *Fail*))
if *length plain* = *id'* η *length key* = *id'* η **for** *key plain*

lemma *resources-indistinguishable*:
shows (*UNIV* <+> *nlists UNIV* (*id'* η) <+> *UNIV*) $\vdash_R$
RES (*callee-sec-channel sim* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*) (*None* :: *bool list option*, (), *Void*)
≈
RES (†*auth-channel.auth-oracle* $\oplus_O$ *key-channel-send* $\oplus_O$ *key-channel-recv*) (*None* :: *bool list option*, *Void*)
(**is** *?A* $\vdash_R$ *RES* (*?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*) *?SL* ≈ *RES* (*?R1* $\oplus_O$ *?R2* $\oplus_O$ *?R3*) *?SR*)

⟨*proof*⟩

lemma *real-resource-wiring*:
  shows *cipher.res* (*key* η) (*enc* η) (*dec* η)
    = *RES* (†*auth-channel.auth-oracle* $\oplus_O$ *key-channel-send* $\oplus_O$ *key-channel-recv*)
(*None*, *Void*)
  including *lifting-syntax*
⟨*proof*⟩

lemma *ideal-resource-wiring*:
  shows (*CNV callee s*) $|_=$ 1_C ▷ *channel.res sec-channel.sec-oracle*
   = *RES* (*callee-sec-channel callee* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*
) (*s*, (), *Void*) (**is** *?L1* ▷ - = *?R*)
⟨*proof*⟩

end

lemma *eq-$\mathcal{I}$-gpv-Done1*:
  eq-$\mathcal{I}$-gpv A $\mathcal{I}$ (*Done x*) *gpv* ⟷ *lossless-spmf* (*the-gpv gpv*) ∧ (∀ *a*∈*set-spmf*
(*the-gpv gpv*). *eq-$\mathcal{I}$-generat A $\mathcal{I}$* (*eq-$\mathcal{I}$-gpv A $\mathcal{I}$*) (*Pure x*) *a*)
  ⟨*proof*⟩

lemma *eq-$\mathcal{I}$-gpv-Done2*:
  eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv (*Done x*) ⟷ *lossless-spmf* (*the-gpv gpv*) ∧ (∀ *a*∈*set-spmf*
(*the-gpv gpv*). *eq-$\mathcal{I}$-generat A $\mathcal{I}$* (*eq-$\mathcal{I}$-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 *$\mathcal{I}$-real* ≡ $\lambda\eta$. *$\mathcal{I}$-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η))
   and *$\mathcal{I}$-ideal* ≡ $\lambda\eta$. *$\mathcal{I}$-uniform UNIV* {*None*, *Some* η}
   and *$\mathcal{I}$-common* ≡ $\lambda\eta$. *$\mathcal{I}$-uniform* (*nlists UNIV* η) *UNIV* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-uniform UNIV*
(*insert None* (*Some* ‘ *nlists UNIV* η))
  shows
   constructive-security2 CIPHER.res (λ-. *S-CHAN.res*) (λ-. *CNV sim None*)
    $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common (λ-. ∞) *False* (λ-. *auth-sec-wiring*)
⟨*proof*⟩

end

end

10 Security of message authentication

theory *Message-Authentication-Code* **imports**
  System-Construction

begin

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

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

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

fun *valid-mac-query* :: *security* ⇒ (*bool list* × *bool list*) *insec-query* ⇒ *bool* **where**
valid-mac-query η (*ForwardOrEdit* (*Some* (*a*, *m*))) ⟷ $a \in vld\ \eta \wedge m \in vld\ \eta$
| *valid-mac-query* η - = *True*

fun *sim* :: ('*b list* × '*b list*) *option* + *unit* ⇒ ('*b list* × '*b list*) *insec-query*
⇒ (('*b list* × '*b list*) *option* × (('*b list* × '*b list*) *option* + *unit*), *auth-query* , '*b list option*) *gpv* **where**
sim (*Inr* ()) - = *Done* (*None*, *Inr*())
| *sim* (*Inl None*) (*Edit* (*a*′, *m*′)) = *do* { - ← *Pause Drop Done*; *Done* (*None*, *Inr* ())}
| *sim* (*Inl* (*Some* (*a*, *m*))) (*Edit* (*a*′, *m*′)) = (*if* *a* = *a*′ ∧ *m* = *m*′
then do { - ← *Pause Forward Done*; *Done* (*None*, *Inl* (*Some* (*a*, *m*)))}
else do { - ← *Pause Drop Done*; *Done* (*None*, *Inr* ())})
| *sim* (*Inl None*) *Forward* = *do* {
Pause Forward Done;
Done (*None*, *Inl None*) }
| *sim* (*Inl* (*Some* -)) *Forward* = *do* {
Pause Forward Done;
Done (*None*, *Inr* ()) }
| *sim* (*Inl None*) *Drop* = *do* {
Pause Drop Done;
Done (*None*, *Inl None*) }
| *sim* (*Inl* (*Some* -)) *Drop* = *do* {
Pause Drop Done;
Done (*None*, *Inr* ()) }
| *sim* (*Inl* (*Some* (*a*, *m*))) *Look* = *do* {
lo ← *Pause Look Done*;
(*case lo of*
Some m ⇒ *Done* (*Some* (*a*, *m*), *Inl* (*Some* (*a*, *m*)))
| *None* ⇒ *Done* (*None*, *Inl* (*Some* (*a*, *m*))))}
| *sim* (*Inl None*) *Look* = *do* {
lo ← *Pause Look Done*;
(*case lo of*
Some m ⇒ *do* {
a ← *lift-spmf* (*spmf-of-set* (*nlists UNIV* (*length m*)));
Done (*Some* (*a*, *m*), *Inl* (*Some* (*a*, *m*)))}
| *None* ⇒ *Done* (*None*, *Inl None*))}

context
  fixes *η* :: *security*
begin

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

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

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

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

private fun *lazy-channel-recv* :: (*bool list cstate* × (*bool list* × *bool list*) *option* ×

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

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

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

private abbreviation *callee-auth-channel* **where**
callee-auth-channel callee ≡ *lift-state-oracle extend-state-oracle* (*attach-callee callee auth-channel.auth-oracle*)

private abbreviation
valid-insecQ ≡ {(*x* :: (*bool list* × *bool list*) *insec-query*). *case x of*
ForwardOrEdit (*Some* (*a*, *m*)) ⇒ *length a* = id' η ∧ *length m* = id' η
| - ⇒ *True*}

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

(*map-spmf* (λx. ((*True*, ()), [$m \mapsto$ *fst* x, $m' \mapsto$ *snd* x], *Fail*)) (*spmf-of-set* (*nlists* *UNIV* η $\times$ *nlists* *UNIV* η $\cap$ {x. *snd* $x \neq a'$})))
if *length* m = *id'* η **and** *length* m'= *id'* η
| *S* (*map-spmf* (λx. (*Fail*, *None*, *as*($m' \mapsto x$))) *spmf-s*)
(*map-spmf* (λp. ((*True*, ()), *as*($m' \mapsto$ *fst* p, $m \mapsto$ *snd* p), *Fail*)) (*mk-lossless* (*pair-spmf* *spmf-s* (*spmf-of-set* (*nlists* *UNIV* η)))))
if *length* m = *id'* η **and** *length* m'= *id'* η **and** *lossless-spmf* *spmf-s*

private lemma *trace-eq-lazy*:
assumes $\eta > 0$
shows (*valid-insecQ* <+> *nlists* *UNIV* (*id'* η) <+> *UNIV*) $\vdash_R$
RES (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*)
$\approx$
RES (††*insec-channel.insec-oracle* $\oplus_O$ *rorc-channel-send* $\oplus_O$ *rorc-channel-recv*) ((*False*, ()), *Map.empty*, *Void*)
(**is** *?A* $\vdash_R$ *RES* (*?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*) *?SL* $\approx$ *RES* (*?R1* $\oplus_O$ *?R2* $\oplus_O$ *?R3*) *?SR*)
⟨*proof*⟩ **lemma** *game-difference*:
defines $\mathcal{I} \equiv \mathcal{I}$*-uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert* *None* (*Some* ‘ (*nlists* *UNIV* η $\times$ *nlists* *UNIV* η))) $\oplus_\mathcal{I}$
($\mathcal{I}$*-uniform* (*vld* η) *UNIV* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* *UNIV* (*insert* *None* (*Some* ‘ *vld* η)))
assumes *bound*: *interaction-bounded-by'* (λ-. *True*) $\mathcal{A}$ q
and *lossless*: *plossless-gpv* $\mathcal{I}$ $\mathcal{A}$
and *WT*: $\mathcal{I} \vdash g$ $\mathcal{A}$ $\surd$
shows
|*spmf* (*connect* $\mathcal{A}$ (*RES* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv-f*) (*Void*, *None*, *Map.empty*))) *True* −
spmf (*connect* $\mathcal{A}$ (*RES* (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv*) (*Void*, *None*, *Map.empty*))) *True*|
$\leq q$ / *real* (2 ^ η) (**is** *?LHS* $\leq$ -)
⟨*proof*⟩ **inductive** S' :: (((*bool list* $\times$ *bool list*) *option* + *unit*) $\times$ *unit* $\times$ *bool list cstate*) *spmf* $\Rightarrow$
(*bool list cstate* $\times$ (*bool list* $\times$ *bool list*) *option* $\times$ (*bool list* $\Rightarrow$ *bool list option*)) *spmf* $\Rightarrow$ *bool* **where**
S' (*return-spmf* (*Inl None*, (), *Void*))
(*return-spmf* (*Void*, *None*, *Map.empty*))
| S' (*return-spmf* (*Inl None*, (), *Store* m))
(*return-spmf* (*Store* m, *None*, *Map.empty*))
if *length* m = *id'* η
| S' (*return-spmf* (*Inr* (), (), *Collect* m))
(*return-spmf* (*Collect* m, *None*, *Map.empty*))
if *length* m = *id'* η
| S' (*return-spmf* (*Inl* (*Some* (a, m)), (), *Store* m))
(*return-spmf* (*Store* m, *None*, [$m \mapsto a$]))
if *length* m = *id'* η
| S' (*return-spmf* (*Inr* (), (), *Collect* m))
(*return-spmf* (*Collect* m, *None*, [$m \mapsto a$]))

if $length\ m = id'\ \eta$
$|\ S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Fail))$
$\quad (return\text{-}spmf\ (Fail,\ None,\ Map.empty))$
$|\ S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Fail))$
$\quad (return\text{-}spmf\ (Fail,\ None,\ [m \mapsto x]))$
if $length\ m = id'\ \eta$
$|\ S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Void))$
$\quad (return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ Map.empty))$
if $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$
$|\ S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Fail))$
$\quad (return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ Map.empty))$
if $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$
$|\ S'\ (return\text{-}spmf\ (Inr\ (),\ (),\ Store\ m))$
$\quad (return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ Map.empty))$
if $length\ m = id'\ \eta$ **and** $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$
$|\ S'\ (return\text{-}spmf\ (Inl\ (Some\ (a',\ m')),\ (),\ Collect\ m'))$
$\quad (return\text{-}spmf\ (Collect\ m',\ Some\ (a',\ m'),\ [m' \mapsto a']))$
if $length\ m' = id'\ \eta$ **and** $length\ a' = id'\ \eta$

$|\ S'\ (return\text{-}spmf\ (Inl\ None,\ (),\ cstate.Collect\ m))$
$\quad (return\text{-}spmf\ (cstate.Collect\ m,\ None,\ Map.empty))$
if $length\ m = id'\ \eta$
$|\ S'\ (return\text{-}spmf\ (Inl\ None,\ (),\ cstate.Fail))$
$\quad (return\text{-}spmf\ (cstate.Fail,\ None,\ Map.empty))$

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

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

private lemma *trace-eq-sim*:
shows (*valid-insecQ* <+> *nlists UNIV* (*id'* η) <+> *UNIV*) $\vdash_R$
RES (*callee-auth-channel sim* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*)
(*Inl None*, (), *Void*)
$\approx$
RES (*lazy-channel-insec* $\oplus_O$ *lazy-channel-send* $\oplus_O$ *lazy-channel-recv-f*) (*Void*, *None*, *Map.empty*)
(**is** *?A* $\vdash_R$ *RES* (*?L1* $\oplus_O$ *?L2* $\oplus_O$ *?L3*) *?SL* $\approx$ *RES* (*?R1* $\oplus_O$ *?R2* $\oplus_O$ *?R3*) *?SR*)
⟨*proof*⟩ **lemma** *real-resource-wiring*: *macode.res* (*rnd* η) (*mac* η) =
RES (††*insec-channel.insec-oracle* $\oplus_O$ *rorc-channel-send* $\oplus_O$ *rorc-channel-recv*)
((*False*, ()), *Map.empty*, *Void*)
(**is** *?L* = *?R*) **including** *lifting-syntax*
⟨*proof*⟩ **lemma** *ideal-resource-wiring*: (*CNV callee s*) $|_=$ 1_C $\rhd$ *channel.res auth-channel.auth-oracle* =
RES (*callee-auth-channel callee* $\oplus_O$ ††*channel.send-oracle* $\oplus_O$ ††*channel.recv-oracle*) (*s*, (), *Void*) (**is** *?L1* $\rhd$ - = *?R*)
⟨*proof*⟩

lemma *all-together*:
defines $\mathcal{I} \equiv \mathcal{I}$-*uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert None* (*Some* ‘ (*nlists UNIV* η $\times$ *nlists UNIV* η))) $\oplus_{\mathcal{I}}$
($\mathcal{I}$-*uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*uniform UNIV* (*insert None* (*Some* ‘ *vld* η)))
assumes $\eta > 0$
and *interaction-bounded-by'* (λ-. *True*) ($\mathcal{A}$ η) *q*
and *lossless*: *plossless-gpv* $\mathcal{I}$ ($\mathcal{A}$ η)
and *WT*: $\mathcal{I} \vdash g\ \mathcal{A}\ \eta\ \surd$
shows
|*spmf* (*connect* ($\mathcal{A}$ η) (*CNV sim* (*Inl None*) $|_=$ 1_C $\rhd$ *channel.res auth-channel.auth-oracle*)) *True* −
spmf (*connect* ($\mathcal{A}$ η) (*macode.res* (*rnd* η) (*mac* η))) *True*| $\leq$ *q* / *real* (*2* ^ η)
⟨*proof*⟩

end

context begin
interpretation *MAC*: *macode rnd* η *mac* η **for** η ⟨*proof*⟩
interpretation *A-CHAN*: *auth-channel* ⟨*proof*⟩

lemma *WT-enm*:
$X \neq \{\} \Longrightarrow \mathcal{I}\textit{-uniform}\ (\textit{vld}\ \eta)\ \textit{UNIV}, \mathcal{I}\textit{-uniform}\ (\textit{vld}\ \eta)\ X \oplus_{\mathcal{I}} \mathcal{I}\textit{-uniform}\ (X \times \textit{vld}\ \eta)\ \textit{UNIV} \vdash_C \textit{MAC.enm}\ \eta\ \surd$
⟨*proof*⟩

lemma *WT-dem*: $\mathcal{I}\textit{-uniform UNIV}\ (\textit{insert None}\ (\textit{Some}\ `\ \textit{vld}\ \eta)), \mathcal{I}\textit{-full} \oplus_{\mathcal{I}} \mathcal{I}\textit{-uniform UNIV}\ (\textit{insert None}\ (\textit{Some}\ `\ (\textit{nlists UNIV}\ \eta \times \textit{nlists UNIV}\ \eta))) \vdash_C \textit{MAC.dem}\ \eta\ \surd$
⟨*proof*⟩

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

lemma *secure-mac*:
defines $\mathcal{I}\textit{-real} \equiv \lambda\eta.\ \mathcal{I}\textit{-uniform}\ \{x.\ \textit{valid-mac-query}\ \eta\ x\}\ (\textit{insert None}\ (\textit{Some}\ `\ (\textit{nlists UNIV}\ \eta \times \textit{nlists UNIV}\ \eta)))$
and $\mathcal{I}\textit{-ideal} \equiv \lambda\eta.\ \mathcal{I}\textit{-uniform UNIV}\ (\textit{insert None}\ (\textit{Some}\ `\ \textit{nlists UNIV}\ \eta))$
and $\mathcal{I}\textit{-common} \equiv \lambda\eta.\ \mathcal{I}\textit{-uniform}\ (\textit{vld}\ \eta)\ \textit{UNIV} \oplus_{\mathcal{I}} \mathcal{I}\textit{-uniform UNIV}\ (\textit{insert None}\ (\textit{Some}\ `\ \textit{vld}\ \eta))$
shows
constructive-security MAC.res (λ-. *A-CHAN.res*) (λ-. *CNV sim* (*Inl None*))
$\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* (λ-. *enat q*) *True* (λ-. *insec-auth-wiring*)
⟨*proof*⟩

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* η $\oplus_O$ *CIPHER.KEY.key-oracle* η)
($\lambda x.$ *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 η $\oplus_O$ *CIPHER.KEY.key-oracle* η
$\lambda x.$ *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* η $\surd$
⟨*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 η $\surd$
⟨*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* $\equiv$ λ-. $\mathcal{I}$*-full*
and $\mathcal{I}$*-common* $\equiv$ $\lambda\eta.$ $\mathcal{I}$*-uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*
shows
constructive-security
($\lambda\eta.$ 1_C $|_=$ (*CIPHER.enc* η $|_=$ *CIPHER.dec* η) $\odot$ *parallel-wiring* $\rhd$
parallel-resource1-wiring $\rhd$
CIPHER.KEY.res η $\parallel$
(1_C $|_=$ *MAC.enm* η $|_=$ *MAC.dem* η $\rhd$

1_C $|_=$ *parallel-wiring* $\triangleright$
parallel-resource1-wiring $\triangleright$ *MAC.RO.res* η $\|$ *INSEC.res*))
(λ-. *SEC.res*)
($\lambda\eta$. *CNV Message-Authentication-Code.sim* (*Inl None*) $\odot$ *CNV One-Time-Pad.sim None*)
($\lambda\eta$. $\mathcal{I}$-*uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert None* (*Some* ' (*nlists UNIV* η $\times$ *nlists UNIV* η))))
($\lambda\eta$. $\mathcal{I}$-*uniform UNIV* {*None*, *Some* η})
($\lambda\eta$. $\mathcal{I}$-*uniform* (*nlists UNIV* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*uniform UNIV* (*insert None* (*Some* ' *nlists UNIV* η)))
(λ-. *enat q*) *True* ($\lambda\eta$. (*id*, *map-option length*) $\circ_w$ (*insec-query-of*, *map-option snd*))
⟨*proof*⟩

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.