Constructive Cryptography in HOL

Andreas Lochbihler and S. Reza Sefidgar

April 19, 2020

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 *More-CryptHOL* **imports**
CryptHOL.CryptHOL
begin

lemma *is-empty-image* [*simp*]: *Set.is-empty* (*f* ‘ *A*) = *Set.is-empty A*
⟨*proof*⟩

lemma *inj-on-map-sum* [*simp*]:
⟦ *inj-on f A*; *inj-on g B* ⟧ ⟹ *inj-on* (*map-sum f g*) (*A* <+> *B*)
⟨*proof*⟩

lemma *inv-into-map-sum*:
inv-into (*A* <+> *B*) (*map-sum f g*) *x* = *map-sum* (*inv-into A f*) (*inv-into B g*) *x*
if *x* ∈ *f* ‘ *A* <+> *g* ‘ *B inj-on f A inj-on g B*
⟨*proof*⟩

lemma *Pair-fst-Unity*: (*fst x*, ()) = *x*
⟨*proof*⟩

fun *rsuml* :: (*'a* + *'b*) + *'c* ⇒ *'a* + (*'b* + *'c*) **where**
rsuml (*Inl* (*Inl a*)) = *Inl a*
| *rsuml* (*Inl* (*Inr b*)) = *Inr* (*Inl b*)
| *rsuml* (*Inr c*) = *Inr* (*Inr c*)

fun *lsumr* :: *'a* + (*'b* + *'c*) ⇒ (*'a* + *'b*) + *'c* **where**
lsumr (*Inl a*) = *Inl* (*Inl a*)
| *lsumr* (*Inr* (*Inl b*)) = *Inl* (*Inr b*)
| *lsumr* (*Inr* (*Inr c*)) = *Inr c*

lemma *rsuml-lsumr* [*simp*]: *rsuml* (*lsumr x*) = *x*
⟨*proof*⟩

lemma *lsumr-rsuml* [*simp*]: *lsumr* (*rsuml x*) = *x*
⟨*proof*⟩

definition *rprodl* :: (*'a* × *'b*) × *'c* ⇒ *'a* × (*'b* × *'c*) **where** *rprodl* = (λ((*a*, *b*), *c*). (*a*, (*b*, *c*)))

lemma *rprodl-simps* [*simp*]: *rprodl* ((*a*, *b*), *c*) = (*a*, (*b*, *c*))
⟨*proof*⟩

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

definition *lprodr* :: $'a \times ('b \times 'c) \Rightarrow ('a \times 'b) \times 'c$ **where** *lprodr* = $(\lambda(a,\ b,\ c).\ ((a,\ b),\ c))$

lemma *lprodr-simps* [*simp*]: *lprodr* $(a,\ b,\ c) = ((a,\ b),\ c)$
⟨*proof*⟩

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

lemma *lprodr-inverse* [*simp*]: *rprodl* (*lprodr x*) = *x*
⟨*proof*⟩

lemma *rprodl-inverse* [*simp*]: *lprodr* (*rprodl x*) = *x*
⟨*proof*⟩

lemma *rel-fun-comp*:
$\bigwedge f\ g\ h.$ *rel-fun A B* (*f* ∘ *g*) *h* = *rel-fun A* ($\lambda x.$ *B* (*f x*)) *g h*
$\bigwedge f\ g\ h.$ *rel-fun A B f* (*g* ∘ *h*) = *rel-fun A* ($\lambda x\ y.$ *B x* (*g y*)) *f h*
⟨*proof*⟩

lemma *rel-fun-map-fun1*: *rel-fun* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *A f g* ⟹ *rel-fun* (=) *A* (*map-fun h id f*) *g*
⟨*proof*⟩

lemma *rel-fun-map-fun2*: *rel-fun* (*eq-on* (*range h*)) *A f g* ⟹ *rel-fun* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *A f* (*map-fun h id g*)
⟨*proof*⟩

lemma *map-fun2-id*: *map-fun f g x* = *g* ∘ *map-fun f id x*
⟨*proof*⟩

lemma *rel-fun-refl-eq-onp*:
($\bigwedge z.\ z \in f\ `\ X \Longrightarrow A\ z\ z$) ⟹ *rel-fun* (*eq-on X*) *A f f*
⟨*proof*⟩

lemma *map-fun-id2-in*: *map-fun g h f* = *map-fun g id* (*h* ∘ *f*)
⟨*proof*⟩

lemma *Domainp-rel-fun-le*: *Domainp* (*rel-fun A B*) ≤ *pred-fun* (*Domainp A*) (*Domainp B*)
⟨*proof*⟩

lemma *eq-onE*: ⟦ *eq-on X a b*; ⟦ *b* ∈ *X*; *a* = *b* ⟧ ⟹ *thesis* ⟧ ⟹ *thesis* ⟨*proof*⟩

lemma *Domainp-eq-on* [*simp*]: *Domainp* (*eq-on X*) = ($\lambda x.$ *x* ∈ *X*)
⟨*proof*⟩

declare *eq-on-def* [*simp del*]

lemma *pred-prod-mono'* [*mono*]:
pred-prod A B xy ⟶ *pred-prod A' B' xy*
if ⋀*x*. *A x* ⟶ *A' x* ⋀*y*. *B y* ⟶ *B' y*
⟨*proof*⟩

fun *rel-witness-prod* :: ('*a* × '*b*) × ('*c* × '*d*) ⇒ (('*a* × '*c*) × ('*b* × '*d*)) **where**
rel-witness-prod ((*a*, *b*), (*c*, *d*)) = ((*a*, *c*), (*b*, *d*))

consts *relcompp-witness* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*b* ⇒ '*c* ⇒ *bool*) ⇒ '*a* × '*c* ⇒ '*b*
specification (*relcompp-witness*)
relcompp-witness1: (*A OO B*) (*fst xy*) (*snd xy*) ⟹ *A* (*fst xy*) (*relcompp-witness A B xy*)
relcompp-witness2: (*A OO B*) (*fst xy*) (*snd xy*) ⟹ *B* (*relcompp-witness A B xy*) (*snd xy*)
⟨*proof*⟩

lemmas *relcompp-witness*[*of* - - (*x*, *y*) **for** *x y*, *simplified*] = *relcompp-witness1 relcompp-witness2*

hide-fact (**open**) *relcompp-witness1 relcompp-witness2*

lemma *relcompp-witness-eq* [*simp*]: *relcompp-witness* (=) (=) (*x*, *x*) = *x*
⟨*proof*⟩

fun *rel-witness-option* :: '*a option* × '*b option* ⇒ ('*a* × '*b*) *option* **where**
rel-witness-option (*Some x*, *Some y*) = *Some* (*x*, *y*)
| *rel-witness-option* (*None*, *None*) = *None*
| *rel-witness-option* - = *None* — Just to make the definition complete

lemma *rel-witness-option*:
shows *set-rel-witness-option*: ⟦ *rel-option A x y*; (*a*, *b*) ∈ *set-option* (*rel-witness-option* (*x*, *y*)) ⟧ ⟹ *A a b*
and *map1-rel-witness-option*: *rel-option A x y* ⟹ *map-option fst* (*rel-witness-option* (*x*, *y*)) = *x*
and *map2-rel-witness-option*: *rel-option A x y* ⟹ *map-option snd* (*rel-witness-option* (*x*, *y*)) = *y*
⟨*proof*⟩

lemma *rel-witness-option1*:
assumes *rel-option A x y*
shows *rel-option* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) *x* (*rel-witness-option* (*x*, *y*))
⟨*proof*⟩

lemma *rel-witness-option2*:
assumes *rel-option A x y*
shows *rel-option* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (*rel-witness-option* (*x*, *y*)) *y*

⟨*proof*⟩

definition *rel-witness-fun* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*b* ⇒ '*c* ⇒ *bool*) ⇒ ('*a* ⇒ '*d*) × ('*c* ⇒ '*e*) ⇒ ('*b* ⇒ '*d* × '*e*) **where**
rel-witness-fun A A' = (λ(*f*, *g*) *b*. (*f* (*THE a*. *A a b*), *g* (*THE c*. *A' b c*)))

lemma
assumes *fg*: *rel-fun* (*A OO A'*) *B f g*
and *A*: *left-unique A right-total A*
and *A'*: *right-unique A' left-total A'*
shows *rel-witness-fun1*: *rel-fun A* (λ*x* (*x'*, *y*). *x* = *x'* ∧ *B x' y*) *f* (*rel-witness-fun A A'* (*f*, *g*))
and *rel-witness-fun2*: *rel-fun A'* (λ(*x*, *y'*) *y*. *y* = *y'* ∧ *B x y'*) (*rel-witness-fun A A'* (*f*, *g*)) *g*
⟨*proof*⟩

lemma *rel-witness-fun-eq* [*simp*]: *rel-witness-fun* (=) (=) (*f*, *g*) = (λ*x*. (*f x*, *g x*))
⟨*proof*⟩

consts *rel-witness-pmf* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a pmf* × '*b pmf* ⇒ ('*a* × '*b*) *pmf*
specification (*rel-witness-pmf*)
set-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) ⟹ *set-pmf* (*rel-witness-pmf A xy*) ⊆ {(*a*, *b*). *A a b*}
map1-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) ⟹ *map-pmf fst* (*rel-witness-pmf A xy*) = *fst xy*
map2-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) ⟹ *map-pmf snd* (*rel-witness-pmf A xy*) = *snd xy*
⟨*proof*⟩

lemmas *set-rel-witness-pmf* = *set-rel-witness-pmf'*[*of* - (*x*, *y*) **for** *x y*, *simplified*]
lemmas *map1-rel-witness-pmf* = *map1-rel-witness-pmf'*[*of* - (*x*, *y*) **for** *x y*, *simplified*]
lemmas *map2-rel-witness-pmf* = *map2-rel-witness-pmf'*[*of* - (*x*, *y*) **for** *x y*, *simplified*]
lemmas *rel-witness-pmf* = *set-rel-witness-pmf map1-rel-witness-pmf map2-rel-witness-pmf*

lemma *rel-witness-pmf1*:
assumes *rel-pmf A p q*
shows *rel-pmf* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) *p* (*rel-witness-pmf A* (*p*, *q*))
⟨*proof*⟩

lemma *rel-witness-pmf2*:
assumes *rel-pmf A p q*
shows *rel-pmf* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (*rel-witness-pmf A* (*p*, *q*)) *q*
⟨*proof*⟩

definition *rel-witness-spmf* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a spmf* × '*b spmf* ⇒ ('*a* ×

$'b$) *spmf* **where**
rel-witness-spmf A = *map-pmf rel-witness-option* ∘ *rel-witness-pmf* (*rel-option A*)

lemma assumes *rel-spmf A p q*
shows *rel-witness-spmf1*: *rel-spmf* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) *p* (*rel-witness-spmf A* (*p*, *q*))
and *rel-witness-spmf2*: *rel-spmf* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (*rel-witness-spmf A* (*p*, *q*)) *q*
⟨*proof*⟩

primrec (*transfer*) *enforce-option* :: (*'a* ⇒ *bool*) ⇒ *'a option* ⇒ *'a option* **where**
enforce-option P (*Some x*) = (*if P x then Some x else None*)
| *enforce-option P None* = *None*

lemma *set-enforce-option* [*simp*]: *set-option* (*enforce-option P x*) = {*a* ∈ *set-option x*. *P a*}
⟨*proof*⟩

lemma *enforce-map-option*: *enforce-option P* (*map-option f x*) = *map-option f* (*enforce-option* (*P* ∘ *f*) *x*)
⟨*proof*⟩

lemma *enforce-bind-option* [*simp*]:
enforce-option P (*Option.bind x f*) = *Option.bind x* (*enforce-option P* ∘ *f*)
⟨*proof*⟩

lemma *enforce-option-alt-def*:
enforce-option P x = *Option.bind x* (λ*a*. *Option.bind* (*assert-option* (*P a*)) (λ-:: *unit*. *Some a*))
⟨*proof*⟩

lemma *enforce-option-eq-None-iff* [*simp*]:
enforce-option P x = *None* ⟷ (∀ *a*. *x* = *Some a* ⟶ ¬ *P a*)
⟨*proof*⟩

lemma *enforce-option-eq-Some-iff* [*simp*]:
enforce-option P x = *Some y* ⟷ *x* = *Some y* ∧ *P y*
⟨*proof*⟩

lemma *Some-eq-enforce-option-iff* [*simp*]:
Some y = *enforce-option P x* ⟷ *x* = *Some y* ∧ *P y*
⟨*proof*⟩

lemma *enforce-option-top* [*simp*]: *enforce-option* ⊤ = *id*
⟨*proof*⟩

lemma *enforce-option-K-True* [*simp*]: *enforce-option* (λ-. *True*) *x* = *x*

⟨*proof*⟩

lemma *enforce-option-bot* [*simp*]: *enforce-option* ⊥ = (λ-. *None*)
⟨*proof*⟩

lemma *enforce-option-K-False* [*simp*]: *enforce-option* (λ-. *False*) *x* = *None*
⟨*proof*⟩

lemma *enforce-pred-id-option*: *pred-option P x* ⟹ *enforce-option P x* = *x*
⟨*proof*⟩

lemma *rel-fun-refl*: ⟦ *A* ≤ (=); (=) ≤ *B* ⟧ ⟹ (=) ≤ *rel-fun A B*
⟨*proof*⟩

lemma *rel-fun-mono-strong*:
⟦ *rel-fun A B f g*; *A'* ≤ *A*; ⋀*x y*. ⟦ *x* ∈ *f* ‘ {*x*. *Domainp A' x*}; *y* ∈ *g* ‘ {*x*. *Rangep A' x*}; *B x y* ⟧ ⟹ *B' x y* ⟧ ⟹ *rel-fun A' B' f g*
⟨*proof*⟩

lemma *rel-fun-refl-strong*:
assumes *A* ≤ (=) ⋀*x*. *x* ∈ *f* ‘ {*x*. *Domainp A x*} ⟹ *B x x*
shows *rel-fun A B f f*
⟨*proof*⟩

lemma *Grp-iff*: *BNF-Def.Grp B g x y* ⟷ *y* = *g x* ∧ *x* ∈ *B* ⟨*proof*⟩

lemma *Rangep-Grp*: *Rangep* (*BNF-Def.Grp A f*) = (λ*x*. *x* ∈ *f* ‘ *A*)
⟨*proof*⟩

lemma *Domainp-Grp*: *Domainp* (*BNF-Def.Grp A f*) = (λ*x*. *x* ∈ *A*)
⟨*proof*⟩

lemma *rel-fun-Grp*:
rel-fun (*BNF-Def.Grp UNIV h*)$^{-1-1}$ (*BNF-Def.Grp A g*) = *BNF-Def.Grp* {*f*. *f* ‘ *range h* ⊆ *A*} (*map-fun h g*)
⟨*proof*⟩

lemma *wf-strict-prefix*: *wfP strict-prefix*
⟨*proof*⟩

lemma *strict-prefix-setD*:
strict-prefix xs ys ⟹ *set xs* ⊆ *set ys*
⟨*proof*⟩

lemma *weight-assert-spmf* [*simp*]: *weight-spmf* (*assert-spmf b*) = *indicator* {*True*} *b*

⟨*proof*⟩

definition *enforce-spmf* :: (*'a* ⇒ *bool*) ⇒ *'a spmf* ⇒ *'a spmf* **where**
enforce-spmf P = *map-pmf* (*enforce-option P*)

lemma *enforce-spmf-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*A* ===> (=)) ===> *rel-spmf A* ===> *rel-spmf A*) *enforce-spmf enforce-spmf*
⟨*proof*⟩

lemma *enforce-return-spmf* [*simp*]:
enforce-spmf P (*return-spmf x*) = (*if P x then return-spmf x else return-pmf None*)
⟨*proof*⟩

lemma *enforce-return-pmf-None* [*simp*]:
enforce-spmf P (*return-pmf None*) = *return-pmf None*
⟨*proof*⟩

lemma *enforce-map-spmf*:
enforce-spmf P (*map-spmf f p*) = *map-spmf f* (*enforce-spmf* (*P* ∘ *f*) *p*)
⟨*proof*⟩

lemma *enforce-bind-spmf* [*simp*]:
enforce-spmf P (*bind-spmf p f*) = *bind-spmf p* (*enforce-spmf P* ∘ *f*)
⟨*proof*⟩

lemma *set-enforce-spmf* [*simp*]: *set-spmf* (*enforce-spmf P p*) = {*a* ∈ *set-spmf p*. *P a*}
⟨*proof*⟩

lemma *enforce-spmf-alt-def*:
enforce-spmf P p = *bind-spmf p* (λ*a*. *bind-spmf* (*assert-spmf* (*P a*)) (λ- :: *unit*. *return-spmf a*))
⟨*proof*⟩

lemma *bind-enforce-spmf* [*simp*]:
bind-spmf (*enforce-spmf P p*) *f* = *bind-spmf p* (λ*x*. *if P x then f x else return-pmf None*)
⟨*proof*⟩

lemma *weight-enforce-spmf*:
weight-spmf (*enforce-spmf P p*) = *weight-spmf p* − *measure* (*measure-spmf p*) {*x*. ¬ *P x*} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *lossless-enforce-spmf* [*simp*]:
lossless-spmf (*enforce-spmf P p*) ⟷ *lossless-spmf p* ∧ *set-spmf p* ⊆ {*x*. *P x*}
⟨*proof*⟩

lemma *enforce-spmf-top* [*simp*]: *enforce-spmf* $\top$ = *id*
⟨*proof*⟩

lemma *enforce-spmf-K-True* [*simp*]: *enforce-spmf* (λ-. *True*) *p* = *p*
⟨*proof*⟩

lemma *enforce-spmf-bot* [*simp*]: *enforce-spmf* $\bot$ = (λ-. *return-pmf None*)
⟨*proof*⟩

lemma *enforce-spmf-K-False* [*simp*]: *enforce-spmf* (λ-. *False*) *p* = *return-pmf None*
⟨*proof*⟩

lemma *enforce-pred-id-spmf*: *enforce-spmf P p* = *p* **if** *pred-spmf P p*
⟨*proof*⟩

lemma *map-the-spmf-of-pmf* [*simp*]: *map-pmf the* (*spmf-of-pmf p*) = *p*
⟨*proof*⟩

lemma *bind-bind-conv-pair-spmf*:
bind-spmf p (λ*x*. *bind-spmf q* (*f x*)) = *bind-spmf* (*pair-spmf p q*) (λ(*x*, *y*). *f x y*)
⟨*proof*⟩

lemma *cond-pmf-of-set*:
assumes *fin*: *finite A* **and** *nonempty*: *A* ∩ *B* ≠ {}
shows *cond-pmf* (*pmf-of-set A*) *B* = *pmf-of-set* (*A* ∩ *B*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *cond-spmf-spmf-of-set*:
cond-spmf (*spmf-of-set A*) *B* = *spmf-of-set* (*A* ∩ *B*) **if** *finite A*
⟨*proof*⟩

lemma *pair-pmf-of-set*:
assumes *A*: *finite A A* ≠ {}
and *B*: *finite B B* ≠ {}
shows *pair-pmf* (*pmf-of-set A*) (*pmf-of-set B*) = *pmf-of-set* (*A* × *B*)
⟨*proof*⟩

lemma *pair-spmf-of-set*:
pair-spmf (*spmf-of-set A*) (*spmf-of-set B*) = *spmf-of-set* (*A* × *B*)
⟨*proof*⟩

lemma *emeasure-cond-pmf*:
fixes *p A*
defines *q* ≡ *cond-pmf p A*
assumes *set-pmf p* ∩ *A* ≠ {}
shows *emeasure* (*measure-pmf q*) *B* = *emeasure* (*measure-pmf p*) (*A* ∩ *B*) / *emeasure* (*measure-pmf p*) *A*

⟨*proof*⟩

lemma *measure-cond-pmf*:
measure (*measure-pmf* (*cond-pmf p A*)) *B* = *measure* (*measure-pmf p*) (*A* ∩ *B*) / *measure* (*measure-pmf p*) *A*
if *set-pmf p* ∩ *A* ≠ {}
⟨*proof*⟩

lemma *emeasure-measure-pmf-zero-iff*: *emeasure* (*measure-pmf p*) *s* = *0* ⟷ *set-pmf p* ∩ *s* = {} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *emeasure-cond-spmf*:
emeasure (*measure-spmf* (*cond-spmf p A*)) *B* = *emeasure* (*measure-spmf p*) (*A* ∩ *B*) / *emeasure* (*measure-spmf p*) *A*
⟨*proof*⟩

lemma *measure-cond-spmf*:
measure (*measure-spmf* (*cond-spmf p A*)) *B* = *measure* (*measure-spmf p*) (*A* ∩ *B*) / *measure* (*measure-spmf p*) *A*
⟨*proof*⟩

lemma *lossless-cond-spmf* [*simp*]: *lossless-spmf* (*cond-spmf p A*) ⟷ *set-spmf p* ∩ *A* ≠ {}
⟨*proof*⟩

lemma *measure-spmf-eq-density*: *measure-spmf p* = *density* (*count-space UNIV*) (*spmf p*)
⟨*proof*⟩

lemma *integral-measure-spmf*:
fixes *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
assumes *A*: *finite A*
shows (⋀*a*. *a* ∈ *set-spmf M* ⟹ *f a* ≠ *0* ⟹ *a* ∈ *A*) ⟹ (*LINT x*|*measure-spmf M*. *f x*) = (∑ *a*∈*A*. *spmf M a* ∗*R f a*)
⟨*proof*⟩

lemma *image-set-spmf-eq*:
f ‘ *set-spmf p* = *g* ‘ *set-spmf q* **if** *ASSUMPTION* (*map-spmf f p* = *map-spmf g q*)
⟨*proof*⟩

lemma *map-spmf-const*: *map-spmf* (λ-. *x*) *p* = *scale-spmf* (*weight-spmf p*) (*return-spmf x*)
⟨*proof*⟩

lemma *cond-return-pmf* [*simp*]: *cond-pmf* (*return-pmf x*) *A* = *return-pmf x* **if** *x*

$\in A$
⟨*proof*⟩

lemma *cond-return-spmf* [*simp*]: *cond-spmf* (*return-spmf x*) *A* = (*if x* ∈ *A then return-spmf x else return-pmf None*)
⟨*proof*⟩

lemma *measure-range-Some-eq-weight*:
measure (*measure-pmf p*) (*range Some*) = *weight-spmf p*
⟨*proof*⟩

lemma *restrict-spmf-eq-return-pmf-None* [*simp*]:
restrict-spmf p A = *return-pmf None* ⟷ *set-spmf p* ∩ *A* = {}
⟨*proof*⟩

lemma *integrable-scale-measure* [*simp*]:
⟦ *integrable M f*; *r* < ⊤ ⟧ ⟹ *integrable* (*scale-measure r M*) *f*
for *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
⟨*proof*⟩

lemma *integral-scale-measure*:
assumes *integrable M f r* < ⊤
shows *integral*^{L} (*scale-measure r M*) *f* = *enn2real r* ∗ *integral*^{L} *M f*
⟨*proof*⟩

definition *mk-lossless* :: *'a spmf* ⇒ *'a spmf* **where**
mk-lossless p = *scale-spmf* (*inverse* (*weight-spmf p*)) *p*

lemma *mk-lossless-idem* [*simp*]: *mk-lossless* (*mk-lossless p*) = *mk-lossless p*
⟨*proof*⟩

lemma *mk-lossless-return* [*simp*]: *mk-lossless* (*return-pmf x*) = *return-pmf x*
⟨*proof*⟩

lemma *mk-lossless-map* [*simp*]: *mk-lossless* (*map-spmf f p*) = *map-spmf f* (*mk-lossless p*)
⟨*proof*⟩

lemma *spmf-mk-lossless* [*simp*]: *spmf* (*mk-lossless p*) *x* = *spmf p x* / *weight-spmf p*
⟨*proof*⟩

lemma *set-spmf-mk-lossless* [*simp*]: *set-spmf* (*mk-lossless p*) = *set-spmf p*
⟨*proof*⟩

lemma *mk-lossless-lossless* [*simp*]: *lossless-spmf p* ⟹ *mk-lossless p* = *p*
⟨*proof*⟩

lemma *mk-lossless-eq-return-pmf-None* [*simp*]: *mk-lossless p* = *return-pmf None*

$\longleftrightarrow p = \textit{return-pmf None}$
⟨*proof*⟩

lemma *return-pmf-None-eq-mk-lossless* [*simp*]: *return-pmf None* = *mk-lossless p* $\longleftrightarrow p = \textit{return-pmf None}$
⟨*proof*⟩

lemma *mk-lossless-spmf-of-set* [*simp*]: *mk-lossless* (*spmf-of-set A*) = *spmf-of-set A*
⟨*proof*⟩

lemma *weight-mk-lossless*: *weight-spmf* (*mk-lossless p*) = (*if p* = *return-pmf None then 0 else 1*)
⟨*proof*⟩

lemma *mk-lossless-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-spmf A* ===> *rel-spmf A*) *mk-lossless mk-lossless*
⟨*proof*⟩

lemma *rel-spmf-mk-losslessI*:
rel-spmf A p q $\Longrightarrow$ *rel-spmf A* (*mk-lossless p*) (*mk-lossless q*)
⟨*proof*⟩

lemma *rel-spmf-restrict-spmfI*:
rel-spmf ($\lambda x\ y.\ (x \in A \wedge y \in B \wedge R\ x\ y) \vee x \notin A \wedge y \notin B$) *p q*
$\Longrightarrow$ *rel-spmf R* (*restrict-spmf p A*) (*restrict-spmf q B*)
⟨*proof*⟩

lemma *cond-spmf-alt*: *cond-spmf p A* = *mk-lossless* (*restrict-spmf p A*)
⟨*proof*⟩

lemma *cond-spmf-bind*:
cond-spmf (*bind-spmf p f*) *A* = *mk-lossless* ($p \gg\!= (\lambda x.\ f\ x \restriction A)$)
⟨*proof*⟩

lemma *cond-spmf-UNIV* [*simp*]: *cond-spmf p UNIV* = *mk-lossless p*
⟨*proof*⟩

lemma *cond-pmf-singleton*:
cond-pmf p A = *return-pmf x* **if** *set-pmf p* $\cap$ *A* = $\{x\}$
⟨*proof*⟩

definition *cond-spmf-fst* :: ('*a* × '*b*) *spmf* ⇒ '*a* ⇒ '*b spmf* **where**
cond-spmf-fst p a = *map-spmf snd* (*cond-spmf p* ($\{a\}$ × *UNIV*))

lemma *cond-spmf-fst-return-spmf* [*simp*]:
cond-spmf-fst (*return-spmf* (*x*, *y*)) *x* = *return-spmf y*
⟨*proof*⟩

lemma *cond-spmf-fst-map-Pair* [*simp*]: *cond-spmf-fst* (*map-spmf* (*Pair x*) *p*) *x* = *mk-lossless p*
⟨*proof*⟩

lemma *cond-spmf-fst-map-Pair'* [*simp*]: *cond-spmf-fst* (*map-spmf* (λ*y*. (*x*, *f y*)) *p*) *x* = *map-spmf f* (*mk-lossless p*)
⟨*proof*⟩

lemma *cond-spmf-fst-eq-return-None* [*simp*]: *cond-spmf-fst p x* = *return-pmf None* ⟷ *x* ∉ *fst* ‘ *set-spmf p*
⟨*proof*⟩

lemma *cond-spmf-fst-map-Pair1*:
cond-spmf-fst (*map-spmf* (λ*x*. (*f x*, *g x*)) *p*) (*f x*) = *return-spmf* (*g* (*inv-into* (*set-spmf p*) *f* (*f x*)))
if *x* ∈ *set-spmf p inj-on f* (*set-spmf p*)
⟨*proof*⟩

lemma *lossless-cond-spmf-fst* [*simp*]: *lossless-spmf* (*cond-spmf-fst p x*) ⟷ *x* ∈ *fst* ‘ *set-spmf p*
⟨*proof*⟩

lemma *cond-spmf-fst-inverse*:
bind-spmf (*map-spmf fst p*) (λ*x*. *map-spmf* (*Pair x*) (*cond-spmf-fst p x*)) = *p*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

fun *rel-witness-generat* :: ('*a*, '*c*, '*e*) *generat* × ('*b*, '*d*, '*f*) *generat* ⇒ ('*a* × '*b*, '*c* × '*d*, '*e* × '*f*) *generat* **where**
rel-witness-generat (*Pure x*, *Pure y*) = *Pure* (*x*, *y*)
| *rel-witness-generat* (*IO out c*, *IO out' c'*) = *IO* (*out*, *out'*) (*c*, *c'*)

lemma *rel-witness-generat*:
assumes *rel-generat A C R x y*
shows *pures-rel-witness-generat*: *generat-pures* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*a*, *b*). *A a b*}
and *outs-rel-witness-generat*: *generat-outs* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*c*, *d*). *C c d*}
and *conts-rel-witness-generat*: *generat-conts* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*e*, *f*). *R e f*}
and *map1-rel-witness-generat*: *map-generat fst fst fst* (*rel-witness-generat* (*x*, *y*)) = *x*
and *map2-rel-witness-generat*: *map-generat snd snd snd* (*rel-witness-generat* (*x*, *y*)) = *y*
⟨*proof*⟩

lemmas *set-rel-witness-generat* = *pures-rel-witness-generat outs-rel-witness-generat conts-rel-witness-generat*

lemma *rel-witness-generat1*:
assumes *rel-generat A C R x y*
shows *rel-generat* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) (λ*c* (*c'*, *d*). *c* = *c'* ∧ *C c' d*) (λ*r* (*r'*, *s*). *r* = *r'* ∧ *R r' s*) *x* (*rel-witness-generat* (*x*, *y*))
⟨*proof*⟩

lemma *rel-witness-generat2*:
assumes *rel-generat A C R x y*
shows *rel-generat* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (λ(*c*, *d'*) *d*. *d* = *d'* ∧ *C c d'*) (λ(*r*, *s'*) *s*. *s* = *s'* ∧ *R r s'*) (*rel-witness-generat* (*x*, *y*)) *y*
⟨*proof*⟩

lemma *rel-gpv''-map-gpv1*:
rel-gpv'' A C R (*map-gpv f g gpv*) *gpv'* = *rel-gpv''* (λ*a*. *A* (*f a*)) (λ*c*. *C* (*g c*)) *R gpv gpv'* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *rel-gpv''-map-gpv2*:
rel-gpv'' A C R gpv (*map-gpv f g gpv'*) = *rel-gpv''* (λ*a b*. *A a* (*f b*)) (λ*c d*. *C c* (*g d*)) *R gpv gpv'*
⟨*proof*⟩

lemmas *rel-gpv''-map-gpv* = *rel-gpv''-map-gpv1*[*abs-def*] *rel-gpv''-map-gpv2*

lemma *rel-gpv''-map-gpv'* [*simp*]:
shows ⋀*f g h gpv*. *NO-MATCH id f* ∨ *NO-MATCH id g*
⟹ *rel-gpv'' A C R* (*map-gpv' f g h gpv*) = *rel-gpv''* (λ*a*. *A* (*f a*)) (λ*c*. *C* (*g c*)) *R* (*map-gpv' id id h gpv*)
and ⋀*f g h gpv gpv'*. *NO-MATCH id f* ∨ *NO-MATCH id g*
⟹ *rel-gpv'' A C R gpv* (*map-gpv' f g h gpv'*) = *rel-gpv''* (λ*a b*. *A a* (*f b*)) (λ*c d*. *C c* (*g d*)) *R gpv* (*map-gpv' id id h gpv'*)
⟨*proof*⟩

lemmas *rel-gpv-map-gpv'* = *rel-gpv''-map-gpv'*[**where** *R*=(=), *folded rel-gpv-conv-rel-gpv''*]

definition *rel-witness-gpv* :: ('*a* ⇒ '*d* ⇒ *bool*) ⇒ ('*b* ⇒ '*e* ⇒ *bool*) ⇒ ('*c* ⇒ '*g* ⇒ *bool*) ⇒ ('*g* ⇒ '*f* ⇒ *bool*) ⇒ ('*a*, '*b*, '*c*) *gpv* × ('*d*, '*e*, '*f*) *gpv* ⇒ ('*a* × '*d*, '*b* × '*e*, '*g*) *gpv* **where**
rel-witness-gpv A C R R' = *corec-gpv* (
map-spmf (*map-generat id id* (λ(*rpv*, *rpv'*). (*Inr* ∘ *rel-witness-fun R R'* (*rpv*, *rpv'*))) ∘ *rel-witness-generat*) ∘
rel-witness-spmf (*rel-generat A C* (*rel-fun* (*R OO R'*) (*rel-gpv'' A C* (*R OO R'*)))) ∘ *map-prod the-gpv the-gpv*)

lemma *rel-witness-gpv-sel* [*simp*]:
the-gpv (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) =
map-spmf (*map-generat id id* (λ(*rpv*, *rpv'*). (*rel-witness-gpv A C R R'* ∘ *rel-witness-fun R R'* (*rpv*, *rpv'*))) ∘ *rel-witness-generat*)
(*rel-witness-spmf* (*rel-generat A C* (*rel-fun* (*R OO R'*) (*rel-gpv'' A C* (*R OO R'*)))) (*the-gpv gpv*, *the-gpv gpv'*))
⟨*proof*⟩

lemma assumes *rel-gpv'' A C* (*R OO R'*) *gpv gpv'*
and *R*: *left-unique R right-total R*
and *R'*: *right-unique R' left-total R'*
shows *rel-witness-gpv1*: *rel-gpv''* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) (λ*c* (*c'*, *d*). *c* = *c'* ∧ *C c' d*) *R gpv* (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) (**is** *?thesis1*)
and *rel-witness-gpv2*: *rel-gpv''* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (λ(*c*, *d'*) *d*. *d* = *d'* ∧ *C c d'*) *R'* (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) *gpv'* (**is** *?thesis2*)
⟨*proof*⟩

lemma *rel-gpv''-neg-distr*:
assumes *R*: *left-unique R right-total R*
and *R'*: *right-unique R' left-total R'*
shows *rel-gpv''* (*A OO A'*) (*C OO C'*) (*R OO R'*) ≤ *rel-gpv'' A C R OO rel-gpv'' A' C' R'*
⟨*proof*⟩

lemma *rel-gpv''-mono'* [*mono*]:
assumes ⋀*x y*. *A x y* ⟶ *A' x y*
and ⋀*x y*. *C x y* ⟶ *C' x y*
and ⋀*x y*. *R' x y* ⟶ *R x y*
shows *rel-gpv'' A C R gpv gpv'* ⟶ *rel-gpv'' A' C' R' gpv gpv'*
⟨*proof*⟩

context includes *I.lifting* **begin**

lift-definition *I-uniform* :: *'out set* ⇒ *'in set* ⇒ (*'out*, *'in*) *I* **is** λ*A B x*. *if x* ∈ *A then B else* {} ⟨*proof*⟩

lemma *outs-I-uniform* [*simp*]: *outs-I* (*I-uniform A B*) = (*if B* = {} *then* {} *else A*)
⟨*proof*⟩

lemma *responses-I-uniform* [*simp*]: *responses-I* (*I-uniform A B*) *x* = (*if x* ∈ *A then B else* {})
⟨*proof*⟩

lemma *I-uniform-UNIV* [*simp*]: *I-uniform UNIV UNIV* = *I-full*
⟨*proof*⟩

lifting-update *I.lifting*

lifting-forget $\mathcal{I}$.*lifting*

end

lemma $\mathcal{I}$-*eqI*: $[\![$ *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$; $\bigwedge x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}' \Longrightarrow$ *responses-*$\mathcal{I}$ $\mathcal{I}$ x = *responses-*$\mathcal{I}$ $\mathcal{I}'$ x $]\!] \Longrightarrow \mathcal{I} = \mathcal{I}'$
including $\mathcal{I}$.*lifting* ⟨*proof*⟩

instantiation $\mathcal{I}$ :: (*type*, *type*) *order* **begin**

definition *less-eq-*$\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*a*, '*b*) $\mathcal{I}$ ⇒ *bool*
where *le-*$\mathcal{I}$*-def*: *less-eq-*$\mathcal{I}$ $\mathcal{I}$ $\mathcal{I}' \longleftrightarrow$ *outs-*$\mathcal{I}$ $\mathcal{I} \subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}' \wedge (\forall x{\in}$*outs-*$\mathcal{I}$ $\mathcal{I}$. *responses-*$\mathcal{I}$ $\mathcal{I}'$ $x \subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ x)

definition *less-*$\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*a*, '*b*) $\mathcal{I}$ ⇒ *bool*
where *less-*$\mathcal{I}$ = *mk-less* (≤)

instance
⟨*proof*⟩
end

instantiation $\mathcal{I}$ :: (*type*, *type*) *order-bot* **begin**
definition *bot-*$\mathcal{I}$ :: ('*a*, '*b*) $\mathcal{I}$ **where** *bot-*$\mathcal{I}$ = $\mathcal{I}$-*uniform* {} *UNIV*
instance ⟨*proof*⟩
end

lemma *outs-*$\mathcal{I}$*-bot* [*simp*]: *outs-*$\mathcal{I}$ *bot* = {}
⟨*proof*⟩

lemma *respones-*$\mathcal{I}$*-bot* [*simp*]: *responses-*$\mathcal{I}$ *bot* x = {}
⟨*proof*⟩

lemma *outs-*$\mathcal{I}$*-mono*: $\mathcal{I} \leq \mathcal{I}' \Longrightarrow$ *outs-*$\mathcal{I}$ $\mathcal{I} \subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}'$
⟨*proof*⟩

lemma *responses-*$\mathcal{I}$*-mono*: $[\![$ $\mathcal{I} \leq \mathcal{I}'$; $x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!] \Longrightarrow$ *responses-*$\mathcal{I}$ $\mathcal{I}'$ $x \subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ x
⟨*proof*⟩

lemma $\mathcal{I}$-*uniform-empty* [*simp*]: $\mathcal{I}$-*uniform* {} A = *bot*
⟨*proof*⟩ **including** $\mathcal{I}$.*lifting* ⟨*proof*⟩

lemma *WT-gpv-*$\mathcal{I}$*-mono*: $[\![$ $\mathcal{I} \vdash g$ *gpv* $\surd$; $\mathcal{I} \leq \mathcal{I}'$ $]\!] \Longrightarrow \mathcal{I}' \vdash g$ *gpv* $\surd$
⟨*proof*⟩

lemma *results-gpv-mono*:
assumes *le*: $\mathcal{I}' \leq \mathcal{I}$ **and** *WT*: $\mathcal{I}' \vdash g$ *gpv* $\surd$
shows *results-gpv* $\mathcal{I}$ *gpv* ⊆ *results-gpv* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

lemma *$\mathcal{I}$-uniform-mono*:
$\mathcal{I}$-uniform A B $\leq$ *$\mathcal{I}$-uniform C D* **if** $A \subseteq C$ $D \subseteq B$ $D = \{\} \longrightarrow B = \{\}$
⟨*proof*⟩

context begin
qualified inductive *outsp-gpv* :: (*'out*, *'in*) $\mathcal{I} \Rightarrow$ *'out* $\Rightarrow$ (*'a*, *'out*, *'in*) *gpv* $\Rightarrow$ *bool*
for $\mathcal{I}$ *x* **where**
IO: *IO x c* $\in$ *set-spmf* (*the-gpv gpv*) $\Longrightarrow$ *outsp-gpv* $\mathcal{I}$ *x gpv*
| *Cont*: ⟦ *IO out rpv* $\in$ *set-spmf* (*the-gpv gpv*); *input* $\in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *out*; *outsp-gpv* $\mathcal{I}$ *x* (*rpv input*) ⟧
$\Longrightarrow$ *outsp-gpv* $\mathcal{I}$ *x gpv*

definition *outs-gpv* :: (*'out*, *'in*) $\mathcal{I} \Rightarrow$ (*'a*, *'out*, *'in*) *gpv* $\Rightarrow$ *'out set*
where *outs-gpv* $\mathcal{I}$ *gpv* $\equiv \{x.$ *outsp-gpv* $\mathcal{I}$ *x gpv*$\}$

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

context begin
⟨*ML*⟩

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

inductive-simps *outs-gpv-GPV* [*to-set*, *simp*]: *outsp-gpv* $\mathcal{I}$ *x* (*GPV gpv*)

end

lemma *outs-gpv-Done* [*iff*]: *outs-gpv* $\mathcal{I}$ (*Done x*) $= \{\}$
⟨*proof*⟩

lemma *outs-gpv-Fail* [*iff*]: *outs-gpv* $\mathcal{I}$ *Fail* $= \{\}$
⟨*proof*⟩

lemma *outs-gpv-Pause* [*simp*]:
outs-gpv $\mathcal{I}$ (*Pause out c*) = *insert out* ($\bigcup$ *input*$\in$*responses-$\mathcal{I}$* $\mathcal{I}$ *out*. *outs-gpv* $\mathcal{I}$ (*c input*))
⟨*proof*⟩

lemma *outs-gpv-lift-spmf* [*iff*]: *outs-gpv* $\mathcal{I}$ (*lift-spmf p*) $= \{\}$
⟨*proof*⟩

lemma *outs-gpv-assert-gpv* [*simp*]: *outs-gpv* $\mathcal{I}$ (*assert-gpv b*) = {}
⟨*proof*⟩

lemma *outs-gpv-bind-gpv* [*simp*]:
outs-gpv $\mathcal{I}$ (*gpv* ⤜ *f*) = *outs-gpv* $\mathcal{I}$ *gpv* ∪ (⋃ *x*∈*results-gpv* $\mathcal{I}$ *gpv*. *outs-gpv* $\mathcal{I}$ (*f x*))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *outs-gpv-*$\mathcal{I}$*-full*: *outs-gpv* $\mathcal{I}$*-full* = *outs'-gpv*
⟨*proof*⟩

lemma *outs'-bind-gpv* [*simp*]:
outs'-gpv (*bind-gpv gpv f*) = *outs'-gpv gpv* ∪ (⋃ *x*∈*results'-gpv gpv*. *outs'-gpv* (*f x*))
⟨*proof*⟩

lemma *outs-gpv-map-gpv-id* [*simp*]: *outs-gpv* $\mathcal{I}$ (*map-gpv f id gpv*) = *outs-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *outs-gpv-map-gpv-id'* [*simp*]: *outs-gpv* $\mathcal{I}$ (*map-gpv f* (λ*x*. *x*) *gpv*) = *outs-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *outs'-gpv-bind-option* [*simp*]:
outs'-gpv (*monad.bind-option Fail x f*) = (⋃ *y*∈*set-option x*. *outs'-gpv* (*f y*))
⟨*proof*⟩

lemma *WT-gpv-outs-gpv*:
assumes $\mathcal{I}$ ⊢*g gpv* √
shows *outs-gpv* $\mathcal{I}$ *gpv* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$
⟨*proof*⟩

context includes $\mathcal{I}$*.lifting* **begin**

lift-definition *map-*$\mathcal{I}$ :: ('*out'* ⇒ '*out*) ⇒ ('*in* ⇒ '*in'*) ⇒ ('*out*, '*in*) $\mathcal{I}$ ⇒ ('*out'*, '*in'*) $\mathcal{I}$
is λ*f g resp x*. *g* ‘ *resp* (*f x*) ⟨*proof*⟩

lemma *outs-*$\mathcal{I}$*-map-*$\mathcal{I}$ [*simp*]:
outs-$\mathcal{I}$ (*map-*$\mathcal{I}$ *f g* $\mathcal{I}$) = *f* −‘ *outs-*$\mathcal{I}$ $\mathcal{I}$
⟨*proof*⟩

lemma *responses-*$\mathcal{I}$*-map-*$\mathcal{I}$ [*simp*]:
responses-$\mathcal{I}$ (*map-*$\mathcal{I}$ *f g* $\mathcal{I}$) *x* = *g* ‘ *responses-*$\mathcal{I}$ $\mathcal{I}$ (*f x*)
⟨*proof*⟩

lemma *map-*$\mathcal{I}$*-*$\mathcal{I}$*-uniform* [*simp*]:

map-$\mathcal{I}$ f g (*$\mathcal{I}$-uniform A B*) = *$\mathcal{I}$-uniform* (*f −‘ A*) (*g ‘ B*)
⟨*proof*⟩

lemma *map-$\mathcal{I}$-id* [*simp*]: *map-$\mathcal{I}$ id id $\mathcal{I}$* = *$\mathcal{I}$*
⟨*proof*⟩

lemma *map-$\mathcal{I}$-id0*: *map-$\mathcal{I}$ id id* = *id*
⟨*proof*⟩

lemma *map-$\mathcal{I}$-comp* [*simp*]: *map-$\mathcal{I}$ f g* (*map-$\mathcal{I}$ f′ g′ $\mathcal{I}$*) = *map-$\mathcal{I}$* (*f′* ∘ *f*) (*g* ∘ *g′*) *$\mathcal{I}$*
⟨*proof*⟩

lemma *map-$\mathcal{I}$-cong*: *map-$\mathcal{I}$ f g $\mathcal{I}$* = *map-$\mathcal{I}$ f′ g′ $\mathcal{I}$′*
if *$\mathcal{I}$* = *$\mathcal{I}$′* **and** *f*: *f* = *f′* **and** ⋀*x y*. ⟦ *x* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$′*; *y* ∈ *responses-$\mathcal{I}$ $\mathcal{I}$′ x* ⟧ ⟹ *g y* = *g′ y*
⟨*proof*⟩

lifting-update *$\mathcal{I}$.lifting*
lifting-forget *$\mathcal{I}$.lifting*
end

functor *map-$\mathcal{I}$* ⟨*proof*⟩

lemma *WT-gpv-map-gpv′*: *$\mathcal{I}$* ⊢*g map-gpv′ f g h gpv* √ **if** *map-$\mathcal{I}$ g h $\mathcal{I}$* ⊢*g gpv* √
⟨*proof*⟩

lemma *WT-gpv-map-gpv*: *$\mathcal{I}$* ⊢*g map-gpv f g gpv* √ **if** *map-$\mathcal{I}$ g id $\mathcal{I}$* ⊢*g gpv* √
⟨*proof*⟩

lemma *results-gpv-map-gpv′* [*simp*]:
results-gpv $\mathcal{I}$ (*map-gpv′ f g h gpv*) = *f ‘* (*results-gpv* (*map-$\mathcal{I}$ g h $\mathcal{I}$*) *gpv*)
⟨*proof*⟩

lemma *map-$\mathcal{I}$-plus-$\mathcal{I}$* [*simp*]:
map-$\mathcal{I}$ (*map-sum f1 f2*) (*map-sum g1 g2*) (*$\mathcal{I}$1* ⊕$_{\mathcal{I}}$ *$\mathcal{I}$2*) = *map-$\mathcal{I}$ f1 g1 $\mathcal{I}$1* ⊕$_{\mathcal{I}}$ *map-$\mathcal{I}$ f2 g2 $\mathcal{I}$2*
⟨*proof*⟩

lemma *le-plus-$\mathcal{I}$-iff* [*simp*]:
$\mathcal{I}$1 ⊕$_{\mathcal{I}}$ *$\mathcal{I}$2* ≤ *$\mathcal{I}$1′* ⊕$_{\mathcal{I}}$ *$\mathcal{I}$2′* ⟷ *$\mathcal{I}$1* ≤ *$\mathcal{I}$1′* ∧ *$\mathcal{I}$2* ≤ *$\mathcal{I}$2′*
⟨*proof*⟩

inductive *pred-gpv′* :: (*'a* ⇒ *bool*) ⇒ (*'out* ⇒ *bool*) ⇒ *'in set* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *bool* **for** *P Q X gpv* **where**
pred-gpv′ P Q X gpv
if ⋀*x*. *x* ∈ *results-gpv* (*$\mathcal{I}$-uniform UNIV X*) *gpv* ⟹ *P x* ⋀*out*. *out* ∈ *outs-gpv*

$(\mathcal{I}\text{-}uniform\ UNIV\ X)\ gpv \Longrightarrow Q\ out$

lemma *pred-gpv-conv-pred-gpv'*: *pred-gpv P Q = pred-gpv' P Q UNIV*
⟨*proof*⟩

lemma *rel-gpv''-Grp*: **includes** *lifting-syntax* **shows**
rel-gpv'' (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$
=
BNF-Def.Grp {*x*. *results-gpv* (*I-uniform UNIV* (*range h*)) *x* ⊆ *A* ∧ *outs-gpv* (*I-uniform UNIV* (*range h*)) *x* ⊆ *B*} (*map-gpv' f g h*)
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *rel-gpv''-map-gpv'1*:
rel-gpv'' *A* *C* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *gpv gpv'* ⟹ *rel-gpv''* *A* *C* (=) (*map-gpv' id id h gpv*) *gpv'*
⟨*proof*⟩

lemma *rel-gpv''-map-gpv'2*:
rel-gpv'' *A* *C* (*eq-on* (*range h*)) *gpv gpv'* ⟹ *rel-gpv''* *A* *C* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *gpv* (*map-gpv' id id h gpv'*)
⟨*proof*⟩

context
fixes *A* :: *'a* ⇒ *'d* ⇒ *bool*
and *C* :: *'c* ⇒ *'g* ⇒ *bool*
and *R* :: *'b* ⇒ *'e* ⇒ *bool*
begin

private lemma *f11*: *Pure x* ∈ *set-spmf* (*the-gpv gpv*) ⟹
Domainp (*rel-generat A C* (*rel-fun R* (*rel-gpv'' A C R*))) (*Pure x*) ⟹ *Domainp A x*
⟨*proof*⟩ **lemma** *f21*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟹
rel-generat A C (*rel-fun R* (*rel-gpv'' A C R*)) (*IO out c*) *ba* ⟹ *Domainp C out*
⟨*proof*⟩ **lemma** *f12*:
assumes *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *input* ∈ *responses-I* (*I-uniform UNIV* {*x*. *Domainp R x*}) *out*
and *x* ∈ *results-gpv* (*I-uniform UNIV* {*x*. *Domainp R x*}) (*c input*)
and *Domainp* (*rel-gpv'' A C R*) *gpv*
shows *Domainp* (*rel-gpv'' A C R*) (*c input*)
⟨*proof*⟩ **lemma** *f22*:
assumes *IO out' rpv* ∈ *set-spmf* (*the-gpv gpv*)
and *input* ∈ *responses-I* (*I-uniform UNIV* {*x*. *Domainp R x*}) *out'*
and *out* ∈ *outs-gpv* (*I-uniform UNIV* {*x*. *Domainp R x*}) (*rpv input*)
and *Domainp* (*rel-gpv'' A C R*) *gpv*
shows *Domainp* (*rel-gpv'' A C R*) (*rpv input*)
⟨*proof*⟩

lemma *Domainp-rel-gpv''-le*:

Domainp (*rel-gpv''* *A* *C* *R*) ≤ *pred-gpv'* (*Domainp* *A*) (*Domainp* *C*) {*x*. *Domainp* *R* *x*}
⟨*proof*⟩

end

lemma *map-gpv'-id12*: *map-gpv'* *f* *g* *h* *gpv* = *map-gpv'* *id* *id* *h* (*map-gpv* *f* *g* *gpv*)
⟨*proof*⟩

lemma *rel-gpv''-refl*: ⟦ (=) ≤ *A*; (=) ≤ *C*; *R* ≤ (=) ⟧ ⟹ (=) ≤ *rel-gpv''* *A* *C* *R*
⟨*proof*⟩

context
fixes *A* *A'* :: *'a* ⇒ *'b* ⇒ *bool*
and *C* *C'* :: *'c* ⇒ *'d* ⇒ *bool*
and *R* *R'* :: *'e* ⇒ *'f* ⇒ *bool*

begin

private abbreviation *foo* **where**
foo ≡ (λ *fx* *fy* *gpvx* *gpvy* *g1* *g2*.
∀ *x* *y*. *x* ∈ *fx* (*I-uniform* *UNIV* (*Collect* (*Domainp* *R'*))) *gpvx* ⟶
y ∈ *fy* (*I-uniform* *UNIV* (*Collect* (*Rangep* *R'*))) *gpvy* ⟶ *g1* *x* *y* ⟶ *g2* *x* *y*)

private lemma *f1*: *foo* *results-gpv* *results-gpv* *gpv* *gpv'* *A* *A'* ⟹
x ∈ *set-spmf* (*the-gpv* *gpv*) ⟹ *y* ∈ *set-spmf* (*the-gpv* *gpv'*) ⟹
a ∈ *generat-conts* *x* ⟹ *b* ∈ *generat-conts* *y* ⟹ *R'* *a'* *α* ⟹ *R'* *β* *b'* ⟹
foo *results-gpv* *results-gpv* (*a* *a'*) (*b* *b'*) *A* *A'*
⟨*proof*⟩ **lemma** *f2*: *foo* *outs-gpv* *outs-gpv* *gpv* *gpv'* *C* *C'* ⟹
x ∈ *set-spmf* (*the-gpv* *gpv*) ⟹ *y* ∈ *set-spmf* (*the-gpv* *gpv'*) ⟹
a ∈ *generat-conts* *x* ⟹ *b* ∈ *generat-conts* *y* ⟹ *R'* *a'* *α* ⟹ *R'* *β* *b'* ⟹
foo *outs-gpv* *outs-gpv* (*a* *a'*) (*b* *b'*) *C* *C'*
⟨*proof*⟩

lemma *rel-gpv''-mono-strong*:
⟦ *rel-gpv''* *A* *C* *R* *gpv* *gpv'*;
⋀*x* *y*. ⟦ *x* ∈ *results-gpv* (*I-uniform* *UNIV* {*x*. *Domainp* *R'* *x*}) *gpv*; *y* ∈ *results-gpv* (*I-uniform* *UNIV* {*x*. *Rangep* *R'* *x*}) *gpv'*; *A* *x* *y* ⟧ ⟹ *A'* *x* *y*;
⋀*x* *y*. ⟦ *x* ∈ *outs-gpv* (*I-uniform* *UNIV* {*x*. *Domainp* *R'* *x*}) *gpv*; *y* ∈ *outs-gpv* (*I-uniform* *UNIV* {*x*. *Rangep* *R'* *x*}) *gpv'*; *C* *x* *y* ⟧ ⟹ *C'* *x* *y*;
R' ≤ *R* ⟧
⟹ *rel-gpv''* *A'* *C'* *R'* *gpv* *gpv'*
⟨*proof*⟩

end

lemma *rel-gpv''-refl-strong*:
assumes $\bigwedge x.\ x \in$ *results-gpv* ($\mathcal{I}$*-uniform UNIV* {x. *Domainp R x*}) *gpv* $\Longrightarrow$ *A x x*
and $\bigwedge x.\ x \in$ *outs-gpv* ($\mathcal{I}$*-uniform UNIV* {x. *Domainp R x*}) *gpv* $\Longrightarrow$ *C x x*
and $R \leq (=)$
shows *rel-gpv'' A C R gpv gpv*
⟨*proof*⟩

lemma *rel-gpv''-refl-eq-on*:
⟦ $\bigwedge x.\ x \in$ *results-gpv* ($\mathcal{I}$*-uniform UNIV X*) *gpv* $\Longrightarrow$ *A x x*; $\bigwedge$*out. out* $\in$ *outs-gpv* ($\mathcal{I}$*-uniform UNIV X*) *gpv* $\Longrightarrow$ *B out out* ⟧
$\Longrightarrow$ *rel-gpv'' A B* (*eq-on X*) *gpv gpv*
⟨*proof*⟩

lemma *pred-gpv'-mono'* [*mono*]:
pred-gpv' A C R gpv $\longrightarrow$ *pred-gpv' A' C' R gpv*
if $\bigwedge x.\ A\ x \longrightarrow A'\ x$ $\bigwedge x.\ C\ x \longrightarrow C'\ x$
⟨*proof*⟩

primcorec *enforce-*$\mathcal{I}$*-gpv* :: (*'out*, *'in*) $\mathcal{I}$ $\Rightarrow$ (*'a*, *'out*, *'in*) *gpv* $\Rightarrow$ (*'a*, *'out*, *'in*) *gpv* **where**
enforce-$\mathcal{I}$*-gpv* $\mathcal{I}$ *gpv* = *GPV*
(*map-spmf* (*map-generat id id* ((∘) (*enforce-*$\mathcal{I}$*-gpv* $\mathcal{I}$)))
(*map-spmf* (λ*generat. case generat of Pure x* ⇒ *Pure x* | *IO out rpv* ⇒ *IO out* (λ*input. if input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out then rpv input else Fail*))
(*enforce-spmf* (*pred-generat* ⊤ (λx. $x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$) ⊤) (*the-gpv gpv*))))

lemma *enforce-*$\mathcal{I}$*-gpv-Done* [*simp*]: *enforce-*$\mathcal{I}$*-gpv* $\mathcal{I}$ (*Done x*) = *Done x*
⟨*proof*⟩

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

lemma *enforce-*$\mathcal{I}$*-gpv-Pause* [*simp*]:
enforce-$\mathcal{I}$*-gpv* $\mathcal{I}$ (*Pause out rpv*) =
(*if out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ *then Pause out* (λ*input. if input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out then enforce-*$\mathcal{I}$*-gpv* $\mathcal{I}$ (*rpv input*) *else Fail*) *else Fail*)
⟨*proof*⟩

lemma *enforce-*$\mathcal{I}$*-gpv-lift-spmf* [*simp*]: *enforce-*$\mathcal{I}$*-gpv* $\mathcal{I}$ (*lift-spmf p*) = *lift-spmf p*
⟨*proof*⟩

lemma *enforce-*$\mathcal{I}$*-gpv-bind-gpv* [*simp*]:
enforce-$\mathcal{I}$*-gpv* $\mathcal{I}$ (*bind-gpv gpv f*) = *bind-gpv* (*enforce-*$\mathcal{I}$*-gpv* $\mathcal{I}$ *gpv*) (*enforce-*$\mathcal{I}$*-gpv* $\mathcal{I}$ ∘ *f*)
⟨*proof*⟩

lemma *enforce-$\mathcal{I}$-gpv-parametric'*:
includes *lifting-syntax*
notes [*transfer-rule*] = *corec-gpv-parametric' the-gpv-parametric' Fail-parametric'*
assumes [*transfer-rule*]: *bi-unique C bi-unique R*
shows (*rel-$\mathcal{I}$ C R* ===> *rel-gpv'' A C R* ===> *rel-gpv'' A C R*) *enforce-$\mathcal{I}$-gpv enforce-$\mathcal{I}$-gpv*
⟨*proof*⟩

lemma *enforce-$\mathcal{I}$-gpv-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
bi-unique C ⟹ (*rel-$\mathcal{I}$ C* (=) ===> *rel-gpv A C* ===> *rel-gpv A C*) *enforce-$\mathcal{I}$-gpv enforce-$\mathcal{I}$-gpv*
⟨*proof*⟩

lemma *WT-enforce-$\mathcal{I}$-gpv* [*simp*]: *$\mathcal{I}$ ⊢g enforce-$\mathcal{I}$-gpv $\mathcal{I}$ gpv* √
⟨*proof*⟩

lemma *WT-gpv-parametric'*: **includes** *lifting-syntax* **shows**
bi-unique C ⟹ (*rel-$\mathcal{I}$ C R* ===> *rel-gpv'' A C R* ===> (=)) *WT-gpv WT-gpv*
⟨*proof*⟩

lemma *WT-gpv-map-gpv-id* [*simp*]: *$\mathcal{I}$ ⊢g map-gpv f id gpv* √ ⟷ *$\mathcal{I}$ ⊢g gpv* √
⟨*proof*⟩

locale *raw-converter-invariant* =
fixes *$\mathcal{I}$* :: ('*call*, '*ret*) *$\mathcal{I}$*
and *$\mathcal{I}'$* :: ('*call'*, '*ret'*) *$\mathcal{I}$*
and *callee* :: '*s* ⇒ '*call* ⇒ ('*ret* × '*s*, '*call'*, '*ret'*) *gpv*
and *I* :: '*s* ⇒ *bool*
assumes *results-callee*: ⋀*s x*. ⟦ *x* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$*; *I s* ⟧ ⟹ *results-gpv $\mathcal{I}'$* (*callee s x*) ⊆ *responses-$\mathcal{I}$ $\mathcal{I}$ x* × {*s*. *I s*}
and *WT-callee*: ⋀*x s*. ⟦ *x* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$*; *I s* ⟧ ⟹ *$\mathcal{I}'$ ⊢g callee s x* √
begin

context begin
private lemma *aux*:
set-spmf (*inline1 callee gpv s*) ⊆ {*Inr* (*out*, *callee'*, *rpv'*) | *out callee' rpv'*.
∃ *call*∈*outs-$\mathcal{I}$ $\mathcal{I}$*. ∃ *s*. *I s* ∧ (∀ *x* ∈ *responses-$\mathcal{I}$ $\mathcal{I}'$ out*. *callee' x* ∈ *sub-gpvs $\mathcal{I}'$* (*callee s call*))} ∪
{*Inl* (*x*, *s'*) | *x s'*. *x* ∈ *results-gpv $\mathcal{I}$ gpv* ∧ *I s'*}
(**is** *?concl* (*inline1 callee*) *gpv s* **is** - ⊆ *?rhs1* ∪ *?rhs2 gpv*)
if *$\mathcal{I}$ ⊢g gpv* √ *I s*
⟨*proof*⟩

lemma *inline1-in-sub-gpvs-callee*:
assumes *Inr* (*out*, *callee'*, *rpv'*) ∈ *set-spmf* (*inline1 callee gpv s*)
and *WT*: *$\mathcal{I}$ ⊢g gpv* √
and *s*: *I s*
shows ∃ *call*∈*outs-$\mathcal{I}$ $\mathcal{I}$*. ∃ *s*. *I s* ∧ (∀ *x* ∈ *responses-$\mathcal{I}$ $\mathcal{I}'$ out*. *callee' x* ∈ *sub-gpvs $\mathcal{I}'$* (*callee s call*))

⟨*proof*⟩

lemma *inline1-Inl-results-gpv*:
assumes *Inl* (*x*, *s'*) ∈ *set-spmf* (*inline1 callee gpv s*)
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *s*: *I s*
shows *x* ∈ *results-gpv* $\mathcal{I}$ *gpv* ∧ *I s'*
⟨*proof*⟩
end

lemma *inline1-in-sub-gpvs*:
assumes *Inr* (*out*, *callee'*, *rpv'*) ∈ *set-spmf* (*inline1 callee gpv s*)
and (*x*, *s'*) ∈ *results-gpv* $\mathcal{I}'$ (*callee' input*)
and *input* ∈ *responses-I* $\mathcal{I}'$ *out*
and $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I s*
shows *rpv' x* ∈ *sub-gpvs* $\mathcal{I}$ *gpv* ∧ *I s'*
⟨*proof*⟩

lemma *WT-gpv-inline1*:
assumes *Inr* (*out*, *rpv*, *rpv'*) ∈ *set-spmf* (*inline1 callee gpv s*)
and $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I s*
shows *out* ∈ *outs-I* $\mathcal{I}'$ (**is** *?thesis1*)
and *input* ∈ *responses-I* $\mathcal{I}'$ *out* ⟹ $\mathcal{I}' \vdash g$ *rpv input* $\surd$ (**is** *PROP ?thesis2*)
and ⟦ *input* ∈ *responses-I* $\mathcal{I}'$ *out*; (*x*, *s'*) ∈ *results-gpv* $\mathcal{I}'$ (*rpv input*) ⟧ ⟹ $\mathcal{I}$ $\vdash g$ *rpv' x* $\surd$ ∧ *I s'* (**is** *PROP ?thesis3*)
⟨*proof*⟩

lemma *WT-gpv-inline-invar*:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I s*
shows $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$
⟨*proof*⟩

end

lemma *WT-gpv-inline*:
assumes ⋀*s x*. *x* ∈ *outs-I* $\mathcal{I}$ $\mathcal{I}$ ⟹ *results-gpv* $\mathcal{I}'$ (*callee s x*) ⊆ *responses-I* $\mathcal{I}$ *x* × *UNIV*
and ⋀*x s*. *x* ∈ *outs-I* $\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}' \vdash g$ *callee s x* $\surd$
and $\mathcal{I} \vdash g$ *gpv* $\surd$
shows $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$
⟨*proof*⟩

lemma *results-gpv-sub-gvps*: *gpv'* ∈ *sub-gpvs* $\mathcal{I}$ *gpv* ⟹ *results-gpv* $\mathcal{I}$ *gpv'* ⊆ *results-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *in-results-gpv-sub-gvps*: ⟦ $x \in$ *results-gpv* $\mathcal{I}$ *gpv'*; *gpv'* $\in$ *sub-gpvs* $\mathcal{I}$ *gpv* ⟧ $\Longrightarrow$ $x \in$ *results-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

context *raw-converter-invariant* **begin**
lemma *results-gpv-inline-aux*:
assumes (*x*, *s'*) $\in$ *results-gpv* $\mathcal{I}'$ (*inline-aux callee y*)
shows ⟦ *y* = *Inl* (*gpv*, *s*); $\mathcal{I} \vdash g$ *gpv* $\surd$; *I s* ⟧ $\Longrightarrow$ *x* $\in$ *results-gpv* $\mathcal{I}$ *gpv* $\wedge$ *I s'*
and ⟦ *y* = *Inr* (*rpv*, *callee'*); $\forall$(*z*, *s'*) $\in$ *results-gpv* $\mathcal{I}'$ *callee'*. $\mathcal{I} \vdash g$ *rpv z* $\surd$ $\wedge$ *I s'* ⟧
$\Longrightarrow$ $\exists$(*z*, *s''*) $\in$ *results-gpv* $\mathcal{I}'$ *callee'*. *x* $\in$ *results-gpv* $\mathcal{I}$ (*rpv z*) $\wedge$ *I s''* $\wedge$ *I s'*
⟨*proof*⟩

lemma *results-gpv-inline*:
⟦(*x*, *s'*) $\in$ *results-gpv* $\mathcal{I}'$ (*inline callee gpv s*); $\mathcal{I} \vdash g$ *gpv* $\surd$; *I s*⟧ $\Longrightarrow$ *x* $\in$ *results-gpv* $\mathcal{I}$ *gpv* $\wedge$ *I s'*
⟨*proof*⟩

end

lemma *inline-map-gpv*:
inline callee (*map-gpv f g gpv*) *s* = *map-gpv* (*apfst f*) *id* (*inline* (λ*s x*. *callee s* (*g x*)) *gpv s*)
⟨*proof*⟩

lemma $\mathcal{I}$*-full-le-plus-*$\mathcal{I}$: $\mathcal{I}$*-full* $\leq$ *plus-*$\mathcal{I}$ $\mathcal{I}$*1* $\mathcal{I}$*2* **if** $\mathcal{I}$*-full* $\leq$ $\mathcal{I}$*1* $\mathcal{I}$*-full* $\leq$ $\mathcal{I}$*2*
⟨*proof*⟩

lemma *plus-*$\mathcal{I}$*-mono*: *plus-*$\mathcal{I}$ $\mathcal{I}$*1* $\mathcal{I}$*2* $\leq$ *plus-*$\mathcal{I}$ $\mathcal{I}$*1'* $\mathcal{I}$*2'* **if** $\mathcal{I}$*1* $\leq$ $\mathcal{I}$*1'* $\mathcal{I}$*2* $\leq$ $\mathcal{I}$*2'*
⟨*proof*⟩

primcorec (*transfer*) *left-gpv* :: ('*a*, '*out*, '*in*) *gpv* $\Rightarrow$ ('*a*, '*out* + '*out'*, '*in* + '*in'*) *gpv* **where**
the-gpv (*left-gpv gpv*) =
map-spmf (*map-generat id Inl* (λ*rpv input*. *case input of Inl input'* $\Rightarrow$ *left-gpv* (*rpv input'*) | - $\Rightarrow$ *Fail*)) (*the-gpv gpv*)

abbreviation *left-rpv* :: ('*a*, '*out*, '*in*) *rpv* $\Rightarrow$ ('*a*, '*out* + '*out'*, '*in* + '*in'*) *rpv* **where**
left-rpv rpv $\equiv$ λ*input*. *case input of Inl input'* $\Rightarrow$ *left-gpv* (*rpv input'*) | - $\Rightarrow$ *Fail*

primcorec (*transfer*) *right-gpv* :: ('*a*, '*out*, '*in*) *gpv* $\Rightarrow$ ('*a*, '*out'* + '*out*, '*in'* + '*in*) *gpv* **where**
the-gpv (*right-gpv gpv*) =

map-spmf (*map-generat id Inr* (λ*rpv input*. *case input of Inr input'* ⇒ *right-gpv* (*rpv input'*) | - ⇒ *Fail*)) (*the-gpv gpv*)

abbreviation *right-rpv* :: ('*a*, '*out*, '*in*) *rpv* ⇒ ('*a*, '*out'* + '*out*, '*in'* + '*in*) *rpv*
where
right-rpv rpv ≡ λ*input*. *case input of Inr input'* ⇒ *right-gpv* (*rpv input'*) | - ⇒ *Fail*

context
includes *lifting-syntax*
notes [*transfer-rule*] = *corec-gpv-parametric' Fail-parametric' the-gpv-parametric'*
begin

lemmas *left-gpv-parametric* = *left-gpv.transfer*

lemma *left-gpv-parametric'*:
(*rel-gpv''* *A* *C* *R* ===> *rel-gpv''* *A* (*rel-sum* *C* *C'*) (*rel-sum* *R* *R'*)) *left-gpv* *left-gpv*
⟨*proof*⟩

lemmas *right-gpv-parametric* = *right-gpv.transfer*

lemma *right-gpv-parametric'*:
(*rel-gpv''* *A* *C'* *R'* ===> *rel-gpv''* *A* (*rel-sum* *C* *C'*) (*rel-sum* *R* *R'*)) *right-gpv* *right-gpv*
⟨*proof*⟩

end

lemma *left-gpv-Done* [*simp*]: *left-gpv* (*Done x*) = *Done x*
⟨*proof*⟩

lemma *right-gpv-Done* [*simp*]: *right-gpv* (*Done x*) = *Done x*
⟨*proof*⟩

lemma *left-gpv-Pause* [*simp*]:
left-gpv (*Pause x rpv*) = *Pause* (*Inl x*) (λ*input*. *case input of Inl input'* ⇒ *left-gpv* (*rpv input'*) | - ⇒ *Fail*)
⟨*proof*⟩

lemma *right-gpv-Pause* [*simp*]:
right-gpv (*Pause x rpv*) = *Pause* (*Inr x*) (λ*input*. *case input of Inr input'* ⇒ *right-gpv* (*rpv input'*) | - ⇒ *Fail*)
⟨*proof*⟩

lemma *left-gpv-map*: *left-gpv* (*map-gpv f g gpv*) = *map-gpv f* (*map-sum g h*) (*left-gpv gpv*)
⟨*proof*⟩

lemma *right-gpv-map*: *right-gpv* (*map-gpv f g gpv*) = *map-gpv f* (*map-sum h g*) (*right-gpv gpv*)
⟨*proof*⟩

lemma *results'-gpv-left-gpv* [*simp*]:
results'-gpv (*left-gpv gpv* :: ('*a*, '*out* + '*out'*, '*in* + '*in'*) *gpv*) = *results'-gpv gpv*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *results'-gpv-right-gpv* [*simp*]:
results'-gpv (*right-gpv gpv* :: ('*a*, '*out'* + '*out*, '*in'* + '*in*) *gpv*) = *results'-gpv gpv*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *left-gpv-Inl-transfer*: *rel-gpv''* (=) (λ*l r*. *l* = *Inl r*) (λ*l r*. *l* = *Inl r*) (*left-gpv gpv*) *gpv*
⟨*proof*⟩

lemma *right-gpv-Inr-transfer*: *rel-gpv''* (=) (λ*l r*. *l* = *Inr r*) (λ*l r*. *l* = *Inr r*) (*right-gpv gpv*) *gpv*
⟨*proof*⟩

lemma *exec-gpv-plus-oracle-left*: *exec-gpv* (*plus-oracle oracle1 oracle2*) (*left-gpv gpv*) *s* = *exec-gpv oracle1 gpv s*
⟨*proof*⟩

lemma *exec-gpv-plus-oracle-right*: *exec-gpv* (*plus-oracle oracle1 oracle2*) (*right-gpv gpv*) *s* = *exec-gpv oracle2 gpv s*
⟨*proof*⟩

lemma *left-gpv-bind-gpv*: *left-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*left-gpv gpv*) (*left-gpv* ∘ *f*)
⟨*proof*⟩

lemma *inline1-left-gpv*:
inline1 (λ*s q*. *left-gpv* (*callee s q*)) *gpv s* =
map-spmf (*map-sum id* (*map-prod Inl* (*map-prod left-rpv id*))) (*inline1 callee gpv s*)
⟨*proof*⟩

lemma *left-gpv-inline*: *left-gpv* (*inline callee gpv s*) = *inline* (λ*s q*. *left-gpv* (*callee s q*)) *gpv s*
⟨*proof*⟩

lemma *right-gpv-bind-gpv*: *right-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*right-gpv gpv*) (*right-gpv* ∘ *f*)
⟨*proof*⟩

lemma *inline1-right-gpv*:

inline1 (λ*s q*. *right-gpv* (*callee s q*)) *gpv s* =
map-spmf (*map-sum id* (*map-prod Inr* (*map-prod right-rpv id*))) (*inline1 callee gpv s*)
⟨*proof*⟩

lemma *right-gpv-inline*: *right-gpv* (*inline callee gpv s*) = *inline* (λ*s q*. *right-gpv* (*callee s q*)) *gpv s*
⟨*proof*⟩

lemma *WT-gpv-left-gpv*: $\mathcal{I}1 \vdash g$ *gpv* $\surd \Longrightarrow \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash g$ *left-gpv gpv* $\surd$
⟨*proof*⟩

lemma *WT-gpv-right-gpv*: $\mathcal{I}2 \vdash g$ *gpv* $\surd \Longrightarrow \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash g$ *right-gpv gpv* $\surd$
⟨*proof*⟩

lemma *results-gpv-left-gpv* [*simp*]: *results-gpv* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) (*left-gpv gpv*) = *results-gpv* $\mathcal{I}1$ *gpv*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *results-gpv-right-gpv* [*simp*]: *results-gpv* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) (*right-gpv gpv*) = *results-gpv* $\mathcal{I}2$ *gpv*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *left-gpv-Fail* [*simp*]: *left-gpv Fail* = *Fail*
⟨*proof*⟩

lemma *right-gpv-Fail* [*simp*]: *right-gpv Fail* = *Fail*
⟨*proof*⟩

lemma *rsuml-lsumr-left-gpv-left-gpv*:*map-gpv′ id rsuml lsumr* (*left-gpv* (*left-gpv gpv*)) = *left-gpv gpv*
⟨*proof*⟩

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

lemma *rsuml-lsumr-right-gpv*: *map-gpv′ id rsuml lsumr* (*right-gpv gpv*) = *right-gpv* (*right-gpv gpv*)
⟨*proof*⟩

lemma *map-gpv′-map-gpv-swap*:
map-gpv′ f g h (*map-gpv f′ id gpv*) = *map-gpv* (*f* ∘ *f′*) *id* (*map-gpv′ id g h gpv*)
⟨*proof*⟩

lemma *lsumr-rsuml-left-gpv*: *map-gpv′ id lsumr rsuml* (*left-gpv gpv*) = *left-gpv* (*left-gpv gpv*)

⟨*proof*⟩

lemma *lsumr-rsuml-right-gpv-left-gpv*:
map-gpv' *id* *lsumr* *rsuml* (*right-gpv* (*left-gpv* *gpv*)) = *left-gpv* (*right-gpv* *gpv*)
⟨*proof*⟩

lemma *lsumr-rsuml-right-gpv-right-gpv*:
map-gpv' *id* *lsumr* *rsuml* (*right-gpv* (*right-gpv* *gpv*)) = *right-gpv* *gpv*
⟨*proof*⟩

lemma *in-set-spmf-extend-state-oracle* [*simp*]:
x ∈ *set-spmf* (*extend-state-oracle* *oracle* *s* *y*) ⟷
fst (*snd* *x*) = *fst* *s* ∧ (*fst* *x*, *snd* (*snd* *x*)) ∈ *set-spmf* (*oracle* (*snd* *s*) *y*)
⟨*proof*⟩

lemma *extend-state-oracle-plus-oracle*:
extend-state-oracle (*plus-oracle* *oracle1* *oracle2*) = *plus-oracle* (*extend-state-oracle* *oracle1*) (*extend-state-oracle* *oracle2*)
⟨*proof*⟩

definition *stateless-callee* :: ('*a* ⇒ ('*b*, '*out*, '*in*) *gpv*) ⇒ ('*s* ⇒ '*a* ⇒ ('*b* × '*s*, '*out*, '*in*) *gpv*) **where**
stateless-callee *callee* *s* = *map-gpv* (λ*b*. (*b*, *s*)) *id* ∘ *callee*

lemma *stateless-callee-parametric'*:
includes *lifting-syntax* **notes** [*transfer-rule*] = *map-gpv-parametric'* **shows**
((*A* ===> *rel-gpv''* *B* *C* *R*) ===> *S* ===> *A* ===> (*rel-gpv''* (*rel-prod* *B* *S*) *C* *R*))
stateless-callee *stateless-callee*
⟨*proof*⟩

lemma *id-oralce-alt-def*: *id-oracle* = *stateless-callee* (λ*x*. *Pause* *x* *Done*)
⟨*proof*⟩

context
fixes *left* :: '*s1* ⇒ '*x1* ⇒ ('*y1* × '*s1*, '*call1*, '*ret1*) *gpv*
and *right* :: '*s2* ⇒ '*x2* ⇒ ('*y2* × '*s2*, '*call2*, '*ret2*) *gpv*
begin

fun *parallel-intercept* :: '*s1* × '*s2* ⇒ '*x1* + '*x2* ⇒ (('*y1* + '*y2*) × ('*s1* × '*s2*), '*call1* + '*call2*, '*ret1* + '*ret2*) *gpv*
where
parallel-intercept (*s1*, *s2*) (*Inl* *a*) = *left-gpv* (*map-gpv* (*map-prod* *Inl* (λ*s1'*. (*s1'*, *s2*))) *id* (*left* *s1* *a*))
| *parallel-intercept* (*s1*, *s2*) (*Inr* *b*) = *right-gpv* (*map-gpv* (*map-prod* *Inr* (*Pair* *s1*)) *id* (*right* *s2* *b*))

end

lemma *expectation-gpv-$\mathcal{I}$-mono*:
defines *expectation-gpv'* $\equiv$ *expectation-gpv*
assumes *le*: $\mathcal{I} \leq \mathcal{I}'$
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *expectation-gpv fail* $\mathcal{I}$ *f gpv* $\leq$ *expectation-gpv' fail* $\mathcal{I}'$ *f gpv*
⟨*proof*⟩

lemma *pgen-lossless-gpv-mono*:
assumes $*$: *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv*
and *le*: $\mathcal{I} \leq \mathcal{I}'$
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *fail*: *fail* $\leq$ *1*
shows *pgen-lossless-gpv fail* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

lemma *plossless-gpv-mono*:
$[\![$ *plossless-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g$ *gpv* $\surd$ $]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

lemma *pfinite-gpv-mono*:
$[\![$ *pfinite-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g$ *gpv* $\surd$ $]\!] \Longrightarrow$ *pfinite-gpv* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

lemma *pgen-lossless-gpv-parametric'*: **includes** *lifting-syntax* **shows**
((=) ===> *rel-*$\mathcal{I}$ *C R* ===> *rel-gpv''* *A C R* ===> (=)) *pgen-lossless-gpv pgen-lossless-gpv*
⟨*proof*⟩

lemma *pgen-lossless-gpv-parametric*: **includes** *lifting-syntax* **shows**
((=) ===> *rel-*$\mathcal{I}$ *C* (=) ===> *rel-gpv A C* ===> (=)) *pgen-lossless-gpv pgen-lossless-gpv*
⟨*proof*⟩

lemma *pgen-lossless-gpv-map-gpv-id* [*simp*]:
pgen-lossless-gpv fail $\mathcal{I}$ (*map-gpv f id gpv*) = *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv*
⟨*proof*⟩

context *raw-converter-invariant* **begin**

lemma *expectation-gpv-le-inline*:
defines *expectation-gpv2* $\equiv$ *expectation-gpv 0* $\mathcal{I}'$
assumes *callee*: $\bigwedge s\ x.\ [\![\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* $]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I*: *I s*
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* $\leq$ *expectation-gpv2* ($\lambda(x,\ s).\ f\ x$) (*inline callee gpv s*)

⟨*proof*⟩

lemma *plossless-inline*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ [\![\ I\ s;\ x \in$ *outs-*$\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
⟨*proof*⟩

end

lemma *expectation-left-gpv* [*simp*]:
expectation-gpv fail ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) *f* (*left-gpv gpv*) = *expectation-gpv fail* $\mathcal{I}$ *f gpv*
⟨*proof*⟩

lemma *expectation-right-gpv* [*simp*]:
expectation-gpv fail ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) *f* (*right-gpv gpv*) = *expectation-gpv fail* $\mathcal{I}'$ *f gpv*
⟨*proof*⟩

lemma *pgen-lossless-left-gpv* [*simp*]: *pgen-lossless-gpv fail* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*left-gpv gpv*) = *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *pgen-lossless-right-gpv* [*simp*]: *pgen-lossless-gpv fail* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*right-gpv gpv*) = *pgen-lossless-gpv fail* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

lemma (**in** *raw-converter-invariant*) *expectation-gpv-le-inline-invariant*:
defines *expectation-gpv2* ≡ *expectation-gpv 0* $\mathcal{I}'$
assumes *callee*: $\bigwedge s\ x.\ [\![\ x \in$ *outs-*$\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I*: *I s*
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv2* ($\lambda(x,\ s).\ f\ x$) (*inline callee gpv s*)
⟨*proof*⟩

lemma (**in** *raw-converter-invariant*) *plossless-inline-invariant*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ [\![\ x \in$ *outs-*$\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
⟨*proof*⟩

context *callee-invariant-on* **begin**

lemma *raw-converter-invariant*: *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ ($\lambda s\ x.$ *lift-spmf* (*callee s x*)) *I*

⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *plossless-exec-gpv*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ [\![\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* $]\!] \Longrightarrow$ *lossless-spmf* (*callee s x*)
and *I*: *I s*
shows *lossless-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

end

definition *extend-state-oracle2* :: ('*call*, '*ret*, '*s*) *callee* ⇒ ('*call*, '*ret*, '*s* × '*s*′) *callee* (-† [*1000*] *1000*)
where *extend-state-oracle2 callee* = (λ(*s*, *s*′) *x*. *map-spmf* (λ(*y*, *s*). (*y*, (*s*, *s*′))) (*callee s x*))

lemma *extend-state-oracle2-simps* [*simp*]:
extend-state-oracle2 callee (*s*, *s*′) *x* = *map-spmf* (λ(*y*, *s*). (*y*, (*s*, *s*′))) (*callee s x*)
⟨*proof*⟩

lemma *extend-state-oracle2-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) ===> *rel-prod S S*′ ===> *C* ===> *rel-spmf* (*rel-prod R* (*rel-prod S S*′)))
extend-state-oracle2 extend-state-oracle2
⟨*proof*⟩

lemma *callee-invariant-extend-state-oracle2-const* [*simp*]:
callee-invariant oracle† (λ(*s*, *s*′). *I s*′)
⟨*proof*⟩

lemma *callee-invariant-extend-state-oracle2-const*′:
callee-invariant oracle† (λ*s*. *I* (*snd s*))
⟨*proof*⟩

lemma *extend-state-oracle2-plus-oracle*:
extend-state-oracle2 (*plus-oracle oracle1 oracle2*) = *plus-oracle* (*extend-state-oracle2 oracle1*) (*extend-state-oracle2 oracle2*)
⟨*proof*⟩

lemma *parallel-oracle-conv-plus-oracle*:
parallel-oracle oracle1 oracle2 = *plus-oracle* (*oracle1*†) (†*oracle2*)
⟨*proof*⟩

lemma *map-sum-parallel-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *map-sum f g* −−−> *map-spmf* (*map-prod* (*map-sum h k*) *id*)) (*parallel-oracle oracle1 oracle2*)
= *parallel-oracle* ((*id* −−−> *f* −−−> *map-spmf* (*map-prod h id*)) *oracle1*) ((*id*

$---> g ---> map\text{-}spmf\ (map\text{-}prod\ k\ id))\ oracle2)$
⟨*proof*⟩

lemma *map-sum-plus-oracle*: **includes** *lifting-syntax* **shows**
$(id ---> map\text{-}sum\ f\ g ---> map\text{-}spmf\ (map\text{-}prod\ (map\text{-}sum\ h\ k)\ id))$
$(plus\text{-}oracle\ oracle1\ oracle2)$
$= plus\text{-}oracle\ ((id ---> f ---> map\text{-}spmf\ (map\text{-}prod\ h\ id))\ oracle1)\ ((id$
$---> g ---> map\text{-}spmf\ (map\text{-}prod\ k\ id))\ oracle2)$
⟨*proof*⟩

lemma *map-rsuml-plus-oracle*: **includes** *lifting-syntax* **shows**
$(id ---> rsuml ---> (map\text{-}spmf\ (map\text{-}prod\ lsumr\ id)))\ (oracle1 \oplus_O (oracle2$
$\oplus_O\ oracle3)) =$
$((oracle1 \oplus_O\ oracle2) \oplus_O\ oracle3)$
⟨*proof*⟩

lemma *map-lsumr-plus-oracle*: **includes** *lifting-syntax* **shows**
$(id ---> lsumr ---> (map\text{-}spmf\ (map\text{-}prod\ rsuml\ id)))\ ((oracle1 \oplus_O\ ora$-
$cle2) \oplus_O\ oracle3) =$
$(oracle1 \oplus_O (oracle2 \oplus_O\ oracle3))$
⟨*proof*⟩

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

definition *lift-state-oracle*
$:: (('s \Rightarrow 'a \Rightarrow (('b \times 't) \times 's)\ spmf) \Rightarrow ('s' \Rightarrow 'a \Rightarrow (('b \times 't) \times 's')\ spmf))$
$\Rightarrow ('t \times 's \Rightarrow 'a \Rightarrow ('b \times 't \times 's)\ spmf) \Rightarrow ('t \times 's' \Rightarrow 'a \Rightarrow ('b \times 't \times 's')$
$spmf)$ **where**
$lift\text{-}state\text{-}oracle\ F\ oracle =$
$(\lambda(t,\ s')\ a.\ map\text{-}spmf\ rprodl\ (F\ ((Pair\ t ---> id ---> map\text{-}spmf\ lprodr)$
$oracle)\ s'\ a))$

lemma *lift-state-oracle-simps* [*simp*]:
$lift\text{-}state\text{-}oracle\ F\ oracle\ (t,\ s')\ a = map\text{-}spmf\ rprodl\ (F\ ((Pair\ t ---> id$
$---> map\text{-}spmf\ lprodr)\ oracle)\ s'\ a)$
⟨*proof*⟩

lemma *lift-state-oracle-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
$(((S ===> A ===> rel\text{-}spmf\ (rel\text{-}prod\ (rel\text{-}prod\ B\ T)\ S)) ===> S' ===>$
$A ===> rel\text{-}spmf\ (rel\text{-}prod\ (rel\text{-}prod\ B\ T)\ S'))$
$===> (rel\text{-}prod\ T\ S ===> A ===> rel\text{-}spmf\ (rel\text{-}prod\ B\ (rel\text{-}prod\ T\ S)))$
$===> rel\text{-}prod\ T\ S' ===> A ===> rel\text{-}spmf\ (rel\text{-}prod\ B\ (rel\text{-}prod\ T\ S')))$
$lift\text{-}state\text{-}oracle\ lift\text{-}state\text{-}oracle$
⟨*proof*⟩

lemma *lift-state-oracle-extend-state-oracle*:
includes *lifting-syntax*
assumes $\bigwedge B.\ Transfer.Rel\ (((=) ===> (=) ===> rel\text{-}spmf\ (rel\text{-}prod\ B\ (=)))$
$===> (=) ===> (=) ===> rel\text{-}spmf\ (rel\text{-}prod\ B\ (=)))\ G\ F$

shows *lift-state-oracle F (extend-state-oracle oracle) = extend-state-oracle (G oracle)*
⟨*proof*⟩

lemma *lift-state-oracle-compose*:
lift-state-oracle F (lift-state-oracle G oracle) = lift-state-oracle (F ∘ G) oracle
⟨*proof*⟩

lemma *lift-state-oracle-id* [*simp*]: *lift-state-oracle id = id*
⟨*proof*⟩

lemma *rprodl-extend-state-oracle*: **includes** *lifting-syntax* **shows**
(rprodl ---> id ---> map-spmf (map-prod id lprodr)) (extend-state-oracle (extend-state-oracle oracle)) =
extend-state-oracle oracle
⟨*proof*⟩

end

lemma *interaction-bound-map-gpv'*:
assumes *surj h*
shows *interaction-bound consider (map-gpv' f g h gpv) = interaction-bound (consider ∘ g) gpv*
⟨*proof*⟩

lemma *interaction-any-bounded-by-map-gpv'*:
assumes *interaction-any-bounded-by gpv n*
and *surj h*
shows *interaction-any-bounded-by (map-gpv' f g h gpv) n*
⟨*proof*⟩

lemma *results'-gpv-map-gpv'*:
assumes *surj h*
shows *results'-gpv (map-gpv' f g h gpv) = f ‘ results'-gpv gpv* (**is** *?lhs = ?rhs*)
⟨*proof*⟩

context fixes *B :: 'b ⇒ 'c set* **and** *x :: 'a* **begin**

primcorec *mk-lossless-gpv :: ('a, 'b, 'c) gpv ⇒ ('a, 'b, 'c) gpv* **where**
the-gpv (mk-lossless-gpv gpv) =
map-spmf (λgenerat. case generat of Pure x ⇒ Pure x
| IO out c ⇒ IO out (λinput. if input ∈ B out then mk-lossless-gpv (c input) else Done x))
(the-gpv gpv)

end

lemma *WT-gpv-outs-gpvI*:

assumes *outs-gpv* $\mathcal{I}$ *gpv* $\subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}$
shows $\mathcal{I} \vdash g$ *gpv* $\surd$
⟨*proof*⟩

lemma *WT-gpv-iff-outs-gpv*:
$\mathcal{I} \vdash g$ *gpv* $\surd \longleftrightarrow$ *outs-gpv* $\mathcal{I}$ *gpv* $\subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}$
⟨*proof*⟩

lemma *WT-gpv-mk-lossless-gpv*:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$
and *outs*: *outs-*$\mathcal{I}$ $\mathcal{I}'$ = *outs-*$\mathcal{I}$ $\mathcal{I}$
shows $\mathcal{I}' \vdash g$ *mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}$) *x gpv* $\surd$
⟨*proof*⟩

lemma *expectation-gpv-mk-lossless-gpv*:
fixes $\mathcal{I}$ *y*
defines *rhs* ≡ *expectation-gpv 0* $\mathcal{I}$ (λ-. *y*)
assumes *WT*: $\mathcal{I}' \vdash g$ *gpv* $\surd$
and *outs*: *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$
shows *expectation-gpv 0* $\mathcal{I}'$ (λ-. *y*) *gpv* ≤ *rhs* (*mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}'$) *x gpv*)
⟨*proof*⟩

lemma *plossless-gpv-mk-lossless-gpv*:
assumes *plossless-gpv* $\mathcal{I}$ *gpv*
and $\mathcal{I} \vdash g$ *gpv* $\surd$
and *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$
shows *plossless-gpv* $\mathcal{I}'$ (*mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}$) *x gpv*)
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *exec-gpv-mk-lossless-gpv*:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I s*
shows *exec-gpv callee* (*mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}$) *x gpv*) *s* = *exec-gpv callee gpv s*
⟨*proof*⟩

lemma *in-results-gpv-restrict-gpvD*:
assumes *x* ∈ *results-gpv* $\mathcal{I}$ (*restrict-gpv* $\mathcal{I}'$ *gpv*)
shows *x* ∈ *results-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *results-gpv-restrict-gpv*:
results-gpv $\mathcal{I}$ (*restrict-gpv* $\mathcal{I}'$ *gpv*) ⊆ *results-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *in-results′-gpv-restrict-gpvD*:
x ∈ *results′-gpv* (*restrict-gpv* $\mathcal{I}'$ *gpv*) ⟹ *x* ∈ *results′-gpv gpv*
⟨*proof*⟩

lemma *expectation-gpv-map-gpv'* [*simp*]:
expectation-gpv fail $\mathcal{I}$ *f* (*map-gpv'* *g h k gpv*) =
expectation-gpv fail (*map-*$\mathcal{I}$ *h k* $\mathcal{I}$) (*f* ∘ *g*) *gpv*
⟨*proof*⟩

lemma *plossless-gpv-map-gpv'* [*simp*]:
pgen-lossless-gpv b $\mathcal{I}$ (*map-gpv' f g h gpv*) ⟷ *pgen-lossless-gpv b* (*map-*$\mathcal{I}$ *g h* $\mathcal{I}$) *gpv*
⟨*proof*⟩

end
theory *Resource* **imports**
More-CryptHOL
begin

1 Resources

1.1 Type definition

codatatype ('*a*, '*b*) *resource*
= *Resource* (*run-resource*: '*a* ⇒ ('*b* × ('*a*, '*b*) *resource*) *spmf*)
for *map*: *map-resource'*
rel: *rel-resource'*

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

1.2 Functor

context
fixes *a* :: '*a* ⇒ '*a'*
and *b* :: '*b* ⇒ '*b'*
begin

primcorec *map-resource* :: ('*a'*, '*b*) *resource* ⇒ ('*a*, '*b'*) *resource* **where**
run-resource (*map-resource res*) = *map-spmf* (*map-prod b map-resource*) ∘ (*run-resource res*) ∘ *a*

lemma *map-resource-sel* [*simp*]:
run-resource (*map-resource res*) *a'* = *map-spmf* (*map-prod b map-resource*) (*run-resource res* (*a a'*))
⟨*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*) ∘ *f* ∘ *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′* ∘ *a*) (*b* ∘ *b′*) *res*
⟨*proof*⟩

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

1.3 Relator

coinductive *rel-resource* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*c* ⇒ '*d* ⇒ *bool*) ⇒ ('*a*, '*c*) *resource* ⇒ ('*b*, '*d*) *resource* ⇒ *bool*
for *A B* **where**
rel-resourceI:
rel-fun A (*rel-spmf* (*rel-prod B* (*rel-resource A B*))) (*run-resource res1*) (*run-resource res2*)
⟹ *rel-resource A B res1 res2*

lemma *rel-resource-coinduct* [*consumes 1*, *case-names rel-resource*, *coinduct pred*: *rel-resource*]:
assumes *X res1 res2*
and ⋀*res1 res2*. *X res1 res2* ⟹
rel-fun A (*rel-spmf* (*rel-prod B* (λ*res1 res2*. *X res1 res2* ∨ *rel-resource A B res1 res2*)))
(*run-resource res1*) (*run-resource res2*)
shows *rel-resource A B res1 res2*
⟨*proof*⟩

lemma *rel-resource-simps* [*simp*, *code*]:
rel-resource A B (*Resource f*) (*Resource g*) ⟷ *rel-fun A* (*rel-spmf* (*rel-prod B* (*rel-resource A B*))) *f g*
⟨*proof*⟩

lemma *rel-resourceD*:
rel-resource A B res1 res2 ⟹ *rel-fun A* (*rel-spmf* (*rel-prod B* (*rel-resource A B*))) (*run-resource res1*) (*run-resource res2*)
⟨*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* ≤ *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-resource A B (*map-resource' f res1*) *res2* = *rel-resource A* (λ*x*. *B* (*f x*)) *res1 res2*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *rel-resource-map-resource'2*:
rel-resource A B res1 (*map-resource' f res2*) = *rel-resource A* (λ*x y*. *B x* (*f y*)) *res1 res2*
⟨*proof*⟩

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

lemma *rel-resource-pos-distr*:
rel-resource A B OO rel-resource A' B' ≤ *rel-resource* (*A OO A'*) (*B OO B'*)
⟨*proof*⟩

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

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

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

definition *rel-witness-resource* :: ('*a* ⇒ '*e* ⇒ *bool*) ⇒ ('*e* ⇒ '*c* ⇒ *bool*) ⇒ ('*b* ⇒ '*d* ⇒ *bool*) ⇒ ('*a*, '*b*) *resource* × ('*c*, '*d*) *resource* ⇒ ('*e*, '*b* × '*d*) *resource* **where**
rel-witness-resource A A' B = *corec-resource* (λ(*res1*, *res2*).
map-spmf (*map-prod id Inr* ∘ *rel-witness-prod*) ∘
rel-witness-spmf (*rel-prod B* (*rel-resource* (*A OO A'*) *B*)) ∘
rel-witness-fun A A' (*run-resource res1*, *run-resource res2*))

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

run-resource (*rel-witness-resource A A′ B* (*res1*, *res2*)) =
map-spmf (*map-prod id* (*rel-witness-resource A A′ B*) ∘ *rel-witness-prod*) ∘
rel-witness-spmf (*rel-prod B* (*rel-resource* (*A OO A′*) *B*)) ∘
rel-witness-fun A A′ (*run-resource res1*, *run-resource res2*)
⟨*proof*⟩

lemma assumes *rel-resource* (*A OO A′*) *B res res′*
and *A*: *left-unique A right-total A*
and *A′*: *right-unique A′ left-total A′*
shows *rel-witness-resource1*: *rel-resource A* (λ*b* (*b′*, *c*). *b* = *b′* ∧ *B b′ c*) *res*
(*rel-witness-resource A A′ B* (*res*, *res′*)) (**is** *?thesis1*)
and *rel-witness-resource2*: *rel-resource A′* (λ(*b*, *c′*) *c*. *c* = *c′* ∧ *B b c′*) (*rel-witness-resource*
A A′ B (*res*, *res′*)) *res′* (**is** *?thesis2*)
⟨*proof*⟩

lemma *rel-resource-neg-distr*:
assumes *A*: *left-unique A right-total A*
and *A′*: *right-unique A′ left-total A′*
shows *rel-resource* (*A OO A′*) (*B OO B′*) ≤ *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* ===> *rel-spmf* (*rel-prod B* (*rel-resource A B*))) ===> *C*) ===> *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*) *I* ⇒ ('*a*, '*b*) *resource* ⇒ *bool*
for *I* **where**
lossless-resourceI: *lossless-resource I res* **if**
⋀*a*. *a* ∈ *outs-I I* ⟹ *lossless-spmf* (*run-resource res a*)
⋀*a b res'*. ⟦ *a* ∈ *outs-I I*; (*b*, *res'*) ∈ *set-spmf* (*run-resource res a*) ⟧ ⟹ *lossless-resource I res'*

lemma *lossless-resource-coinduct* [*consumes 1*, *case-names lossless-resource*, *case-conclusion lossless-resource lossless step*, *coinduct pred*: *lossless-resource*]:
assumes *X res*
and ⋀*res a*. ⟦ *X res*; *a* ∈ *outs-I I* ⟧ ⟹ *lossless-spmf* (*run-resource res a*) ∧
(∀(*b*, *res'*) ∈ *set-spmf* (*run-resource res a*). *X res'* ∨ *lossless-resource I res'*)
shows *lossless-resource I res*
⟨*proof*⟩

lemma *lossless-resourceD*:
⟦ *lossless-resource I res*; *a* ∈ *outs-I I* ⟧
⟹ *lossless-spmf* (*run-resource res a*) ∧ (∀(*x*, *res'*)∈*set-spmf* (*run-resource res a*). *lossless-resource I res'*)
⟨*proof*⟩

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

lemma *lossless-resource-mono'*:
⟦ *lossless-resource* $\mathcal{I}'$ *res*; $\mathcal{I} \leq \mathcal{I}'$ ⟧ ⟹ *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*) = (λ*a*. *map-spmf* (*map-prod id resource-of-oracle*) (*oracle s a*))

end

lemma *resource-of-oracle-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *A* ===> *rel-spmf* (*rel-prod B S*)) ===> *S* ===> *rel-resource A B*) *resource-of-oracle resource-of-oracle*
⟨*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 ∗: ⋀*s x*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *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* ⇒ ('*c*, '*d*) *resource* ⇒ ('*a* + '*c*, '*b* + '*d*) *resource* **where**
run-resource (*parallel-resource res1 res2*) =
(λ*ac*. *case ac of Inl a* ⇒ *map-spmf* (*map-prod Inl* (λ*res1'*. *parallel-resource res1' res2*)) (*run-resource res1 a*)
| *Inr c* ⇒ *map-spmf* (*map-prod Inr* (λ*res2'*. *parallel-resource res1 res2'*)) (*run-resource res2 c*))

lemma *parallel-resource-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-resource A B* ===> *rel-resource C D* ===> *rel-resource* (*rel-sum A C*) (*rel-sum B D*))
parallel-resource parallel-resource
⟨*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 $\bigwedge q\ r\ res'.\ [\![\ q \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I};\ (r,\ res') \in \mathit{set\text{-}spmf}\ (\mathit{run\text{-}resource}\ res\ q)\ ]\!] \Longrightarrow r \in \mathit{responses}\text{-}\mathcal{I}\ \mathcal{I}\ q \wedge \mathcal{I} \vdash \mathit{res}\ res'\ \surd$

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

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

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

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

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

lemma (**in** *callee-invariant-on*) *WT-resource-of-oracle*:
$I\ s \Longrightarrow \mathcal{I} \vdash \mathit{res}\ \mathit{resource\text{-}of\text{-}oracle}\ \mathit{callee}\ s\ \surd$
⟨*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 $\mathcal{I}1 \vdash \mathit{res}\ res1\ \surd$
and $\mathcal{I}2 \vdash \mathit{res}\ res2\ \surd$
shows $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash \mathit{res}\ \mathit{parallel\text{-}resource}\ res1\ res2\ \surd$
⟨*proof*⟩

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

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

interpretation *run-lossless-resource*:
callee-invariant-on run-resource $\lambda res.$ *lossless-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*

$\mathcal{I}\ \mathcal{I}'\ x = (\lambda conv.\ x \in outs\text{-}converter\ \mathcal{I}\ \mathcal{I}'\ conv)$
⟨*proof*⟩

context begin
⟨*ML*⟩

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

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

lemma *resultsp-converter-results-converter-eq* [*pred-set-conv*]:
Converter.resultsp-converter $\mathcal{I}\ \mathcal{I}'\ x = (\lambda conv.\ x \in results\text{-}converter\ \mathcal{I}\ \mathcal{I}'\ conv)$
⟨*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*
:: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('c \Rightarrow 'd \Rightarrow bool) \Rightarrow ('out \Rightarrow 'out' \Rightarrow bool) \Rightarrow ('in \Rightarrow 'in' \Rightarrow bool)$
$\Rightarrow$ ('*a*, '*c*, '*out*, '*in*) *converter* $\Rightarrow$ ('*b*, '*d*, '*out'*, '*in'*) *converter* $\Rightarrow$ *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*)

$\Longrightarrow$ *rel-converter A B C R conv1 conv2*

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

lemma *rel-converter-simps* [*simp*, *code*]:
rel-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* (λ*x*. *B* (*f x*)) (λ*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* (λ*x y*. *B x* (*f y*)) (λ*x 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' ≤ *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* ⟧ ⟹ *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* ⟧ ⟹ *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* ⟧ ⟹ *bi-unique* (*rel-converter A B C R*)
⟨*proof*⟩

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

lemma *rel-witness-converter-sel* [*simp*]:
run-converter (*rel-witness-converter A A' B C R R'* (*conv1*, *conv2*)) =
map-gpv (*map-prod id* (*rel-witness-converter A A' B C R R'*) ∘ *rel-witness-prod*) *id* ∘
rel-witness-gpv (*rel-prod B* (*rel-converter* (*A OO A'*) *B C* (*R OO R'*))) *C R R'* ∘
rel-witness-fun A A' (*run-converter conv1*, *run-converter conv2*)
⟨*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* (λ*b* (*b′*, *c*). *b* = *b′* ∧ *B b′ c*) (λ*c* (*c′*, *d*). *c* = *c′* ∧ *C c′ d*) *R conv* (*rel-witness-converter A A′ B C R R′* (*conv*, *conv′*)) (**is** *?thesis1*)
and *rel-witness-converter2*: *rel-converter A′* (λ(*b*, *c′*) *c*. *c* = *c′* ∧ *B b c′*) (λ(*c*, *d′*) *d*. *d* = *d′* ∧ *C c d′*) *R′* (*rel-witness-converter A A′ B C R R′* (*conv*, *conv′*)) *conv′* (**is** *?thesis2*)
⟨*proof*⟩

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′*) ≤ *rel-converter A B C R OO rel-converter A′ B′ C′ R′*
⟨*proof*⟩

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

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

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

inductive *pred-converter* :: *'a set* ⇒ (*'b* ⇒ *bool*) ⇒ (*'out* ⇒ *bool*) ⇒ *'in set* ⇒ (*'a*, *'b*, *'out*, *'in*) *converter* ⇒ *bool*
for *A B C R conv* **where**
pred-converter A B C R conv **if**
∀ *x*∈*results-converter* (*I-uniform A UNIV*) (*I-uniform UNIV R*) *conv*. *B x*
∀ *out*∈*outs-converter* (*I-uniform A UNIV*) (*I-uniform UNIV R*) *conv*. *C out*

lemma *pred-gpv′-mono-weak*:
pred-gpv′ A C R ≤ *pred-gpv′ A′ C′ R* **if** *A* ≤ *A′ C* ≤ *C′*
⟨*proof*⟩

lemma *Domainp-rel-converter-le*:
Domainp (*rel-converter A B C R*) ≤ *pred-converter* (*Collect* (*Domainp A*)) (*Domainp B*) (*Domainp C*) (*Collect* (*Domainp R*))
(**is** *?lhs* ≤ *?rhs*)

⟨*proof*⟩

lemma *rel-converter-Grp*:
rel-converter (*BNF-Def.Grp UNIV f*)$^{-1-1}$ (*BNF-Def.Grp B g*) (*BNF-Def.Grp C h*) (*BNF-Def.Grp UNIV k*)$^{-1-1}$ =
BNF-Def.Grp {*conv*. *results-converter* (*ℐ-uniform* (*range f*) *UNIV*) (*ℐ-uniform UNIV* (*range k*)) *conv* ⊆ *B* ∧
outs-converter (*ℐ-uniform* (*range f*) *UNIV*) (*ℐ-uniform UNIV* (*range k*)) *conv* ⊆ *C*}
(*map-converter f g h k*)
(**is** *?lhs* = *?rhs*)
including *lifting-syntax*
⟨*proof*⟩

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
(-,/ - ⊢$_C$/ - √ [100, 0, 0] 99)
for $\mathcal{I}$ $\mathcal{I}'$ **where**
WT-converterI: $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv √ **if**
⋀q. q ∈ outs-$\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}'$ ⊢g run-converter conv q √
⋀q r conv'. ⟦ q ∈ outs-$\mathcal{I}$ $\mathcal{I}$; (r, conv') ∈ results-gpv $\mathcal{I}'$ (run-converter conv q) ⟧ ⟹ r ∈ responses-$\mathcal{I}$ $\mathcal{I}$ q ∧ $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv' √

lemma WT-converter-coinduct[consumes 1, case-names WT-converter, case-conclusion WT-converter WT-gpv results-gpv, coinduct pred: WT-converter]:
assumes X conv
and ⋀conv q r conv'. ⟦ X conv; q ∈ outs-$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ $\mathcal{I}'$ ⊢g run-converter conv q √ ∧
((r, conv') ∈ results-gpv $\mathcal{I}'$ (run-converter conv q) ⟶ r ∈ responses-$\mathcal{I}$ $\mathcal{I}$ q ∧ (X conv' ∨ $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv' √))
shows $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv √
⟨proof⟩

lemma WT-converterD:
assumes $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv √ q ∈ outs-$\mathcal{I}$ $\mathcal{I}$
shows WT-converterD-WT: $\mathcal{I}'$ ⊢g run-converter conv q √
and WT-converterD-results: (r, conv') ∈ results-gpv $\mathcal{I}'$ (run-converter conv q) ⟹ r ∈ responses-$\mathcal{I}$ $\mathcal{I}$ q ∧ $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv' √
⟨proof⟩

lemma WT-converterD':
assumes $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv √ q ∈ outs-$\mathcal{I}$ $\mathcal{I}$
shows $\mathcal{I}'$ ⊢g run-converter conv q √ ∧ (∀(r, conv') ∈ results-gpv $\mathcal{I}'$ (run-converter conv q). r ∈ responses-$\mathcal{I}$ $\mathcal{I}$ q ∧ $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ conv' √)
⟨proof⟩

lemma WT-converter-bot1 [simp]: bot, $\mathcal{I}$ ⊢$_C$ conv √
⟨proof⟩

lemma WT-converter-mono:
⟦ $\mathcal{I}$1,$\mathcal{I}$2 ⊢$_C$ conv √; $\mathcal{I}$1' ≤ $\mathcal{I}$1; $\mathcal{I}$2 ≤ $\mathcal{I}$2' ⟧ ⟹ $\mathcal{I}$1',$\mathcal{I}$2' ⊢$_C$ conv √
⟨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*) $\mathcal{I}$ ⇒ ('*out*, '*in*) $\mathcal{I}$ ⇒ ('*a*, '*b*, '*out*, '*in*) *converter* ⇒ *bool*
for $\mathcal{I}$ $\mathcal{I}'$ **where**
plossless-converterI: *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* **if**
⋀*a*. *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*)
⋀*a b conv'*. ⟦ *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; (*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) ⟧ ⟹ *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*: ⋀*conv a*. ⟦ *X conv*; *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*) ∧
(∀(*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*). *X conv'* ∨ *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* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ *plossless-gpv* $\mathcal{I}'$ (*run-converter conv a*) ∧
(∀(*b*, *conv'*) ∈ *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-*$\mathcal{I}$ $\mathcal{I}1'$ ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}1$ $\mathcal{I}2$ ≤ $\mathcal{I}2'$
and *WT*: $\mathcal{I}1$, $\mathcal{I}2$ ⊢$_C$ *conv* √
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* ∧ $\mathcal{I}$,$\mathcal{I}'$ ⊢$_C$ *conv* √)
⟨*proof*⟩

interpretation *run-plossless-converter*: *raw-converter-invariant*
$\mathcal{I}$ $\mathcal{I}'$ *run-converter* λ*conv*. *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* ∧ $\mathcal{I}$,$\mathcal{I}'$ ⊢$_C$ *conv* √ **for** $\mathcal{I}$ $\mathcal{I}'$
⟨*proof*⟩

named-theorems *plossless-intro Introduction rules for probabilistic losslessness*

2.7 Operations

context
fixes *callee* :: *'s* ⇒ *'a* ⇒ (*'b* × *'s*, *'out*, *'in*) *gpv*
begin

primcorec *converter-of-callee* :: *'s* ⇒ (*'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*: ⋀*s q*. *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}'$ ⊢*g callee s q* √

and *res*: $\bigwedge s\ q\ r\ s'.\ [\![\ q \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I};\ (r,\ s') \in \mathit{results}\text{-}\mathit{gpv}\ \mathcal{I}'\ (\mathit{callee}\ s\ q)\ ]\!] \Longrightarrow r \in \mathit{responses}\text{-}\mathcal{I}\ \mathcal{I}\ q$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \mathit{converter}\text{-}\mathit{of}\text{-}\mathit{callee}\ \mathit{callee}\ s\ \surd$
⟨*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* ⇒ $('c, 'd, 'out, 'in)$ *converter* ⇒ $('a + 'c, 'b + 'd, 'out, 'in)$ *converter*
where
run-converter (*parallel-converter conv1 conv2*) = (λ*ac*. *case ac of*
Inl a ⇒ *map-gpv* (*map-prod Inl* (λ*conv1'*. *parallel-converter conv1' conv2*)) *id* (*run-converter conv1 a*)
| *Inr b* ⇒ *map-gpv* (*map-prod Inr* (λ*conv2'*. *parallel-converter conv1 conv2'*)) *id* (*run-converter conv2 b*))

lemma *parallel-callee-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C R* ===> *rel-converter A' B' C R* ===> *rel-converter* (*rel-sum A A'*) (*rel-sum B B'*) *C R*)
parallel-converter parallel-converter
⟨*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* $\surd$]\!]
⟹ *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 $\mathcal{I}1$,$\mathcal{I}2$ ⊢$_C$ *conv1* √
and $\mathcal{I}1'$,$\mathcal{I}2'$ ⊢$_C$ *conv2* √
shows $\mathcal{I}1$ ⊕$_\mathcal{I}$ $\mathcal{I}1'$,$\mathcal{I}2$ ⊕$_\mathcal{I}$ $\mathcal{I}2'$ ⊢$_C$ *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* ($\mathcal{I}1$ ⊕$_\mathcal{I}$ $\mathcal{I}2$) ($\mathcal{I}1'$ ⊕$_\mathcal{I}$ $\mathcal{I}2'$) (*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**
(*rel-converter A B C R* ===> *rel-converter A B* (*rel-sum C C'*) (*rel-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*) = (λ*a*. *map-gpv* (*map-prod id right-interface*) *id* (*right-gpv* (*run-converter conv a*)))

lemma *right-interface-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-converter A B C' R'* ===> *rel-converter A B* (*rel-sum C C'*) (*rel-sum R R'*)) *right-interface right-interface*
⟨*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*) = (λ*x*. *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*) ∧ (∀ (*y*, *s'*)∈*results-gpv* $\mathcal{I}2$ (*callee s x*). *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}1$ *x*)
shows *plossless-converter* $\mathcal{I}1$ $\mathcal{I}2$ (*converter-of-callee callee s*)
⟨*proof*⟩

context
fixes *A* :: *'a set*
and $\mathcal{I}$ :: $('c, 'd)$ $\mathcal{I}$
begin

primcorec *restrict-converter* :: $('a, 'b, 'c, 'd)$ *converter* ⇒ $('a, 'b, 'c, 'd)$ *converter*
where
run-converter (*restrict-converter cnv*) = (λ*a*. *if a* ∈ *A then*
map-gpv (*map-prod id* (λ*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* $\surd$; $\mathcal{I}' \vdash$*res res* $\surd$ ⟧ ⟹ $\mathcal{I} \vdash$*res attach conv res* $\surd$
⟨*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* $\surd$ $\mathcal{I}' \vdash$*res res* $\surd$
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'* $\surd$ ⟧ ⟹ $\mathcal{I}, \mathcal{I}'' \vdash_C$ *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* ≤ *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 ℐ*-full* 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 ⋀*x*. *consider'* (*Inl x*) ⟹ *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 $\bigwedge s\ x.$ *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 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* ⇒ (*'a*, *'b*) *resource* ⇒ (*'a*, *'b*) *resource* ⇒ *bool* (- ⊢$_R$/ - ∼/ - [*100*, *99*, *99*] *99*)
for *A* **where**
eq-resource-onI: *A* ⊢$_R$ *res* ∼ *res'* **if**
⋀*a*. *a* ∈ *A* ⟹ *rel-spmf* (*rel-prod* (=) (*eq-resource-on A*)) (*run-resource res a*) (*run-resource res' a*)

lemma *eq-resource-on-coinduct* [*consumes 1*, *case-names eq-resource-on*, *coinduct pred*: *eq-resource-on*]:
assumes *X res res'*
and ⋀*res res' a*. ⟦ *X res res'*; *a* ∈ *A* ⟧
⟹ *rel-spmf* (*rel-prod* (=) (λ*res res'*. *X res res'* ∨ *A* ⊢$_R$ *res* ∼ *res'*)) (*run-resource res a*) (*run-resource res' a*)
shows *A* ⊢$_R$ *res* ∼ *res'*
⟨*proof*⟩

lemma *eq-resource-onD*:
assumes *A* ⊢$_R$ *res* ∼ *res' a* ∈ *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* ⊢$_R$ *res* ∼ *res*
⟨*proof*⟩

lemma *eq-resource-on-reflI*: *res* = *res'* ⟹ *A* ⊢$_R$ *res* ∼ *res'*
⟨*proof*⟩

lemma *eq-resource-on-sym*: *A* ⊢$_R$ *res* ∼ *res'* **if** *A* ⊢$_R$ *res'* ∼ *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*: $\llbracket\ A' \vdash_R res \sim res';\ A \subseteq A'\ \rrbracket \Longrightarrow A \vdash_R res \sim res'$
⟨*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*: $(S ===> eq\text{-}on\ A ===> rel\text{-}spmf\ (rel\text{-}prod\ (=)\ S))\ r1\ r2$
and S: $S\ s1\ s2$
shows $A \vdash_R resource\text{-}of\text{-}oracle\ r1\ s1 \sim resource\text{-}of\text{-}oracle\ r2\ s2$
⟨*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* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('out, 'in)\ \mathcal{I} \Rightarrow ('c \Rightarrow 'd \Rightarrow bool) \Rightarrow ('a, 'out, 'in \Rightarrow 'c)\ generat \Rightarrow ('b, 'out, 'in \Rightarrow 'd)\ generat \Rightarrow bool$
for $A\ \mathcal{I}\ D$ **where**
Pure: *eq-*$\mathcal{I}$*-generat* $A\ \mathcal{I}\ D$ (*Pure* x) (*Pure* y) **if** $A\ x\ y$
| *IO*: *eq-*$\mathcal{I}$*-generat* $A\ \mathcal{I}\ D$ (*IO out* c) (*IO out* c') **if** $out \in outs\text{-}\mathcal{I}\ \mathcal{I} \bigwedge input.\ input \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out \Longrightarrow D\ (c\ input)\ (c'\ input)$

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

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

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

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

lemma *eq-$\mathcal{I}$-generat-mono'*:
$[\![$ *eq-$\mathcal{I}$-generat A $\mathcal{I}$ D x y*; $\bigwedge x\ y.$ *A x y* $\Longrightarrow$ *A' x y*; $\bigwedge x\ y.$ *D x y* $\Longrightarrow$ *D' x y*; $\mathcal{I} \leq \mathcal{I}'$ $]\!]$
$\Longrightarrow$ *eq-$\mathcal{I}$-generat A' $\mathcal{I}'$ D' x y*
⟨*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* ((∘) *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* ((∘) *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-ℐ-generat-map* [*simp*] =
eq-ℐ-generat-map1[*abs-def*] *eq-ℐ-generat-map2*
eq-ℐ-generat-map1[**where** *g*=*id*, *unfolded fun.map-id0*, *abs-def*] *eq-ℐ-generat-map2*[**where** *g*=*id*, *unfolded fun.map-id0*]

lemma *eq-ℐ-generat-into-rel-generat*:
eq-ℐ-generat A ℐ-full D generat generat' ⟹ *rel-generat A* (=) (*rel-fun* (=) *D*) *generat generat'*
⟨*proof*⟩

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

lemma *eq-ℐ-gpv-coinduct* [*consumes 1*, *case-names eq-ℐ-gpv*, *coinduct pred*: *eq-ℐ-gpv*]:
assumes *X gpv gpv'*
and ⋀*gpv gpv'*. *X gpv gpv'*
⟹ *rel-spmf* (*eq-ℐ-generat A ℐ* (λ*gpv gpv'*. *X gpv gpv'* ∨ *eq-ℐ-gpv A ℐ gpv gpv'*)) (*the-gpv gpv*) (*the-gpv gpv'*)
shows *eq-ℐ-gpv A ℐ gpv gpv'*
⟨*proof*⟩

lemma *eq-ℐ-gpvD*:
eq-ℐ-gpv A ℐ gpv gpv' ⟹ *rel-spmf* (*eq-ℐ-generat A ℐ* (*eq-ℐ-gpv A ℐ*)) (*the-gpv gpv*) (*the-gpv gpv'*)
⟨*proof*⟩

lemma *eq-ℐ-gpv-Done* [*intro*!]: *A x y* ⟹ *eq-ℐ-gpv A ℐ* (*Done x*) (*Done y*)
⟨*proof*⟩

lemma *eq-ℐ-gpv-Done-iff* [*simp*]: *eq-ℐ-gpv A ℐ* (*Done x*) (*Done y*) ⟷ *A x y*
⟨*proof*⟩

lemma *eq-ℐ-gpv-Pause*:
⟦ *out* ∈ *outs-ℐ ℐ*; ⋀*input*. *input* ∈ *responses-ℐ ℐ out* ⟹ *eq-ℐ-gpv A ℐ* (*rpv input*) (*rpv' input*) ⟧
⟹ *eq-ℐ-gpv A ℐ* (*Pause out rpv*) (*Pause out rpv'*)
⟨*proof*⟩

lemma *eq-ℐ-gpv-mono*: *eq-ℐ-gpv A ℐ* ≤ *eq-ℐ-gpv A' ℐ'* **if** *A*: *A* ≤ *A' ℐ* ≤ *ℐ'*
⟨*proof*⟩

lemma *eq-ℐ-gpv-mono'*:
⟦ *eq-ℐ-gpv A ℐ gpv gpv'*; ⋀*x y*. *A x y* ⟹ *A' x y*; *ℐ* ≤ *ℐ'* ⟧ ⟹ *eq-ℐ-gpv A' ℐ' gpv gpv'*
⟨*proof*⟩

lemma *eq-ℐ-gpv-mono''* [*mono*]:
  eq-ℐ-gpv A ℐ gpv gpv' ⟶ *eq-ℐ-gpv A' ℐ gpv gpv'* **if** ⋀*x y*. *A x y* ⟶ *A' x y*
  ⟨*proof*⟩

lemma *eq-ℐ-gpv-conversep*: *eq-ℐ-gpv* A^{-1-1} *ℐ* = (*eq-ℐ-gpv A ℐ*)$^{-1-1}$
⟨*proof*⟩

lemma *eq-ℐ-gpv-reflI*:
  ⟦ ⋀*x*. *x* ∈ *results-gpv ℐ gpv* ⟹ *A x x*; *ℐ* ⊢*g gpv* √ ⟧ ⟹ *eq-ℐ-gpv A ℐ gpv gpv*
  ⟨*proof*⟩

lemma *eq-ℐ-gpv-into-rel-gpv*: *eq-ℐ-gpv A ℐ-full gpv gpv'* ⟹ *rel-gpv A* (=) *gpv gpv'*
  ⟨*proof*⟩

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

lemma *eq-ℐ-gpv-map-gpv1*: *eq-ℐ-gpv A ℐ* (*map-gpv f id gpv*) *gpv'* ⟷ *eq-ℐ-gpv* (λ*x*. *A* (*f x*)) *ℐ gpv gpv'* (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *eq-ℐ-gpv-map-gpv2*: *eq-ℐ-gpv A ℐ gpv* (*map-gpv f id gpv'*) = *eq-ℐ-gpv* (λ*x y*. *A x* (*f y*)) *ℐ gpv gpv'*
  ⟨*proof*⟩

lemmas *eq-ℐ-gpv-map-gpv* [*simp*] = *eq-ℐ-gpv-map-gpv1*[*abs-def*] *eq-ℐ-gpv-map-gpv2*

lemma (**in** *callee-invariant-on*) *eq-ℐ-exec-gpv*:
  ⟦ *eq-ℐ-gpv A ℐ gpv gpv'*; *I s* ⟧ ⟹ *rel-spmf* (*rel-prod A* (*eq-onp I*)) (*exec-gpv callee gpv s*) (*exec-gpv callee gpv' s*)
⟨*proof*⟩

lemma *eq-ℐ-gpv-coinduct-bind* [*consumes 1*, *case-names eq-ℐ-gpv*]:
  fixes *gpv* :: ('*a*, '*out*, '*in*) *gpv* **and** *gpv'* :: ('*a'*, '*out*, '*in*) *gpv*
  assumes *X*: *X gpv gpv'*
    and *step*: ⋀*gpv gpv'*. *X gpv gpv'*
      ⟹ *rel-spmf* (*eq-ℐ-generat A ℐ* (λ*gpv gpv'*. *X gpv gpv'* ∨ *eq-ℐ-gpv A ℐ gpv gpv'* ∨
      (∃ *gpv'' gpv'''* (*B* :: '*b* ⇒ '*b'* ⇒ *bool*) *f g*. *gpv* = *bind-gpv gpv'' f* ∧ *gpv'* = *bind-gpv gpv''' g* ∧ *eq-ℐ-gpv B ℐ gpv'' gpv'''* ∧ (*rel-fun B X*) *f g*))) (*the-gpv gpv*) (*the-gpv gpv'*)
  shows *eq-ℐ-gpv A ℐ gpv gpv'*
⟨*proof*⟩

context
  fixes *S* :: '*s1* ⇒ '*s2* ⇒ *bool*
    and *callee1* :: '*s1* ⇒ '*out* ⇒ ('*in* × '*s1*, '*out'*, '*in'*) *gpv*

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

lemma *eq-*$\mathcal{I}$*-gpv-inline1*:
includes *lifting-syntax*
assumes *S s1 s2 eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ *gpv1 gpv2*
shows *rel-spmf* (*rel-sum* (*rel-prod A S*)
(λ(*q*, *rpv1*, *rpv2*) (*q'*, *rpv1'*, *rpv2'*). *q* = *q'* ∧ *q'* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}'$ ∧ (∃ *q''* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$.
(∀ *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *q'*. *eq-*$\mathcal{I}$*-gpv* (*rel-prod* (*eq-onp* (λ*r'*. *r'* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q''*)) *S*) $\mathcal{I}'$ (*rpv1 r*) (*rpv1' r*)) ∧
(∀ *r'* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q''*. *eq-*$\mathcal{I}$*-gpv A* $\mathcal{I}$ (*rpv2 r'*) (*rpv2' r'*)))))
(*inline1 callee1 gpv1 s1*) (*inline1 callee2 gpv2 s2*)
⟨*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* ($\mathcal{I}$ ⊕$_\mathcal{I}$ $\mathcal{I}'$) (*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* ($\mathcal{I}$ ⊕$_\mathcal{I}$ $\mathcal{I}'$) (*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*
(-,- ⊢$_C$/ - ∼/ - [*100*, *0*, *99*, *99*] *99*)
for $\mathcal{I}$ $\mathcal{I}'$ **where**
eq-$\mathcal{I}$*-converterI*: $\mathcal{I}$, $\mathcal{I}'$ ⊢$_C$ *conv* ∼ *conv'* **if**

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

lemma *eq-$\mathcal{I}$-converter-coinduct* [*consumes 1*, *case-names eq-$\mathcal{I}$-converter*, *coinduct pred*: *eq-$\mathcal{I}$-converter*]:
assumes $X\ \mathit{conv}\ \mathit{conv}'$
and $\bigwedge \mathit{conv}\ \mathit{conv}'\ q.\ [\![\ X\ \mathit{conv}\ \mathit{conv}';\ q \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I}\ ]\!]$
$\Longrightarrow \mathit{eq}\text{-}\mathcal{I}\text{-}\mathit{gpv}\ (\mathit{rel\text{-}prod}\ (\mathit{eq\text{-}onp}\ (\lambda r.\ r \in \mathit{responses}\text{-}\mathcal{I}\ \mathcal{I}\ q))\ (\lambda \mathit{conv}\ \mathit{conv}'.\ X\ \mathit{conv}\ \mathit{conv}' \vee \mathcal{I}, \mathcal{I}' \vdash_C \mathit{conv} \sim \mathit{conv}'))\ \mathcal{I}'$
$(\mathit{run\text{-}converter}\ \mathit{conv}\ q)\ (\mathit{run\text{-}converter}\ \mathit{conv}'\ q)$
shows $\mathcal{I}, \mathcal{I}' \vdash_C \mathit{conv} \sim \mathit{conv}'$
⟨*proof*⟩

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

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

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

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

lemma *eq-$\mathcal{I}$-converter-mono*:
assumes $*$: $\mathcal{I}1, \mathcal{I}2 \vdash_C \mathit{conv} \sim \mathit{conv}'$
and *le*: $\mathcal{I}1' \leq \mathcal{I}1\ \mathcal{I}2 \leq \mathcal{I}2'$
shows $\mathcal{I}1', \mathcal{I}2' \vdash_C \mathit{conv} \sim \mathit{conv}'$
⟨*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 $\mathcal{I}' \vdash \mathit{res}\ \mathit{res}\ \surd\ \mathcal{I}\text{-}\mathit{uniform}\ A\ \mathit{UNIV}, \mathcal{I}' \vdash_C \mathit{conv} \sim \mathit{conv}'$
shows $A \vdash_R \mathit{attach}\ \mathit{conv}\ \mathit{res} \sim \mathit{attach}\ \mathit{conv}'\ \mathit{res}$
⟨*proof*⟩

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

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

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

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

lemma *parallel-converter2-eq-*$\mathcal{I}$*-cong*:
⟦ $\mathcal{I}1, \mathcal{I}1' \vdash_C conv1 \sim conv1'$; $\mathcal{I}2, \mathcal{I}2' \vdash_C conv2 \sim conv2'$ ⟧
⟹ $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \vdash_C$ *parallel-converter2 conv1 conv2* ∼ *parallel-converter2 conv1' conv2'*
⟨*proof*⟩

lemma *id-converter-eq-self*: $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *id-converter* ∼ *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* ∼ *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* ⇒ '*s spmf* ⇒ ('*a* × '*b*) *list* ⇒ '*a* ⇒ '*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* ⇒ ('*a*, '*b*, '*s2*) *callee* ⇒ '*a set* ⇒ '*s1 spmf* ⇒ '*s2 spmf* ⇒ *bool* **where**
trace-callee-eq callee1 callee2 A p q ⟷
(∀ *xs*. *set xs* ⊆ *A* × *UNIV* ⟶ (∀ *x* ∈ *A*. *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*))

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

lemma *trace-callee-eqI*:
assumes ⋀*xs x*. ⟦ *set xs* ⊆ *A* × *UNIV*; *x* ∈ *A* ⟧
⟹ *trace-callee callee1 p xs x* = *trace-callee callee2 q xs x*
shows *trace-callee-eq callee1 callee2 A p q*
⟨*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*) ⟧
⟹ *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 $a \in A$
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 $a \in A$
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 ⋀*p q a*. ⟦ *S p q*; $a \in A$ ⟧ ⟹
bind-spmf p (λ*s*. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ*s*. *map-spmf fst* (*callee2 s a*))
and ⋀*p q a s b s'*. ⟦ *S p q*; $a \in A$; $s \in$ *set-spmf q*; $(b, s') \in$ *set-spmf* (*callee2 s a*) ⟧
⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s a*)) *b*)
⟨*proof*⟩

lemma *set-spmf-cond-spmf-fst*: *set-spmf* (*cond-spmf-fst p a*) = *snd* ‘ (*set-spmf p* ∩ {*a*} × *UNIV*)
⟨*proof*⟩

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* √
and *out*: *outs-gpv* $\mathcal{I}$ *gpv* ⊆ *A*
and *pq*: *lossless-spmf p lossless-spmf q*
and *I1*: ∀ *x*∈*set-spmf p*. *I1 x*
and *I2*: ∀ *y*∈*set-spmf q*. *I2 y*
shows *bind-spmf p* (*run-gpv callee1 gpv*) = *bind-spmf q* (*run-gpv callee2 gpv*)
⟨*proof*⟩

lemma *trace-callee-eq'-run-gpv*:
fixes *callee1* :: ('*a*, '*b*, '*s1*) *callee* **and** *callee2* :: ('*a*, '*b*, '*s2*) *callee*
assumes *trace-eq*: $A \vdash_C$ *callee1*(*s1*) ≈ *callee2*(*s2*)
and *inv1*: *callee-invariant-on callee1 I1* $\mathcal{I}$
and *inv1*: *callee-invariant-on callee2 I2* $\mathcal{I}$
and *WT*: $\mathcal{I} \vdash g$ *gpv* √

and *out*: *outs-gpv* $\mathcal{I}$ *gpv* $\subseteq$ *A*
and *I1*: *I1 s1*
and *I2*: *I2 s2*
shows *run-gpv callee1 gpv s1* = *run-gpv callee2 gpv s2*
⟨*proof*⟩

abbreviation *trace-eq* :: *'a set* ⇒ (*'a*, *'b*) *resource spmf* ⇒ (*'a*, *'b*) *resource spmf* ⇒ *bool* **where**
trace-eq ≡ *trace-callee-eq run-resource run-resource*

abbreviation *trace-eq'* :: *'a set* ⇒ (*'a*, *'b*) *resource* ⇒ (*'a*, *'b*) *resource* ⇒ *bool* ((-) ⊢$_R$/ (-)/ ≈ (-) [*90*, *90*, *90*] *91*) **where**
A ⊢$_R$ *res* ≈ *res'* ≡ *trace-eq A* (*return-spmf res*) (*return-spmf res'*)

lemma *trace-callee-resource-of-oracle2*:
trace-callee run-resource (*map-spmf* (*resource-of-oracle callee*) *p*) *xs x* =
trace-callee callee p xs x
⟨*proof*⟩

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 ⊢$_R$ *resource-of-oracle callee1 s1* ≈ *resource-of-oracle callee2 s2* =
A ⊢$_C$ *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* $\approx$ *res2*
and *WT*: $\mathcal{I} \vdash g$ *d* $\surd$
and *out*: *outs-gpv* $\mathcal{I}$ *d* $\subseteq A$
and *WT1*: $\mathcal{I} \vdash$*res res1* $\surd$
and *WT2*: $\mathcal{I} \vdash$*res res2* $\surd$
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** $\parallel$ *50*)
no-notation *plus-oracle* (**infix** $\oplus_O$ *500*)

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

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* ≡ $lassocr_C \odot (1_C \mid_= swap_C) \odot rassocl_C$
definition *swap-lassocr* **where** *swap-lassocr* ≡ $rassocl_C \odot (swap_C \mid_= 1_C) \odot lassocr_C$

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

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*]: $\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3), \mathcal{I}2 \oplus_\mathcal{I} (\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3) \vdash_C swap\text{-}lassocr \surd$
⟨*proof*⟩

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

lemma *WT-parallel-wiring* [*WT-intro*]:
$(\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2) \oplus_\mathcal{I} (\mathcal{I}3 \oplus_\mathcal{I} \mathcal{I}4), (\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3) \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}4) \vdash_C parallel\text{-}wiring \surd$
⟨*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* $|_=$ *conv2*) $|_=$ *conv3*) =
map-converter rsuml lsumr id id (*conv1* $|_=$ *conv2* $|_=$ *conv3*)
⟨*proof*⟩

lemma *comp-lassocr*$_C$: ((*conv1* $|_=$ *conv2*) $|_=$ *conv3*) $\odot$ *lassocr*$_C$ = *lassocr*$_C$ $\odot$ (*conv1* $|_=$ *conv2* $|_=$ *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* $|_=$ (*conv2* $|_=$ *conv3*)) =
map-converter lsumr rsuml id id ((*conv1* $|_=$ *conv2*) $|_=$ *conv3*)
⟨*proof*⟩

lemma *comp-rassocl*$_C$:
(*conv1* $|_=$ *conv2* $|_=$ *conv3*) $\odot$ *rassocl*$_C$ = *rassocl*$_C$ $\odot$ ((*conv1* $|_=$ *conv2*) $|_=$ *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 |$_=$ conv2) =
map-converter swap-sum swap-sum id id (conv2 |$_=$ conv1)
⟨*proof*⟩

lemma *comp-swap$_C$*: *(conv1 |$_=$ conv2) ⊙ swap$_C$ = swap$_C$ ⊙ (conv2 |$_=$ conv1)*
⟨*proof*⟩

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

lemma *comp-swap-lassocr*: *(conv1 |$_=$ conv2 |$_=$ conv3) ⊙ swap-lassocr = swap-lassocr ⊙ (conv2 |$_=$ conv1 |$_=$ conv3)*
⟨*proof*⟩

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

lemma *comp-parallel-wiring*:
((C1 |$_=$ C2) |$_=$ (C3 |$_=$ C4)) ⊙ parallel-wiring = parallel-wiring ⊙ ((C1 |$_=$ C3) |$_=$ (C2 |$_=$ C4))
⟨*proof*⟩

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

lemma *attach-converter-of-resource-conv-parallel-resource*:
converter-of-resource res |$_\propto$ 1$_C$ ▷ res' = res ∥ res'
⟨*proof*⟩

lemma *attach-converter-of-resource-conv-parallel-resource2*:
1$_C$ |$_\propto$ converter-of-resource res ▷ res' = res' ∥ res
⟨*proof*⟩

lemma *plossless-parallel-wiring* [*plossless-intro*]:
plossless-converter ((ℐ1 ⊕$_ℐ$ ℐ2) ⊕$_ℐ$ (ℐ3 ⊕$_ℐ$ ℐ4)) ((ℐ1 ⊕$_ℐ$ ℐ3) ⊕$_ℐ$ (ℐ2 ⊕$_ℐ$ ℐ4)) parallel-wiring
⟨*proof*⟩

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

run-converter lassocr$_C$ x = Pause (rsuml x) (λx. Done (lsumr x, lassocr$_C$))
⟨*proof*⟩

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

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

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

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

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

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

lemma *bound-lassocr$_C$* [*interaction-bound*]: *interaction-any-bounded-converter 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* $\triangleright$ *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* ⇒ ('*s* ⇒ '*c* ⇒ ('*d* × '*s*, '*e*, '*f*) *gpv*) ⇒ ('*s* ⇒ '*a* ⇒ ('*b* × '*s*, '*e*, '*f*) *gpv*)
where *attach-wiring* = (λ(*f*, *g*). *map-fun id* (*map-fun f* (*map-gpv* (*map-prod g id*) *id*)))

lemma *attach-wiring-simps*: *attach-wiring* (*f*, *g*) = *map-fun id* (*map-fun f* (*map-gpv* (*map-prod g id*) *id*))
⟨*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* ⇒ ('*c*, '*d*, '*e*, '*f*) *wiring* ⇒ ('*a*, '*b*, '*e*, '*f*) *wiring* (**infixl** $\circ_w$ *55*)
where *comp-wiring* = (λ(*f*, *g*) (*f'*, *g'*). (*f'* ∘ *f*, *g* ∘ *g'*))

lemma *comp-wiring-simps*: *comp-wiring* (*f*, *g*) (*f'*, *g'*) = (*f'* ∘ *f*, *g* ∘ *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* ⇒ ('*a'*, '*b'*, '*c'*, '*d'*) *wiring*
⇒ ('*a* + '*a'*, '*b* + '*b'*, '*c* + '*c'*, '*d* + '*d'*) *wiring* (**infix** $|_w$ *501*) **where**
parallel2-wiring = (λ(*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* $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2)$ $(\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2')$ $(conv1 \mid_{=} conv2)$ $(fg \mid_w fg')$
⟨*proof*⟩

lemma *apply-parallel2* [*simp*]:
apply-wiring $(fg \mid_w fg')$ $(res1 \oplus_O res2) = (apply\text{-}wiring\ fg\ res1 \oplus_O apply\text{-}wiring\ fg'\ res2)$
⟨*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 $apply\text{-}lassocr_w$ [*simp*]: *apply-wiring* $lassocr_w$ $(res1 \oplus_O res2 \oplus_O res3) = (res1 \oplus_O res2) \oplus_O res3$
⟨*proof*⟩

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

lemma $apply\text{-}swap_w$ [*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* ⇒ ('*e*, '*f*) $\mathcal{I}$
and *bound* :: *security* ⇒ *enat*
and *lossless* :: *bool*
assumes *WT-real* [*WT-intro*]: ⋀η. $\mathcal{I}$*-real* η ⊕$_\mathcal{I}$ $\mathcal{I}$*-common* η ⊢*res real-resource* η √
and *WT-ideal* [*WT-intro*]: ⋀η. $\mathcal{I}$*-ideal* η ⊕$_\mathcal{I}$ $\mathcal{I}$*-common* η ⊢*res ideal-resource* η √
and *WT-sim* [*WT-intro*]: ⋀η. $\mathcal{I}$*-real* η, $\mathcal{I}$*-ideal* η ⊢$_C$ *sim* η √
and *adv*: ⋀$\mathcal{A}$ :: *security* ⇒ ('*a* + '*e*, '*b* + '*f*) *distinguisher*.
⟦ ⋀η. $\mathcal{I}$*-real* η ⊕$_\mathcal{I}$ $\mathcal{I}$*-common* η ⊢*g* $\mathcal{A}$ η √;
⋀η. *interaction-bounded-by* (λ-. *True*) ($\mathcal{A}$ η) (*bound* η);
⋀η. *lossless* ⟹ *plossless-gpv* ($\mathcal{I}$*-real* η ⊕$_\mathcal{I}$ $\mathcal{I}$*-common* η) ($\mathcal{A}$ η) ⟧
⟹ *negligible* (λη. *advantage* ($\mathcal{A}$ η) (*sim* η |$_=$ 1$_C$ ▷ *ideal-resource* η) (*real-resource* η))

locale *constructive-security* =
constructive-security-aux real-resource ideal-resource sim $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common bound lossless*
for *real-resource* :: *security* ⇒ ('*a* + '*e*, '*b* + '*f*) *resource*
and *ideal-resource* :: *security* ⇒ ('*c* + '*e*, '*d* + '*f*) *resource*
and *sim* :: *security* ⇒ ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$*-real* :: *security* ⇒ ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$*-ideal* :: *security* ⇒ ('*c*, '*d*) $\mathcal{I}$
and $\mathcal{I}$*-common* :: *security* ⇒ ('*e*, '*f*) $\mathcal{I}$
and *bound* :: *security* ⇒ *enat*
and *lossless* :: *bool*
and *w* :: *security* ⇒ ('*c*, '*d*, '*a*, '*b*) *wiring*
+
assumes *correct*: ∃ *cnv*. ∀ $\mathcal{D}$ :: *security* ⇒ ('*c* + '*e*, '*d* + '*f*) *distinguisher*.
(∀η. $\mathcal{I}$*-ideal* η ⊕$_\mathcal{I}$ $\mathcal{I}$*-common* η ⊢*g* $\mathcal{D}$ η √)
⟶ (∀η. *interaction-bounded-by* (λ-. *True*) ($\mathcal{D}$ η) (*bound* η))
⟶ (∀η. *lossless* ⟶ *plossless-gpv* ($\mathcal{I}$*-ideal* η ⊕$_\mathcal{I}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η))
⟶ (∀η. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η)) ∧

negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))

locale *constructive-security2* =
constructive-security-aux real-resource ideal-resource sim $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common bound lossless*
for *real-resource* :: *security* $\Rightarrow$ ('*a* + '*e*, '*b* + '*f*) *resource*
and *ideal-resource* :: *security* $\Rightarrow$ ('*c* + '*e*, '*d* + '*f*) *resource*
and *sim* :: *security* $\Rightarrow$ ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$*-real* :: *security* $\Rightarrow$ ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$*-ideal* :: *security* $\Rightarrow$ ('*c*, '*d*) $\mathcal{I}$
and $\mathcal{I}$*-common* :: *security* $\Rightarrow$ ('*e*, '*f*) $\mathcal{I}$
and *bound* :: *security* $\Rightarrow$ *enat*
and *lossless* :: *bool*
and *w* :: *security* $\Rightarrow$ ('*c*, '*d*, '*a*, '*b*) *wiring*
+
assumes *sim*: $\exists$ *cnv*. $\forall\eta$. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η) $\wedge$ *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-ideal* η) (*cnv* η $\odot$ *sim* η) (*id*, *id*)
begin

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

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* η $\odot$ *sim-inner* η) $\mathcal{I}$*-real* $\mathcal{I}$*-inner* $\mathcal{I}$*-common bound-outer* (*lossless-outer* $\vee$ *lossless-inner*) ($\lambda\eta$. *w1* η $\circ_w$ *w2* η)
⟨*proof*⟩

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

theorem *constructive-security-trivial*:
fixes *res*
assumes [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$ η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\eta \vdash$*res res* η $\surd$
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* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η) (*ideal2* η)
shows *constructive-security* ($\lambda\eta$. *parallel-wiring* $\rhd$ *real1* η $\|$ *real2* η) ($\lambda\eta$. *parallel-wiring* $\rhd$ *ideal1* η $\|$ *ideal2* η) ($\lambda\eta$. *sim1* η $|_=$ *sim2* η)
($\lambda\eta$. $\mathcal{I}$*-real1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2* η) ($\lambda\eta$. $\mathcal{I}$*-inner1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-inner2* η) ($\lambda\eta$. $\mathcal{I}$*-common1* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2* η)
($\lambda\eta$. *min* (*bound1* η) (*bound2* η)) (*lossless1* $\vee$ *lossless2*) ($\lambda\eta$. *w1* η $|_w$ *w2* η)
⟨*proof*⟩

theorem (**in** *constructive-security*) *parallel-realisation1*:
assumes *WT-res*: $\bigwedge\eta$. $\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common'* $\eta \vdash$*res res* η $\surd$
and *lossless-res*: $\bigwedge\eta$. *lossless* $\Longrightarrow$ *lossless-resource* ($\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common'* η) (*res* η)
shows *constructive-security* ($\lambda\eta$. *parallel-wiring* $\rhd$ *res* η $\|$ *real-resource* η)
($\lambda\eta$. *parallel-wiring* $\rhd$ (*res* η $\|$ *ideal-resource* η)) ($\lambda\eta$. *parallel-converter2 id-converter* (*sim* η))
($\lambda\eta$. $\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real* η) ($\lambda\eta$. $\mathcal{I}$*-res* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-ideal* η) ($\lambda\eta$. $\mathcal{I}$*-common'* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-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* $\eta \parallel$ *real-resource* η)
($\lambda\eta$. *parallel-resource1-wiring* $\rhd$ *res* $\eta \parallel$ *ideal-resource* η) *sim*
$\mathcal{I}$*-real* $\mathcal{I}$*-ideal* ($\lambda\eta$. $\mathcal{I}$*-res* $\eta \oplus_{\mathcal{I}} \mathcal{I}$*-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* ⊕$_O$ *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* ≡ *Forwar-*
dOrEdit 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 ⊕O send-oracle ⊕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* $\equiv (1_C \mid_= lassocr_C) \odot$ *swap-lassocr* $\odot (1_C \mid_= (1_C \mid_= swap\text{-}lassocr) \odot$ *swap-lassocr*$) \odot rassocl_C$

definition *res* $= (1_C \mid_= enc \mid_= dec) \rhd (1_C \mid_=$ *parallel-wiring*$) \rhd$ *parallel-resource1-wiring* $\rhd$ (*KEY.res* $\parallel$ *AUTH.res*)

lemma *res-alt-def*: *res* $=$ ((*1*$_C$ $\mid_=$ *enc* $\mid_=$ *dec*) $\odot$ (*1*$_C$ $\mid_=$ *parallel-wiring*)) $\rhd$ *parallel-resource1-wiring* $\rhd$ (*KEY.res* $\parallel$ *AUTH.res*)
⟨*proof*⟩

end

8.5 Message authentication converter

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

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

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

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

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

$\odot$ *swap-lassocr*) $\odot$ $rassocl_C$

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

end

lemma *interface-wiring*:
(*cnv-advr* $|_=$ *cnv-send* $|_=$ *cnv-recv*) $\triangleright$ (1_C $|_=$ *parallel-wiring*) $\triangleright$ *parallel-resource1-wiring* $\triangleright$
(*RES* (*res2-send* $\oplus_O$ *res2-recv*) *res2-s* $\parallel$ *RES* (*res1-advr* $\oplus_O$ *res1-send* $\oplus_O$ *res1-recv*) *res1-s*)
=
cnv-advr $|_=$ *cnv-send* $|_=$ *cnv-recv* $\triangleright$
RES (†*res1-advr* $\oplus_O$ (*res2-send*† $\oplus_O$ †*res1-send*) $\oplus_O$ *res2-recv*† $\oplus_O$ †*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* $\Rightarrow$ *bool list spmf* **where**
key η $\equiv$ *spmf-of-set* (*nlists UNIV* η)

definition *enc* :: *security* $\Rightarrow$ *bool list* $\Rightarrow$ *bool list* $\Rightarrow$ *bool list spmf* **where**
enc η *k m* $\equiv$ *return-spmf* (*k* [$\oplus$] *m*)

definition *dec* :: *security* $\Rightarrow$ *bool list* $\Rightarrow$ *bool list* $\Rightarrow$ *bool list option* **where**
dec η *k c* $\equiv$ *Some* (*k* [$\oplus$] *c*)

definition *sim* :: *'b list option* $\Rightarrow$ *'a* $\Rightarrow$ (*'b list option* $\times$ *'b list option*, *'a*, *nat option*) *gpv* **where**
sim c q $\equiv$ (*do* {
lo $\leftarrow$ *Pause q Done*;
(*case lo of*
Some n $\Rightarrow$ *if c* = *None*
then do {
x $\leftarrow$ *lift-spmf* (*spmf-of-set* (*nlists UNIV n*));

```
        Done (Some x, Some x)}
      else Done (c, c)
  | None   ⇒ Done (None, c))})

context
  fixes η :: security
begin

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

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

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

private inductive S :: (bool list option × unit × bool list cstate) spmf ⇒
  (bool list option × bool list cstate) spmf ⇒ bool where
  S (return-spmf (None, (), Void))
    (return-spmf (None, Void))
| S (return-spmf (None, (), Store plain))
    (map-spmf (λkey. (Some key, Store (key [⊕] plain))) (spmf-of-set (nlists UNIV
η)))
if length plain = id' η
| S (return-spmf (None, (), Collect plain))
     (map-spmf (λkey. (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 (λx. (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*) ⊢$_R$
RES (*callee-sec-channel sim* ⊕$_O$ ††*channel.send-oracle* ⊕$_O$ ††*channel.recv-oracle*)
(*None* :: *bool list option*, (), *Void*)
≈
RES (†*auth-channel.auth-oracle* ⊕$_O$ *key-channel-send* ⊕$_O$ *key-channel-recv*)
(*None* :: *bool list option*, *Void*)
(**is** *?A* ⊢$_R$ *RES* (*?L1* ⊕$_O$ *?L2* ⊕$_O$ *?L3*) *?SL* ≈ *RES* (*?R1* ⊕$_O$ *?R2* ⊕$_O$ *?R3*)
?SR)
⟨*proof*⟩

lemma *real-resource-wiring*:
shows *cipher.res* (*key* η) (*enc* η) (*dec* η)
= *RES* (†*auth-channel.auth-oracle* ⊕$_O$ *key-channel-send* ⊕$_O$ *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* ⊕$_O$ ††*channel.send-oracle* ⊕$_O$ ††*channel.recv-oracle*
) (*s*, (), *Void*) (**is** *?L1* ▷ - = *?R*)
⟨*proof*⟩

end

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

lemma *eq-ℐ-gpv-Done2*:
eq-ℐ-gpv A ℐ gpv (*Done x*) ⟷ *lossless-spmf* (*the-gpv gpv*) ∧ (∀ *a*∈*set-spmf*
(*the-gpv gpv*). *eq-ℐ-generat A ℐ* (*eq-ℐ-gpv A ℐ*) *a* (*Pure x*))
⟨*proof*⟩

context begin
interpretation *CIPHER*: *cipher key* η *enc* η *dec* η **for** η ⟨*proof*⟩
interpretation *S-CHAN*: *sec-channel* ⟨*proof*⟩

lemma *one-time-pad*:
defines *ℐ-real* ≡ λη. *ℐ-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η))
and *ℐ-ideal* ≡ λη. *ℐ-uniform UNIV* {*None*, *Some* η}
and *ℐ-common* ≡ λη. *ℐ-uniform* (*nlists UNIV* η) *UNIV* ⊕$_ℐ$ *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* ∈ *vld* η ∧ *m* ∈ *vld* η
| *valid-mac-query* η - = *True*

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

```
    Done (None, Inr ()) }
| sim (Inl (Some (a, m)))  Look            = do {
    lo ← Pause Look Done;
    (case lo of
      Some m ⇒ Done (Some (a, m), Inl (Some (a, m)))
    | None   ⇒ Done (None, Inl (Some (a, m))))}
| sim (Inl None)          Look            = do {
    lo ← Pause Look Done;
    (case lo of
      Some m ⇒ do {
        a ← lift-spmf (spmf-of-set (nlists UNIV (length m)));
        Done (Some (a, m), Inl (Some (a, m)))}
    | None   ⇒ Done (None, Inl None))}


context
  fixes η :: security
begin

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

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

private definition rorc-channel-recv-f :: ((bool list ⇒ bool list option) × (bool list
× bool list) cstate, unit, bool list option) oracle′ where
  rorc-channel-recv-f s q ≡ do {
    (am, (as, ams)) ←  †channel.recv-oracle s ();
    (case am of
      None ⇒ return-spmf (None, (as, ams))
    | Some (a, m) ⇒ (case as m of
```

```
      None ⇒ do {
        a'' :: bool list ← spmf-of-set (nlists UNIV η − {a});
        a' ← spmf-of-set (nlists UNIV η);
        (if a' = a
          then return-spmf (None, as(m := Some a''), ams)
          else return-spmf (None, as(m := Some a'), ams)) }
    | Some a' ⇒ return-spmf (if a' = a then Some m else None, as, ams)))}

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

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

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

private fun lazy-channel-recv-f :: (bool list  cstate × (bool list × bool list) option
× (bool list ⇒ bool list option), unit, bool list option) oracle' where
  lazy-channel-recv-f (Collect m, None, as)    () = return-spmf (Some m, (Fail,
None, as))
| lazy-channel-recv-f (ms, Some (a', m'), as) () = (case as m' of
    None ⇒ do {
```

```
    a ← spmf-of-set (rnd η);
    return-spmf (None, Fail, None, as (m' := Some a))}
  | Some a ⇒ return-spmf (if a = a' then Some m' else None, Fail, None, as))
| lazy-channel-recv-f s                    () = return-spmf (None, s)

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

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

private inductive S :: (bool list cstate × (bool list × bool list) option × (bool list
⇒ bool list option)) spmf
  ⇒ ((bool × unit) × (bool list ⇒ bool list option) × (bool list × bool list)  cstate)
spmf ⇒ bool where
  S (return-spmf (Void, None, Map.empty))
    (return-spmf ((False, ()), Map.empty, Void))
| S (return-spmf (Store m, None, Map.empty))
    (map-spmf (λa. ((True, ()), [m ↦ a], Store (a, m))) (spmf-of-set (nlists UNIV
η)))
if length m = id' η
| S (return-spmf (Collect m, None, Map.empty))
     (map-spmf (λa. ((True, ()), [m ↦ a], Collect (a, m))) (spmf-of-set (nlists
UNIV η)))
if length m = id' η
| S (return-spmf (Store m, None, [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Store (a, m)))
if length m = id' η and length a = id' η
| S (return-spmf (Collect m, None, [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Collect (a, m)))
if length m = id' η and length a = id' η
| S (return-spmf (Fail, None, Map.empty))
    (map-spmf (λa. ((True, ()), [m ↦ a], Fail)) (spmf-of-set (nlists UNIV η)))
if length m = id' η
| S (return-spmf (Fail, None, [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Fail))
if length m = id' η and length a = id' η
| S (return-spmf (Collect m', Some (a', m'), Map.empty))
    (return-spmf ((False, ()), Map.empty, Collect (a', m')))
if length m' = id' η and length a' = id' η
| S (return-spmf (Collect m', Some (a', m'), [m ↦ a]))
    (return-spmf ((True, ()), [m ↦ a], Collect (a', m')))
if length m = id' η and length a = id' η and length m' = id' η and length a' =
id' η
| S (return-spmf (Collect m', Some (a', m'), Map.empty))
     (map-spmf (λx. ((True, ()), [m ↦ x], Collect (a', m'))) (spmf-of-set (nlists
```

UNIV η)))
if *length* m = *id'* η **and** *length* m' = *id'* η **and** *length* a' = *id'* η
| *S* (*map-spmf* (λx. (*Fail*, *None*, *as*($m' \mapsto x$))) *spmf-s*)
(*map-spmf* (λx. ((*False*, ()), *as*($m' \mapsto x$), *Fail*)) *spmf-s*)
if *length* m' = *id'* η **and** *lossless-spmf spmf-s*
| *S* (*map-spmf* (λx. (*Fail*, *None*, *as*($m' \mapsto x$))) *spmf-s*)
(*map-spmf* (λx. ((*True*, ()), *as*($m' \mapsto x$), *Fail*)) *spmf-s*)
if *length* m' = *id'* η **and** *lossless-spmf spmf-s*
| *S* (*return-spmf* (*Fail*, *None*, [$m' \mapsto a'$]))
(*map-spmf* (λx. ((*True*, ()), [$m \mapsto x$, $m' \mapsto a'$], *Fail*)) (*spmf-of-set* (*nlists* *UNIV* η)))
if *length* m = *id'* η **and** *length* m'= *id'* η **and** *length* a' = *id'* η
| *S* (*map-spmf* (λx. (*Fail*, *None*, [$m' \mapsto x$])) (*spmf-of-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 × *nlists* *UNIV* $\eta \cap \{x.$ *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 × *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*|
≤ q / *real* (2 ^ η) (**is** *?LHS* ≤ -)
⟨*proof*⟩ **inductive** *S'* :: ((((*bool list* × *bool list*) *option* + *unit*) × *unit* × *bool list* *cstate*) *spmf* ⇒
(*bool list cstate* × (*bool list* × *bool list*) *option* × (*bool list* ⇒ *bool list option*))

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

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

| S' (return-spmf (Inr (), (), Fail))
      (return-spmf (Collect m', Some (a', m'), [m ↦ a]))
if length m = id' η and length m' = id' η and length a' = id' η and m ≠ m'
| S' (return-spmf (Inr (), (), Fail))
      (return-spmf (Collect m', Some (a', m'), [m ↦ a]))
if length m = id' η and length m' = id' η and length a' = id' η  and a ≠ a'
| S' (return-spmf (Inl None, (), Collect m'))
      (return-spmf (Collect m', Some (a', m'), [m' ↦ a']))
if length m' = id' η and length a' = id' η
| S' (return-spmf (Inr (), (), Collect m'))
```

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

private lemma *trace-eq-sim*:
shows (*valid-insecQ* <+> *nlists UNIV* (*id'* 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 $\mathcal{I} \equiv$ *I-uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert None* (*Some* ‘ (*nlists UNIV* η × *nlists UNIV* η))) $\oplus_{\mathcal{I}}$
(*I-uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ *I-uniform UNIV* (*insert None* (*Some* ‘ *vld* η)))
assumes $\eta > 0$
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
$|spmf\ (connect\ (\mathcal{A}\ \eta)\ (CNV\ sim\ (Inl\ None)) \mid_=_{1_C} \triangleright channel.res\ auth\text{-}channel.auth\text{-}oracle))\ True\ -$
$spmf\ (connect\ (\mathcal{A}\ \eta)\ (macode.res\ (rnd\ \eta)\ (mac\ \eta)))\ True| \leq q\ /\ real\ (2\ \hat{}\ \eta)$
⟨*proof*⟩

end

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

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

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

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

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

lemma *plossless-dec* [*plossless-intro*]:
plossless-converter (*ℐ-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η))) (*ℐ-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ *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)
(λx. *case x of None* ⇒ *True* | *Some x′* ⇒ *length x′* = η)
(*ℐ-uniform UNIV* (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ *ℐ-full*)
⟨*proof*⟩

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

lemma *WT-enc* [*WT-intro*]: *ℐ-uniform* (*nlists UNIV* η) *UNIV*,
ℐ-uniform UNIV (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ *ℐ-uniform* (*vld* η) *UNIV* $\vdash_C$ *CIPHER.enc* η $\surd$
⟨*proof*⟩

lemma *WT-dec* [*WT-intro*]: *ℐ-uniform UNIV* (*insert None* (*Some* ‘ *nlists UNIV* η)),
ℐ-uniform UNIV (*nlists UNIV* η) $\oplus_{\mathcal{I}}$ *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* ≡ $\lambda\eta$. $\mathcal{I}$-*uniform* {*x*. *valid-mac-query* η *x*} *UNIV*
and $\mathcal{I}$-*ideal* ≡ λ-. $\mathcal{I}$-*full*
and $\mathcal{I}$-*common* ≡ $\lambda\eta$. $\mathcal{I}$-*uniform* (*vld* η) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*
shows
constructive-security
($\lambda\eta$. 1_C |= (*CIPHER.enc* η |= *CIPHER.dec* η) ⊙ *parallel-wiring* ▷
parallel-resource1-wiring ▷
CIPHER.KEY.res η ‖
(1_C |= *MAC.enm* η |= *MAC.dem* η ▷
1_C |= *parallel-wiring* ▷
parallel-resource1-wiring ▷ *MAC.RO.res* η ‖ *INSEC.res*))
(λ-. *SEC.res*)
($\lambda\eta$. *CNV Message-Authentication-Code.sim* (*Inl None*) ⊙ *CNV One-Time-Pad.sim None*)
($\lambda\eta$. $\mathcal{I}$-*uniform* (*Set.Collect* (*valid-mac-query* η)) (*insert None* (*Some* ‘ (*nlists UNIV* η × *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.