Constructive Cryptography in HOL: the Communication Modeling Aspect

Andreas Lochbihler and S. Reza Sefidgar

March 17, 2025

Abstract

Constructive Cryptography (CC) [8, 7, 9] introduces an abstract approach to composable security statements that allows one to focus on a particular aspect of security proofs at a time. Instead of proving the properties of concrete systems, CC studies system classes, i.e., the shared behavior of similar systems, and their transformations.

Modeling of systems communication plays a crucial role in composability and reusability of security statements; yet, this aspect has not been studied in any of the existing CC results. We extend our previous CC formalization [5, 6] with a new semantic domain called Fused Resource Templates (FRT) that abstracts over the systems communication patterns in CC proofs. This widens the scope of cryptography proof formalizations in the CryptHOL library [4, 3, 2].

This formalization is described in [1].

Contents

theory *More-CC* **imports**
Constructive-Cryptography.Constructive-Cryptography
begin

1 Material for Isabelle library

lemma *eq-alt-conversep*: (=) = (*BNF-Def.Grp UNIV id*)$^{-1-1}$
by(*simp add*: *Grp-def fun-eq-iff*)

parametric-constant
swap-parametric [*transfer-rule*]: *prod.swap-def*

lemma *Sigma-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-set A* ===> (*A* ===> *rel-set B*) ===> *rel-set* (*rel-prod A B*)) *Sigma Sigma*
unfolding *Sigma-def* **by** *transfer-prover*

lemma *empty-eq-Plus* [*simp*]: {} = *A* <+> *B* ⟷ *A* = {} ∧ *B* = {}
by *auto*

lemma *insert-Inl-Plus* [*simp*]: *insert* (*Inl x*) (*A* <+> *B*) = *insert x A* <+> *B* **by** *auto*

lemma *insert-Inr-Plus* [*simp*]: *insert* (*Inr x*) (*A* <+> *B*) = *A* <+> *insert x B*
by *auto*

lemma *map-sum-image-Plus* [*simp*]: *map-sum f g* ‘ (*A* <+> *B*) = *f* ‘ *A* <+> *g* ‘ *B*
by(*auto intro*: *rev-image-eqI*)

lemma *Plus-subset-Plus-iff* [*simp*]: *A* <+> *B* ⊆ *C* <+> *D* ⟷ *A* ⊆ *C* ∧ *B* ⊆ *D*
by *auto*

lemma *map-sum-eq-Inl-iff*: *map-sum f g x* = *Inl y* ⟷ (∃ *x'*. *x* = *Inl x'* ∧ *y* = *f x'*)
by(*cases x*) *auto*

lemma *map-sum-eq-Inr-iff*: *map-sum f g x* = *Inr y* ⟷ (∃ *x'*. *x* = *Inr x'* ∧ *y* = *g x'*)
by(*cases x*) *auto*

lemma *surj-map-sum*: *surj* (*map-sum f g*) **if** *surj f surj g*
apply(*safe*; *simp*)
subgoal for *x* **using** *that*
by(*cases x*)(*auto 4 3 intro*: *image-eqI*[**where** *x*=*Inl* -] *image-eqI*[**where** *x*=*Inr* -])
done

lemma *bij-map-sumI* [*simp*]: *bij* (*map-sum f g*) **if** *bij f bij g*

using *that* **by**(*clarsimp simp add*: *bij-def sum.inj-map surj-map-sum*)

lemma *inv-map-sum* [*simp*]:
⟦ *bij f*; *bij g* ⟧ $\Longrightarrow$ *inv-into UNIV* (*map-sum f g*) = *map-sum* (*inv-into UNIV f*) (*inv-into UNIV g*)
by(*rule inj-imp-inv-eq*)(*simp-all add*: *sum.map-comp sum.inj-map bij-def surj-iff sum.map-id*)

context *conditionally-complete-lattice* **begin**

lemma *admissible-le1I*:
ccpo.admissible lub ord ($\lambda x.\ f\ x \le y$)
if *cont lub ord Sup* ($\le$) *f*
by(*rule ccpo.admissibleI*)(*auto simp add*: *that*[*THEN contD*] *intro*!: *cSUP-least*)

lemma *admissible-le1-mcont* [*cont-intro*]:
ccpo.admissible lub ord ($\lambda x.\ f\ x \le y$) **if** *mcont lub ord Sup* ($\le$) *f*
using *that* **by**(*simp add*: *admissible-le1I mcont-def*)

end

lemma *eq-alt-conversep2*: (=) = ((*BNF-Def.Grp UNIV id*)$^{-1-1}$)$^{-1-1}$
by(*auto simp add*: *Grp-def fun-eq-iff*)

lemma *nn-integral-indicator-singleton1* [*simp*]:
assumes [*measurable*]: $\{y\} \in$ *sets M*
shows ($\int^+ x.$ *indicator* $\{y\}$ $x * f\ x\ \partial M$) = *emeasure M* $\{y\} * f\ y$
by(*simp add*: *mult.commute*)

lemma *nn-integral-indicator-singleton1'* [*simp*]:
assumes $\{y\} \in$ *sets M*
shows ($\int^+ x.$ *indicator* $\{x\}$ $y * f\ x\ \partial M$) = *emeasure M* $\{y\} * f\ y$
by(*subst nn-integral-indicator-singleton1*[*symmetric*, *OF assms*])(*rule nn-integral-cong*; *simp split*: *split-indicator*)

1.1 Probabilities

lemma *pmf-eq-1-iff*: *pmf p x* = *1* $\longleftrightarrow$ *p* = *return-pmf x* (**is** *?lhs* = *?rhs*)
proof(*rule iffI*)
assume *?lhs*
have *pmf p i* = *0* **if** $x \ne i$ **for** *i*
proof(*rule antisym*)
have *pmf p i* + *1* $\le$ *pmf p i* + *pmf p x* **using** ‹*?lhs*› **by** *simp*
also have ... = *measure* (*measure-pmf p*) $\{i, x\}$ **using** *that*
by(*subst measure-pmf.finite-measure-eq-sum-singleton*)(*simp-all add*: *pmf.rep-eq*)
also have ... $\le$ *1* **by**(*rule measure-pmf.subprob-measure-le-1*)
finally show *pmf p i* $\le$ *0* **by** *simp*
qed(*rule pmf-nonneg*)

then show *?rhs* **if** *?lhs*
by(*intro pmf-eqI*)(*auto simp add*: *that split*: *split-indicator*)
qed *simp*

lemma *measure-spmf-cong*: *measure* (*measure-spmf p*) *A* = *measure* (*measure-spmf p*) *B*
if *A* ∩ *set-spmf p* = *B* ∩ *set-spmf p*
proof −
have *measure* (*measure-spmf p*) *A* = *measure* (*measure-spmf p*) (*A* ∩ *set-spmf p*) + *measure* (*measure-spmf p*) (*A* − *set-spmf p*)
by(*subst measure-spmf.finite-measure-Union*[*symmetric*])(*auto intro*: *arg-cong2*[**where** *f*=*measure*])
also have *measure* (*measure-spmf p*) (*A* − *set-spmf p*) = *0* **by**(*simp add*: *measure-spmf-zero-iff*)
also have *0* = *measure* (*measure-spmf p*) (*B* − *set-spmf p*) **by**(*simp add*: *measure-spmf-zero-iff*)
also have *measure* (*measure-spmf p*) (*A* ∩ *set-spmf p*) + . . . = *measure* (*measure-spmf p*) *B*
unfolding *that* **by**(*subst measure-spmf.finite-measure-Union*[*symmetric*])(*auto intro*: *arg-cong2*[**where** *f*=*measure*])
finally show *?thesis* **.**
qed

definition *weight-spmf′* **where** *weight-spmf′* = *weight-spmf*
lemma *weight-spmf′-parametric* [*transfer-rule*]: *rel-fun* (*rel-spmf A*) (=) *weight-spmf′ weight-spmf′*
unfolding *weight-spmf′-def* **by**(*rule weight-spmf-parametric*)

lemma *bind-spmf-to-nat-on*:
bind-spmf (*map-spmf* (*to-nat-on* (*set-spmf p*)) *p*) (λ*n*. *f* (*from-nat-into* (*set-spmf p*) *n*)) = *bind-spmf p f*
by(*simp add*: *bind-map-spmf cong*: *bind-spmf-cong*)

lemma *try-cond-spmf-fst*:
try-spmf (*cond-spmf-fst p x*) *q* = (*if x* ∈ *fst* ‘ *set-spmf p then cond-spmf-fst p x else q*)
by (*metis cond-spmf-fst-eq-return-None lossless-cond-spmf-fst try-spmf-lossless try-spmf-return-None*)

lemma *measure-try-spmf*:
measure (*measure-spmf* (*try-spmf p q*)) *A* = *measure* (*measure-spmf p*) *A* + *pmf p None* ∗ *measure* (*measure-spmf q*) *A*
proof −
have *emeasure* (*measure-spmf* (*try-spmf p q*)) *A* = *emeasure* (*measure-spmf p*) *A* + *pmf p None* ∗ *emeasure* (*measure-spmf q*) *A*
by(*fold nn-integral-spmf*)(*simp add*: *spmf-try-spmf nn-integral-add ennreal-mult′ nn-integral-cmult*)
then show *?thesis* **by**(*simp add*: *measure-spmf.emeasure-eq-measure ennreal-mult′*[*symmetric*] *ennreal-plus*[*symmetric*] *del*: *ennreal-plus*)

```
qed

lemma rel-spmf-OO-trans-strong:
  ⟦ rel-spmf R p q; rel-spmf S q r ⟧ ⟹ rel-spmf (R OO eq-onp (λx. x ∈ set-spmf q) OO S) p r
  by(auto intro: rel-spmf-OO-trans rel-spmf-reflI simp add: eq-onp-def)

lemma mcont2mcont-spmf [cont-intro]:
  mcont lub ord Sup (≤) (λp. spmf (f p) x)
  if mcont lub ord lub-spmf (ord-spmf (=)) f
  using that unfolding mcont-def
  apply safe
  subgoal by(rule monotone2monotone, rule monotone-spmf; simp)
  apply(rule contI)
  apply(subst contD[where f=f and luba=lub]; simp)
  apply(subst cont-spmf[THEN contD])
    apply(erule (2) chain-imageI[OF - monotoneD])
   apply simp
  apply(simp add: image-image)
  done

lemma ord-spmf-try-spmf2: ord-spmf R p (try-spmf p q) if rel-spmf R p p
proof –
  have ord-spmf R (bind-pmf p return-pmf) (try-spmf p q) unfolding try-spmf-def
    by(rule rel-pmf-bindI[where R=rel-option R])
      (use that in ‹auto simp add: rel-pmf-return-pmf1 elim!: option.rel-cases›)
  then show ?thesis by(simp add: bind-return-pmf')
qed

lemma ord-spmf-lossless-spmfD1:
  assumes ord-spmf R p q
    and lossless-spmf p
  shows rel-spmf R p q
   by (metis (no-types, lifting) assms lossless-iff-set-pmf-None option.simps(11) ord-option.cases pmf.rel-mono-strong)

lemma restrict-spmf-mono:
  ord-spmf (=) p q ⟹ ord-spmf (=) (p ↿ A) (q ↿ A)
  by(auto simp add: restrict-spmf-def pmf.rel-map elim!: pmf.rel-mono-strong elim: ord-option.cases)

lemma restrict-lub-spmf:
  assumes chain: Complete-Partial-Order.chain (ord-spmf (=)) Y
  shows restrict-spmf (lub-spmf Y) A = lub-spmf ((λp. restrict-spmf p A) ` Y) (is ?lhs = ?rhs)
proof(cases Y = {})
  case Y: False
  have chain': Complete-Partial-Order.chain (ord-spmf (=)) ((λp. p ↿ A) ` Y)
    using chain by(rule chain-imageI)(auto intro: restrict-spmf-mono)
```

show *?thesis* **by**(*rule spmf-eqI*)(*simp add*: *spmf-lub-spmf*[*OF chain'*] *Y image-image spmf-restrict-spmf spmf-lub-spmf*[*OF chain*])
qed *simp*

lemma *mono2mono-restrict-spmf* [*THEN spmf.mono2mono*]:
shows *monotone-restrict-spmf*: *monotone* (*ord-spmf* (=)) (*ord-spmf* (=)) (*λp. p* ↿ *A*)
by(*rule monotoneI*)(*rule restrict-spmf-mono*)

lemma *mcont2mcont-restrict-spmf* [*THEN spmf.mcont2mcont*, *cont-intro*]:
shows *mcont-restrict-spmf*: *mcont lub-spmf* (*ord-spmf* (=)) *lub-spmf* (*ord-spmf* (=)) (*λp. restrict-spmf p A*)
using *monotone-restrict-spmf* **by**(*rule mcontI*)(*simp add*: *contI restrict-lub-spmf*)

lemma *ord-spmf-case-option*: *ord-spmf R* (*case x of None* ⇒ *a* | *Some y* ⇒ *b y*) (*case x of None* ⇒ *a′* | *Some y* ⇒ *b′ y*)
if *ord-spmf R a a′* ⋀*y. ord-spmf R* (*b y*) (*b′ y*) **using** *that* **by**(*cases x*) *auto*

lemma *ord-spmf-map-spmfI*: *ord-spmf* (=) (*map-spmf f p*) (*map-spmf f q*) **if** *ord-spmf* (=) *p q*
using *that* **by**(*auto simp add*: *pmf.rel-map elim*!: *pmf.rel-mono-strong ord-option.cases*)

1.1.1 Conditional probabilities

lemma *mk-lossless-cond-spmf* [*simp*]: *mk-lossless* (*cond-spmf p A*) = *cond-spmf p A*
by(*simp add*: *cond-spmf-alt*)

context
fixes *p* :: *′a pmf*
and *f* :: *′a* ⇒ *′b pmf*
and *A* :: *′b set*
and *F* :: *′a* ⇒ *real*
defines *F* ≡ *λx. pmf p x* ∗ *measure* (*measure-pmf* (*f x*)) *A* / *measure* (*measure-pmf* (*bind-pmf p f*)) *A*
begin

definition *cond-bind-pmf* :: *′a pmf* **where** *cond-bind-pmf* = *embed-pmf F*

lemma *cond-bind-pmf-nonneg*: *F x* ≥ *0*
by(*simp add*: *F-def*)

context assumes *defined*: *A* ∩ (⋃*x*∈*set-pmf p. set-pmf* (*f x*)) ≠ {} **begin**

private lemma *nonzero*: *measure* (*measure-pmf* (*bind-pmf p f*)) *A* > *0*
using *defined* **by**(*auto 4 3 intro*: *measure-pmf-posI*)

lemma *cond-bind-pmf-prob*: (∫⁺ *x. F x* ∂*count-space UNIV*) = *1*
proof –

have *nonzero'*: $(\int^+ x.\ ennreal\ (pmf\ p\ x) * ennreal\ (measure\text{-}pmf.prob\ (f\ x)\ A)\ \partial count\text{-}space\ UNIV) \neq 0$
using *defined* **by**(*auto simp add*: *nn-integral-0-iff-AE AE-count-space pmf-eq-0-set-pmf measure-pmf-zero-iff*)
have *finite*: $(\int^+ x.\ ennreal\ (pmf\ p\ x) * ennreal\ (measure\text{-}pmf.prob\ (f\ x)\ A)\ \partial count\text{-}space\ UNIV) < \top$ (**is** *?lhs* < -)
proof(*rule order.strict-trans1*)
show $?lhs \leq (\int^+ x.\ ennreal\ (pmf\ p\ x) * 1\ \partial count\text{-}space\ UNIV)$
by(*rule nn-integral-mono*)(*simp add*: *mult-left-le*)
show ... $< \top$ **by**(*simp add*: *nn-integral-pmf-eq-1*)
qed
have $(\int^+ x.\ F\ x\ \partial count\text{-}space\ UNIV) =$
$(\sum^+ x.\ ennreal\ (pmf\ p\ x * measure\text{-}pmf.prob\ (f\ x)\ A))\ /\ emeasure\ (measure\text{-}pmf\ (bind\text{-}pmf\ p\ f))\ A$
using *nonzero* **unfolding** *F-def measure-pmf.emeasure-eq-measure*
by(*simp add*: *divide-ennreal*[*symmetric*] *divide-ennreal-def nn-integral-multc*)
also have ... = *1* **unfolding** *emeasure-bind-pmf*
by(*simp add*: *measure-pmf.emeasure-eq-measure nn-integral-measure-pmf ennreal-mult' nonzero' finite*)
finally show *?thesis* **.**
qed

lemma *pmf-cond-bind-pmf*: *pmf cond-bind-pmf x* = *F x*
unfolding *cond-bind-pmf-def* **by**(*rule pmf-embed-pmf*[*OF cond-bind-pmf-nonneg cond-bind-pmf-prob*])

lemma *set-cond-bind-pmf*: *set-pmf cond-bind-pmf* = $\{x \in set\text{-}pmf\ p.\ set\text{-}pmf\ (f\ x) \cap A \neq \{\}\}$
unfolding *cond-bind-pmf-def*
by(*subst set-embed-pmf*[*OF cond-bind-pmf-nonneg cond-bind-pmf-prob*])
(*auto simp add*: *F-def pmf-eq-0-set-pmf measure-pmf-zero-iff*)

lemma *cond-bind-pmf*: *cond-pmf* (*bind-pmf p f*) *A* = *bind-pmf cond-bind-pmf* ($\lambda x.$ *cond-pmf* (*f x*) *A*)
(**is** *?lhs* = *?rhs*)
proof(*rule pmf-eqI*)
fix *i*
have *ennreal* (*pmf ?lhs i*) = *ennreal* (*pmf ?rhs i*)
proof(*cases* $i \in A$)
case *True*
have $ennreal\ (pmf\ ?lhs\ i) = (\int^+ x.\ ennreal\ (pmf\ p\ x) * ennreal\ (pmf\ (f\ x)\ i)\ /\ ennreal\ (measure\text{-}pmf.prob\ (p \gg= f)\ A)\ \partial count\text{-}space\ UNIV)$
using *True defined*
by(*simp add*: *pmf-cond bind-UNION Int-commute divide-ennreal*[*symmetric*] *nonzero ennreal-pmf-bind*)
(*simp add*: *divide-ennreal-def nn-integral-multc*[*symmetric*] *nn-integral-measure-pmf*)
also have ... = $(\int^+ x.\ ennreal\ (F\ x) * ennreal\ (pmf\ (cond\text{-}pmf\ (f\ x)\ A)\ i)\ \partial count\text{-}space\ UNIV)$
using *True nonzero*

```
      apply(intro nn-integral-cong)
      subgoal for x
        by(clarsimp simp add: F-def ennreal-mult'[symmetric] divide-ennreal)
       (cases measure-pmf.prob (f x) A = 0; auto simp add: pmf-cond pmf-eq-0-set-pmf
measure-pmf-zero-iff)
      done
    also have ... = ennreal (pmf ?rhs i)
      by(simp add: ennreal-pmf-bind nn-integral-measure-pmf pmf-cond-bind-pmf)
    finally show ?thesis .
  next
    case False
    then show ?thesis using defined
    by(simp add: pmf-cond bind-UNION Int-commute pmf-eq-0-set-pmf set-cond-bind-pmf)
  qed
  then show pmf ?lhs i = pmf ?rhs i by simp
qed

end

end


lemma cond-spmf-try1:
  cond-spmf (try-spmf p q) A = cond-spmf p A if set-spmf q ∩ A = {}
  apply(rule spmf-eqI)
  using that
  apply(auto simp add: spmf-try-spmf measure-try-spmf measure-spmf-zero-iff)
  apply(subst (2) spmf-eq-0-set-spmf[THEN iffD2])
   apply blast
  apply simp
  apply(simp add: measure-try-spmf measure-spmf-zero-iff)
  done

lemma cond-spmf-cong: cond-spmf p A = cond-spmf p B if A ∩ set-spmf p = B
∩ set-spmf p
  apply(rule spmf-eqI)
   using that by(auto simp add: measure-spmf-zero-iff spmf-eq-0-set-spmf mea-
sure-spmf-cong[OF that])

lemma cond-spmf-pair-spmf:
  cond-spmf (pair-spmf p q) (A × B) = pair-spmf (cond-spmf p A) (cond-spmf q
B) (is ?lhs = ?rhs)
proof(rule spmf-eqI)
  show spmf ?lhs i = spmf ?rhs i for i
  proof(cases i)
    case i [simp]: (Pair a b)
    then show ?thesis by(simp add: measure-pair-spmf-times)
  qed
qed
```

lemma *cond-spmf-pair-spmf1*:
cond-spmf-fst (*map-spmf* (λ((*x*, *s'*), *y*). (*f x*, *s'*, *y*)) (*pair-spmf p q*)) *x* =
pair-spmf (*cond-spmf-fst* (*map-spmf* (λ(*x*, *s'*). (*f x*, *s'*)) *p*) *x*) *q* (**is** *?lhs* = *?rhs*)
if *lossless-spmf q*
proof −
have *?lhs* = *map-spmf* (λ((-, *s'*), *y*). (*s'*, *y*)) (*cond-spmf* (*pair-spmf p q*) ((λ((*x*, *s'*), *y*). (*f x*, *s'*, *y*)) −' ({*x*} × *UNIV*)))
by(*simp add*: *cond-spmf-fst-def spmf.map-comp o-def split-def*)
also have ((λ((*x*, *s'*), *y*). (*f x*, *s'*, *y*)) −' ({*x*} × *UNIV*)) = ((λ(*x*, *y*). (*f x*, *y*)) −' ({*x*} × *UNIV*)) × *UNIV*
by(*auto*)
also have *map-spmf* (λ((-, *s'*), *y*). (*s'*, *y*)) (*cond-spmf* (*pair-spmf p q*) ...) = *?rhs*
by(*simp add*: *cond-spmf-fst-def cond-spmf-pair-spmf that spmf.map-comp pair-map-spmf1 apfst-def map-prod-def split-def*)
finally show *?thesis* **.**
qed

lemma *try-cond-spmf*: *try-spmf* (*cond-spmf p A*) *q* = (*if set-spmf p* ∩ *A* ≠ {} *then cond-spmf p A else q*)
apply(*clarsimp simp add*: *cond-spmf-def lossless-iff-set-pmf-None intro*!: *try-spmf-lossless*)
apply(*subst* (*asm*) *set-cond-pmf*)
apply(*auto simp add*: *in-set-spmf*)
done

lemma *cond-spmf-try2*:
cond-spmf (*try-spmf p q*) *A* = (*if lossless-spmf p then return-pmf None else cond-spmf q A*) **if** *set-spmf p* ∩ *A* = {}
apply(*rule spmf-eqI*)
using *that*
apply(*auto simp add*: *spmf-try-spmf measure-try-spmf measure-spmf-zero-iff lossless-iff-pmf-None*)
apply(*subst spmf-eq-0-set-spmf*[*THEN iffD2*])
apply *blast*
apply(*simp add*: *measure-spmf-zero-iff*[*THEN iffD2*])
done

definition *cond-bind-spmf* :: *'a spmf* ⇒ (*'a* ⇒ *'b spmf*) ⇒ *'b set* ⇒ *'a spmf* **where**

cond-bind-spmf p f A =
(*if* ∃ *x*∈*set-spmf p*. *set-spmf* (*f x*) ∩ *A* ≠ {} *then*
cond-bind-pmf p (λ*x*. *case x of None* ⇒ *return-pmf None* | *Some x* ⇒ *f x*) (*Some* ' *A*)
else return-pmf None)

context begin

private lemma *defined*: ⟦ $y \in$ *set-spmf* (*f x*); $y \in A$; $x \in$ *set-spmf p* ⟧
⟹ *Some* ‘ $A \cap (\bigcup x{\in}$*set-pmf p*. *set-pmf* (*case x of None* ⇒ *return-pmf None* | *Some x* ⇒ *f x*)) $\neq$ {}
by(*fastforce simp add*: *in-set-spmf bind-spmf-def*)

lemma *spmf-cond-bind-spmf* [*simp*]:
spmf (*cond-bind-spmf p f A*) *x* = *spmf p x* ∗ *measure* (*measure-spmf* (*f x*)) *A* / *measure* (*measure-spmf* (*bind-spmf p f*)) *A*
by(*clarsimp simp add*: *cond-bind-spmf-def measure-spmf-zero-iff bind-UNION pmf-cond-bind-pmf defined split*!: *if-split*)
(*fastforce simp add*: *in-set-spmf bind-spmf-def measure-measure-spmf-conv-measure-pmf*)+

lemma *set-cond-bind-spmf* [*simp*]:
set-spmf (*cond-bind-spmf p f A*) = {*x*∈*set-spmf p*. *set-spmf* (*f x*) ∩ *A* ≠ {}}
by(*clarsimp simp add*: *cond-bind-spmf-def set-spmf-def bind-UNION*)
(*subst set-cond-bind-pmf*; *fastforce simp add*: *measure-measure-spmf-conv-measure-pmf*)

lemma *cond-bind-spmf*: *cond-spmf* (*bind-spmf p f*) *A* = *bind-spmf* (*cond-bind-spmf p f A*) (λ*x*. *cond-spmf* (*f x*) *A*)
by(*auto simp add*: *cond-spmf-def bind-UNION cond-bind-spmf-def split*!: *if-splits*)
(*fastforce split*: *option.splits simp add*: *cond-bind-pmf set-cond-bind-pmf defined in-set-spmf bind-spmf-def intro*!: *bind-pmf-cong*[*OF refl*])

end

lemma *cond-spmf-fst-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-spmf* (*rel-prod* (=) *B*) ===> (=) ===> *rel-spmf B*) *cond-spmf-fst cond-spmf-fst*
apply(*rule rel-funI*)+
apply(*clarsimp simp add*: *cond-spmf-fst-def spmf-rel-map elim*!: *rel-spmfE*)
subgoal for *x pq*
by(*subst* (*1 2*) *cond-spmf-cong*[**where** *B*=*fst* −‘ ({*x*} × *UNIV*) ∩ *snd* −‘ ({*x*} × *UNIV*)])
(*fastforce intro*: *rel-spmf-reflI*)+
done

lemma *cond-spmf-fst-map-prod*:
cond-spmf-fst (*map-spmf* (λ(*x*, *y*). (*f x*, *g x y*)) *p*) (*f x*) = *map-spmf* (*g x*) (*cond-spmf-fst p x*)
if *inj-on f* (*insert x* (*fst* ‘ *set-spmf p*))
proof −
have *cond-spmf p* ((λ(*x*, *y*). (*f x*, *g x y*)) −‘ ({*f x*} × *UNIV*)) = *cond-spmf p* (((λ(*x*, *y*). (*f x*, *g x y*)) −‘ ({*f x*} × *UNIV*)) ∩ *set-spmf p*)
by(*rule cond-spmf-cong*) *simp*
also have ((λ(*x*, *y*). (*f x*, *g x y*)) −‘ ({*f x*} × *UNIV*)) ∩ *set-spmf p* = ({*x*} × *UNIV*) ∩ *set-spmf p*
using *that* **by**(*auto 4 3 dest*: *inj-onD intro*: *rev-image-eqI*)
also have *cond-spmf p* . . . = *cond-spmf p* ({*x*} × *UNIV*)
by(*rule cond-spmf-cong*) *simp*
finally show *?thesis*

by(*auto simp add*: *cond-spmf-fst-def spmf.map-comp o-def split-def intro*: *map-spmf-cong*)
qed

lemma *cond-spmf-fst-map-prod-inj*:
cond-spmf-fst (*map-spmf* (λ(*x*, *y*). (*f x*, *g x y*)) *p*) (*f x*) = *map-spmf* (*g x*) (*cond-spmf-fst p x*)
if *inj f*
apply(*rule cond-spmf-fst-map-prod*)
using *that* **by**(*simp add*: *inj-on-def*)

definition *cond-bind-spmf-fst* :: *'a spmf* ⇒ (*'a* ⇒ *'b spmf*) ⇒ *'b* ⇒ *'a spmf* **where**
cond-bind-spmf-fst p f x = *cond-bind-spmf p* (*map-spmf* (λ*b*. (*b*, ())) ∘ *f*) ({*x*} × *UNIV*)

lemma *cond-bind-spmf-fst-map-spmf-fst*:
cond-bind-spmf-fst p (*map-spmf fst* ∘ *f*) *x* = *cond-bind-spmf p f* ({*x*} × *UNIV*)
(**is** *?lhs* = *?rhs*)
proof −
have [*simp*]: (λ*x*. (*fst x*, ())) −' ({*x*} × *UNIV*) = {*x*} × *UNIV* **by** *auto*
have *?lhs* = *cond-bind-spmf p* (λ*x*. *map-spmf* (λ*x*. (*fst x*, ())) (*f x*)) ({*x*} × *UNIV*)
by(*simp add*: *cond-bind-spmf-fst-def spmf.map-comp o-def*)
also have ... = *?rhs* **by**(*rule spmf-eqI*)(*simp add*: *measure-map-spmf map-bind-spmf*[*unfolded o-def*, *symmetric*])
finally show *?thesis* **.**
qed

lemma *cond-spmf-fst-bind*: *cond-spmf-fst* (*bind-spmf p f*) *x* =
bind-spmf (*cond-bind-spmf-fst p* (*map-spmf fst* ∘ *f*) *x*) (λ*y*. *cond-spmf-fst* (*f y*) *x*)
by(*simp add*: *cond-spmf-fst-def cond-bind-spmf map-bind-spmf cond-bind-spmf-fst-map-spmf-fst*)(*simp add*: *o-def*)

lemma *spmf-cond-bind-spmf-fst* [*simp*]:
spmf (*cond-bind-spmf-fst p f x*) *i* = *spmf p i* ∗ *spmf* (*f i*) *x* / *spmf* (*bind-spmf p f*) *x*
by(*simp add*: *cond-bind-spmf-fst-def*)
(*auto simp add*: *spmf-conv-measure-spmf measure-map-spmf map-bind-spmf*[*symmetric*] *intro*!: *arg-cong2*[**where** *f*=(/)] *arg-cong2*[**where** *f*=(∗)] *arg-cong2*[**where** *f*=*measure*])

lemma *set-cond-bind-spmf-fst* [*simp*]:
set-spmf (*cond-bind-spmf-fst p f x*) = {*y* ∈ *set-spmf p*. *x* ∈ *set-spmf* (*f y*)}
by(*auto simp add*: *cond-bind-spmf-fst-def intro*: *rev-image-eqI*)

lemma *map-cond-spmf-fst*: *map-spmf f* (*cond-spmf-fst p x*) = *cond-spmf-fst* (*map-spmf* (*apsnd f*) *p*) *x*
by(*auto simp add*: *cond-spmf-fst-def spmf.map-comp intro*!: *map-spmf-cong arg-cong2*[**where** *f*=*cond-spmf*])

lemma *cond-spmf-fst-try1*:
cond-spmf-fst (*try-spmf p q*) *x* = *cond-spmf-fst p x* **if** $x \notin$ *fst* ‘ *set-spmf q*
using *that*
apply(*simp add*: *cond-spmf-fst-def*)
apply(*subst cond-spmf-try1*)
apply(*auto intro*: *rev-image-eqI*)
done

lemma *cond-spmf-fst-try2*:
cond-spmf-fst (*try-spmf p q*) *x* = (*if lossless-spmf p then return-pmf None else cond-spmf-fst q x*) **if** $x \notin$ *fst* ‘ *set-spmf p*
using *that*
apply(*simp add*: *cond-spmf-fst-def split*!: *if-split*)
apply (*metis cond-spmf-fst-def cond-spmf-fst-eq-return-None*)
by (*metis cond-spmf-fst-def cond-spmf-try2 lossless-cond-spmf lossless-cond-spmf-fst lossless-map-spmf*)

lemma *cond-spmf-fst-map-inj*:
cond-spmf-fst (*map-spmf* (*apfst f*) *p*) (*f x*) = *cond-spmf-fst p x* **if** *inj f*
by(*auto simp add*: *cond-spmf-fst-def spmf.map-comp intro*!: *map-spmf-cong arg-cong2*[**where** *f*=*cond-spmf*] *dest*: *injD*[*OF that*])

lemma *cond-spmf-fst-pair-spmf1*:
cond-spmf-fst (*map-spmf* (λ(*x*, *y*). (*f x*, *g x y*)) (*pair-spmf p q*)) *a* =
bind-spmf (*cond-spmf-fst* (*map-spmf* (λ*x*. (*f x*, *x*)) *p*) *a*) (λ*x*. *map-spmf* (*g x*) (*mk-lossless q*)) (**is** *?lhs* = *?rhs*)
proof −
have (λ(*x*, *y*). (*f x*, *g x y*)) −‘ ({*a*} × *UNIV*) = *f* −‘ {*a*} × *UNIV* **by**(*auto*)
moreover have (λ*x*. (*f x*, *x*)) −‘ ({*a*} × *UNIV*) = *f* −‘ {*a*} **by** *auto*
ultimately show *?thesis*
by(*simp add*: *cond-spmf-fst-def spmf.map-comp o-def split-beta cond-spmf-pair-spmf*)
(*simp add*: *pair-spmf-alt-def map-bind-spmf o-def map-spmf-conv-bind-spmf*)
qed

lemma *cond-spmf-fst-return-spmf'*:
cond-spmf-fst (*return-spmf* (*x*, *y*)) *z* = (*if x* = *z then return-spmf y else return-pmf None*)
by(*simp add*: *cond-spmf-fst-def*)

2 Material for CryptHOL

lemma *left-gpv-lift-spmf* [*simp*]: *left-gpv* (*lift-spmf p*) = *lift-spmf p*
by(*rule gpv.expand*)(*simp add*: *spmf.map-comp o-def*)

lemma *right-gpv-lift-spmf* [*simp*]: *right-gpv* (*lift-spmf p*) = *lift-spmf p*
by(*rule gpv.expand*)(*simp add*: *spmf.map-comp o-def*)

lemma *map'-lift-spmf*: *map-gpv' f g h* (*lift-spmf p*) = *lift-spmf* (*map-spmf f p*)
by(*rule gpv.expand*)(*simp add*: *gpv.map-sel spmf.map-comp o-def*)

lemma *in-set-sample-uniform* [*simp*]: $x \in$ *set-spmf* (*sample-uniform n*) $\longleftrightarrow x <$ n
  by(*simp add*: *sample-uniform-def*)

lemma (**in** *cyclic-group*) *inj-on-generator-iff* [*simp*]: ⟦ $x <$ *order G*; $y <$ *order G* ⟧ $\Longrightarrow$ **g** [^] x = **g** [^] $y \longleftrightarrow x = y$
  using *inj-on-generator* **by**(*auto simp add*: *inj-on-def*)

lemma *map-$\mathcal{I}$-bot* [*simp*]: *map-$\mathcal{I}$ f g* $\bot = \bot$
  unfolding *bot-$\mathcal{I}$-def map-$\mathcal{I}$-$\mathcal{I}$-uniform* **by** *simp*

lemma *map-$\mathcal{I}$-Inr-plus* [*simp*]: *map-$\mathcal{I}$ Inr f* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) = *map-$\mathcal{I}$ id* ($f \circ$ *Inr*) $\mathcal{I}2$
  by(*rule $\mathcal{I}$-eqI*) *auto*

lemma *interaction-bound-map-gpv′-le*:
  defines *ib* ≡ *interaction-bound*
  shows *interaction-bound consider* (*map-gpv′ f g h gpv*) ≤ *ib* (*consider* ∘ *g*) *gpv*
proof(*induction arbitrary*: *gpv rule*: *interaction-bound-fixp-induct*)
  case *adm* **show** *?case* **by** *simp*
  case *bottom* **show** *?case* **by** *simp*
  case (*step interaction-bound′*)
  show *?case* **unfolding** *ib-def*
    by(*subst interaction-bound.simps*)
     (*auto simp add*: *image-comp ib-def split*: *generat.split intro*!: *SUP-mono rev-bexI step.IH*[*unfolded ib-def*])
qed

lemma *interaction-bounded-by-map-gpv′* [*interaction-bound*]:
  assumes *interaction-bounded-by* (*consider* ∘ *g*) *gpv n*
  shows *interaction-bounded-by consider* (*map-gpv′ f g h gpv*) *n*
  using *assms interaction-bound-map-gpv′-le*[*of consider f g h gpv*] **by**(*simp add*: *interaction-bounded-by.simps*)

lemma *map-gpv′-bind-gpv*:
  map-gpv′ f g h (*bind-gpv gpv F*) = *bind-gpv* (*map-gpv′ id g h gpv*) (λ*x*. *map-gpv′ f g h* (*F x*))
  by(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
   (*auto simp del*: *bind-gpv-sel′ simp add*: *bind-gpv.sel spmf-rel-map bind-map-spmf generat.rel-map rel-fun-def intro*!: *rel-spmf-bind-reflI rel-spmf-reflI generat.rel-refl-strong split*!: *generat.split*)

lemma *exec-gpv-map-gpv′*:
  exec-gpv callee (*map-gpv′ f g h gpv*) *s* =
  map-spmf (*map-prod f id*) (*exec-gpv* (*map-fun id* (*map-fun g* (*map-spmf* (*map-prod h id*))) *callee*) *gpv s*)
  using *exec-gpv-parametric′*[
   where *S*=(=) **and** *CALL*=*BNF-Def.Grp UNIV g* **and** *R*=*conversep* (*BNF-Def.Grp UNIV h*) **and** *A*=*BNF-Def.Grp UNIV f*,

unfolded rel-gpv''-Grp, simplified]
apply(*subst* (*asm*) (*2*) *conversep-eq*[*symmetric*])
apply(*subst* (*asm*) *prod.rel-conversep*)
apply(*subst* (*asm*) (*2 4*) *eq-alt*)
apply(*subst* (*asm*) *prod.rel-Grp*)
apply *simp*
apply(*subst* (*asm*) *spmf-rel-conversep*)
apply(*subst* (*asm*) *option.rel-Grp*)
apply(*subst* (*asm*) *pmf.rel-Grp*)
apply *simp*
apply(*subst* (*asm*) *prod.rel-Grp*)
apply *simp*
apply(*subst* (*asm*) (*1 3*) *conversep-conversep*[*symmetric*])
apply(*subst* (*asm*) *rel-fun-conversep*)
apply(*subst* (*asm*) *rel-fun-Grp*)
apply(*subst* (*asm*) *rel-fun-conversep*)
apply *simp*
apply(*simp add*: *option.rel-Grp pmf.rel-Grp fun.rel-Grp*)
apply(*simp add*: *rel-fun-def BNF-Def.Grp-def o-def map-fun-def*)
apply(*erule allE*)+
apply(*drule fun-cong*)
apply(*erule trans*)
apply *simp*
done

lemma *colossless-gpv-sub-gpvs*:
assumes *colossless-gpv* $\mathcal{I}$ *gpv gpv'* $\in$ *sub-gpvs* $\mathcal{I}$ *gpv*
shows *colossless-gpv* $\mathcal{I}$ *gpv'*
using *assms*(*2*,*1*) **by**(*induction*)(*auto dest*: *colossless-gpvD*)

lemma *pfinite-gpv-sub-gpvs*:
assumes *pfinite-gpv* $\mathcal{I}$ *gpv gpv'* $\in$ *sub-gpvs* $\mathcal{I}$ *gpv* $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *pfinite-gpv* $\mathcal{I}$ *gpv'*
using *assms*(*2*,*1*,*3*) **by**(*induction*)(*auto dest*: *pfinite-gpv-ContD WT-gpvD*)

lemma *pfinite-gpv-id-oracle* [*simp*]: *pfinite-gpv* $\mathcal{I}$ (*id-oracle s x*) **if** $x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$
by(*simp add*: *id-oracle-def pgen-lossless-gpv-PauseI*[*OF that*])

2.1 *try-gpv*

lemma *plossless-gpv-try-gpvI*:
assumes *pfinite-gpv* $\mathcal{I}$ *gpv*
and $\neg$ *colossless-gpv* $\mathcal{I}$ *gpv* $\Longrightarrow$ *plossless-gpv* $\mathcal{I}$ *gpv'*
shows *plossless-gpv* $\mathcal{I}$ (*TRY gpv ELSE gpv'*)
using *assms* **unfolding** *pgen-lossless-gpv-def*
by(*cases colossless-gpv* $\mathcal{I}$ *gpv*)(*simp cong*: *expectation-gpv-cong-fail*, *simp*)

lemma *WT-gpv-try-gpvI* [*WT-intro*]:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$

```
    and ¬ colossless-gpv ℐ gpv ⟹ ℐ ⊢g gpv' √
  shows ℐ ⊢g try-gpv gpv gpv' √
   using assms by(coinduction arbitrary: gpv)(auto 4 4 dest: WT-gpvD coloss-
less-gpvD split: if-split-asm)

lemma (in callee-invariant-on) exec-gpv-try-gpv:
  fixes exec-gpv1
  defines exec-gpv1 ≡ exec-gpv
  assumes WT: ℐ ⊢g gpv √
    and pfinite: pfinite-gpv ℐ gpv
    and I: I s
    and f: ⋀s. I s ⟹ f (x, s) = z
    and lossless: ⋀s x. ⟦x ∈ outs-ℐ ℐ; I s⟧ ⟹ lossless-spmf (callee s x)
  shows map-spmf f (exec-gpv callee (try-gpv gpv (Done x)) s) =
    try-spmf (map-spmf f (exec-gpv1 callee gpv s)) (return-spmf z)
  (is ?lhs = ?rhs)
proof −
  note [[show-variants]]
  have le: ord-spmf (=) ?lhs ?rhs using WT I
  proof(induction arbitrary: gpv s rule: exec-gpv-fixp-induct)
    case adm show ?case by simp
    case bottom show ?case by simp
    case (step exec-gpv')
    show ?case using step.prems unfolding exec-gpv1-def
      apply(subst exec-gpv.simps)
      apply(simp add: map-spmf-bind-spmf)
      apply(subst (1 2) try-spmf-def)
      apply(simp add: map-bind-pmf bind-spmf-pmf-assoc o-def)
      apply(simp add: bind-spmf-def bind-map-pmf bind-assoc-pmf)
      apply(rule rel-pmf-bindI[where R=eq-onp (λx. x ∈ set-pmf (the-gpv gpv))])
       apply(rule pmf.rel-refl-strong)
       apply(simp add: eq-onp-def)
       apply(clarsimp split!: option.split generat.split simp add: bind-return-pmf f
map-spmf-bind-spmf o-def eq-onp-def)
      apply(simp add: bind-spmf-def bind-assoc-pmf)
      subgoal for out c
       apply(rule rel-pmf-bindI[where R=eq-onp (λx. x ∈ set-pmf (callee s out))])
         apply(rule pmf.rel-refl-strong)
         apply(simp add: eq-onp-def)
        apply(clarsimp split!: option.split simp add: eq-onp-def)
        apply(simp add: in-set-spmf[symmetric])
        apply(rule spmf.leq-trans)
         apply(rule step.IH)
          apply(frule (1) WT-gpvD)
          apply(erule (1) WT-gpvD)
          apply(drule WT-callee)
          apply(erule (2) WT-calleeD)
          apply(frule (1) WT-gpvD)
         apply(erule (2) callee-invariant)
```

apply(*simp add*: *try-spmf-def exec-gpv1-def*)
done
done
qed

have *lossless-spmf ?lhs*
apply *simp*
apply(*rule plossless-exec-gpv*)
apply(*rule plossless-gpv-try-gpvI*)
apply(*rule pfinite*)
apply *simp*
apply(*rule WT-gpv-try-gpvI*)
apply(*simp add*: *WT*)
apply *simp*
apply(*simp add*: *lossless*)
apply(*simp add*: *I*)
done
from *ord-spmf-lossless-spmfD1*[*OF le this*] **show** *?thesis* **by**(*simp add*: *spmf-rel-eq*)
qed

lemma *try-gpv-bind-gen-lossless′*: — generalises *gen-lossless-gpv ?b ℐ-full ?gpv* ⟹ *TRY ?gpv* ≫= *?f ELSE ?gpv′* = *?gpv* ≫= (λ*x*. *TRY ?f x ELSE ?gpv′*)
assumes *lossless*: *gen-lossless-gpv b ℐ gpv*
and *WT1*: ℐ ⊢*g gpv* √
and *WT2*: ℐ ⊢*g gpv′* √
and *WTf*: ⋀*x*. *x* ∈ *results-gpv ℐ gpv* ⟹ ℐ ⊢*g f x* √
shows *eq-ℐ-gpv* (=) ℐ (*TRY bind-gpv gpv f ELSE gpv′*) (*bind-gpv gpv* (λ*x*. *TRY f x ELSE gpv′*))
using *lossless WT1 WTf*
proof(*coinduction arbitrary*: *gpv*)
case (*eq-ℐ-gpv gpv*)
note [*simp*] = *spmf-rel-map generat.rel-map map-spmf-bind-spmf*
and [*intro*!] = *rel-spmf-reflI rel-generat-reflI rel-funI*
show *?case* **using** *gen-lossless-gpvD*[*OF eq-ℐ-gpv*(*1*)] *WT-gpvD*[*OF eq-ℐ-gpv*(*2*)] *WT-gpvD*[*OF WT2*] *WT-gpvD*[*OF eq-ℐ-gpv*(*3*)[*rule-format*, *OF results-gpv.Pure*]] *WT2*
apply(*auto simp del*: *bind-gpv-sel′ simp add*: *bind-gpv.sel try-spmf-bind-spmf-lossless generat.map-comp o-def intro*!: *rel-spmf-bind-reflI rel-spmf-try-spmf split*!: *generat.split*)
apply(*auto 4 4 intro*!: *eq-ℐ-gpv*(*3*)[*rule-format*] *eq-ℐ-gpv-reflI eq-ℐ-generat-reflI intro*: *results-gpv.IO WT-intro*)
done
qed

— We instantiate the parameter *b* such that it can be used as a conditional simp rule.
lemmas *try-gpv-bind-lossless′* = *try-gpv-bind-gen-lossless′*[**where** *b*=*False*]
and *try-gpv-bind-colossless′* = *try-gpv-bind-gen-lossless′*[**where** *b*=*True*]

lemma *try-gpv-bind-gpv*:

try-gpv (*bind-gpv gpv f*) *gpv'* =
bind-gpv (*try-gpv* (*map-gpv Some id gpv*) (*Done None*)) (*λx. case x of None* ⇒ *gpv'* | *Some x'* ⇒ *try-gpv* (*f x'*) *gpv'*)
by(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
(*auto simp add*: *rel-fun-def generat.rel-map bind-return-pmf spmf-rel-map map-bind-spmf o-def bind-gpv.sel bind-map-spmf try-spmf-def bind-spmf-def spmf.map-comp bind-map-pmf bind-assoc-pmf gpv.map-sel simp del*: *bind-gpv-sel' intro*!: *rel-pmf-bind-reflI generat.rel-refl-strong rel-spmf-reflI split*!: *option.split generat.split*)

lemma *bind-gpv-try-gpv-map-Some*:
bind-gpv (*try-gpv* (*map-gpv Some id gpv*) (*Done None*)) (*λx. case x of None* ⇒ *Fail* | *Some y* ⇒ *f y*) =
bind-gpv gpv f
by(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
(*auto simp add*: *bind-gpv.sel map-bind-spmf bind-map-spmf try-spmf-def bind-spmf-def spmf-rel-map bind-map-pmf gpv.map-sel bind-assoc-pmf bind-return-pmf generat.rel-map rel-fun-def simp del*: *bind-gpv-sel' intro*!: *rel-pmf-bind-reflI rel-spmf-reflI generat.rel-refl-strong split*!: *option.split generat.split*)

lemma *try-gpv-left-gpv*:
assumes $\mathcal{I} \vdash g$ *gpv* √ **and** *WT2*: $\mathcal{I} \vdash g$ *gpv'* √
shows *eq-$\mathcal{I}$-gpv* (=) ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*try-gpv* (*left-gpv gpv*) (*left-gpv gpv'*)) (*left-gpv* (*try-gpv gpv gpv'*))
using *assms*(*1*)
apply(*coinduction arbitrary*: *gpv*)
apply(*auto simp add*: *map-try-spmf spmf.map-comp o-def generat.map-comp spmf-rel-map intro*!: *rel-spmf-try-spmf rel-spmf-reflI*)
subgoal for *gpv generat* **by**(*cases generat*)(*auto dest*: *WT-gpvD*)
subgoal for *gpv generat* **using** *WT2*
by(*cases generat*)(*auto 4 4 dest*: *WT-gpvD intro*!: *eq-$\mathcal{I}$-gpv-reflI WT-gpv-left-gpv*)
done

lemma *try-gpv-right-gpv*:
assumes $\mathcal{I}' \vdash g$ *gpv* √ **and** *WT2*: $\mathcal{I}' \vdash g$ *gpv'* √
shows *eq-$\mathcal{I}$-gpv* (=) ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*try-gpv* (*right-gpv gpv*) (*right-gpv gpv'*)) (*right-gpv* (*try-gpv gpv gpv'*))
using *assms*(*1*)
apply(*coinduction arbitrary*: *gpv*)
apply(*auto simp add*: *map-try-spmf spmf.map-comp o-def generat.map-comp spmf-rel-map intro*!: *rel-spmf-try-spmf rel-spmf-reflI*)
subgoal for *gpv generat* **by**(*cases generat*)(*auto dest*: *WT-gpvD*)
subgoal for *gpv generat* **using** *WT2*
by(*cases generat*)(*auto 4 4 dest*: *WT-gpvD intro*!: *eq-$\mathcal{I}$-gpv-reflI WT-gpv-right-gpv*)
done

lemma *bind-try-Done-Fail*: *bind-gpv* (*TRY gpv ELSE Done x*) *f* = *bind-gpv gpv f*
if *f x* = *Fail*
apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
apply(*auto simp del*: *bind-gpv-sel' simp add*: *bind-gpv.sel map-bind-spmf bind-map-spmf*

try-spmf-def bind-spmf-def map-bind-pmf bind-assoc-pmf bind-map-pmf bind-return-pmf spmf.map-comp o-def that rel-fun-def intro!: *rel-pmf-bind-reflI rel-spmf-reflI generat.rel-refl-strong split*!: *option.split generat.split*)
done

lemma *inline-map-gpv'*:
inline callee (*map-gpv' f g h gpv*) *s* =
map-gpv (*apfst f*) *id* (*inline* (*map-fun id* (*map-fun g* (*map-gpv* (*apfst h*) *id*)) *callee*) *gpv s*)
using *inline-parametric'*[**where** *S*=(=) **and** *C*=*BNF-Def.Grp UNIV g* **and** *R*=*conversep* (*BNF-Def.Grp UNIV h*) **and** *A*=*BNF-Def.Grp UNIV f* **and** *C'*=(=) **and** *R'*=(=)]
apply(*subst* (*asm*) (*2 3 8*) *eq-alt-conversep*)
apply(*subst* (*asm*) (*1 3 4 5*) *eq-alt*)
apply(*subst* (*asm*) (*1*) *eq-alt-conversep2*)
apply(*unfold prod.rel-conversep rel-gpv''-conversep prod.rel-Grp rel-gpv''-Grp*)
apply(*force simp add*: *rel-fun-def Grp-def map-gpv-conv-map-gpv' map-fun-def*[*abs-def*] *o-def apfst-def*)
done

lemma *interaction-bound-try-gpv*:
fixes *consider* **defines** *ib* ≡ *interaction-bound consider*
shows *interaction-bound consider* (*try-gpv gpv gpv'*) ≤ *ib gpv* + *ib gpv'*
proof(*induction arbitrary*: *gpv gpv' rule*: *interaction-bound-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step interaction-bound'*)
show *?case* **unfolding** *ib-def*
apply(*clarsimp simp add*: *generat.map-comp image-image o-def case-map-generat cong del*: *generat.case-cong split*!: *if-split generat.split intro*!: *SUP-least*)
subgoal
apply(*subst interaction-bound.simps*)
apply *simp*
apply(*subst Sup-image-eadd1*[*symmetric*])
apply *clarsimp*
apply(*rule SUP-upper2*)
apply(*rule rev-image-eqI*)
apply *simp*
apply *simp*
apply(*simp add*: *iadd-Suc*)
apply(*subst Sup-image-eadd1*[*symmetric*])
apply *simp*
apply(*rule SUP-mono*)
apply *simp*
apply(*rule exI*)
apply(*rule step.IH*[*unfolded ib-def*])
done
subgoal

```
  apply(subst interaction-bound.simps)
  apply simp
  apply(subst Sup-image-eadd1[symmetric])
   apply clarsimp
  apply(rule SUP-upper2)
   apply(rule rev-image-eqI)
    apply simp
   apply simp
  apply(subst Sup-image-eadd1[symmetric])
   apply simp
  apply(rule SUP-upper2)
   apply(rule rev-image-eqI)
    apply simp
   apply simp
  apply(rule step.IH[unfolded ib-def])
  done
subgoal
  apply(subst interaction-bound.simps)
  apply simp
  apply(subst Sup-image-eadd1[symmetric])
   apply clarsimp
  apply(rule SUP-upper2)
   apply(rule rev-image-eqI)
    apply simp
   apply simp
  apply(simp add: iadd-Suc)
  apply(subst Sup-image-eadd1[symmetric])
   apply simp
  apply(rule SUP-mono)
  apply simp
  apply(rule exI)
  apply(rule step.IH[unfolded ib-def])
  done
subgoal
  apply(subst interaction-bound.simps)
  apply simp
  apply(subst Sup-image-eadd1[symmetric])
   apply clarsimp
  apply(rule SUP-upper2)
   apply(rule rev-image-eqI)
    apply simp
   apply simp
  apply(subst Sup-image-eadd1[symmetric])
   apply simp
  apply(rule SUP-upper2)
   apply(rule rev-image-eqI)
    apply simp
   apply simp
  apply(rule step.IH[unfolded ib-def])
```

done
subgoal
apply(*subst* (*2*) *interaction-bound.simps*)
apply *simp*
apply(*subst Sup-image-eadd2*[*symmetric*])
apply *clarsimp*
apply *simp*
apply(*rule SUP-upper2*)
apply(*rule rev-image-eqI*)
apply *simp*
apply *simp*
apply(*simp add*: *iadd-Suc-right*)
apply(*subst Sup-image-eadd2*[*symmetric*])
apply *clarsimp*
apply(*rule SUP-mono*)
apply *clarsimp*
apply(*rule exI*)
apply(*rule order-trans*)
apply(*rule step.hyps*)
apply(*rule enat-le-plus-same*)
done
subgoal
apply(*subst* (*2*) *interaction-bound.simps*)
apply *simp*
apply(*subst Sup-image-eadd2*[*symmetric*])
apply *clarsimp*
apply *simp*
apply(*rule SUP-upper2*)
apply(*rule rev-image-eqI*)
apply *simp*
apply *simp*
apply(*subst Sup-image-eadd2*[*symmetric*])
apply *clarsimp*
apply(*rule SUP-upper2*)
apply(*rule imageI*)
apply *simp*
apply(*rule order-trans*)
apply(*rule step.hyps*)
apply(*rule enat-le-plus-same*)
done
done
qed

lemma *interaction-bounded-by-try-gpv* [*interaction-bound*]:
interaction-bounded-by consider (*try-gpv gpv1 gpv2*) (*bound1* + *bound2*)
if *interaction-bounded-by consider gpv1 bound1 interaction-bounded-by consider gpv2 bound2*
using *that interaction-bound-try-gpv*[*of consider gpv1 gpv2*]
by(*simp add*: *interaction-bounded-by.simps*)(*meson add-left-mono-trans add-right-mono*

le-left-mono)

2.2 term *gpv-stop*

lemma *interaction-bounded-by-gpv-stop* [*interaction-bound*]:
  assumes *interaction-bounded-by consider gpv n*
  shows *interaction-bounded-by consider* (*gpv-stop gpv*) *n*
  using *assms* **by**(*simp add*: *interaction-bounded-by.simps*)

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

lift-definition *stop-ℐ* :: (*'a*, *'b*) *ℐ* ⇒ (*'a*, *'b option*) *ℐ* **is**
  λresp x. if (*resp x* = {}) *then* {} *else insert None* (*Some* ‘ *resp x*) **.**

lemma *outs-stop-ℐ* [*simp*]: *outs-ℐ* (*stop-ℐ ℐ*) = *outs-ℐ ℐ*
  by *transfer auto*

lemma *responses-stop-ℐ* [*simp*]:
  responses-ℐ (*stop-ℐ ℐ*) *x* = (*if x* ∈ *outs-ℐ ℐ then insert None* (*Some* ‘ *responses-ℐ ℐ x*) *else* {})
  by *transfer auto*

lemma *stop-ℐ-full* [*simp*]: *stop-ℐ ℐ-full* = *ℐ-full*
  by *transfer*(*auto simp add*: *fun-eq-iff notin-range-Some*)

lemma *stop-ℐ-uniform* [*simp*]:
  stop-ℐ (*ℐ-uniform A B*) = (*if B* = {} *then* ⊥ *else ℐ-uniform A* (*insert None* (*Some* ‘ *B*)))
  unfolding *bot-ℐ-def* **by** *transfer*(*simp add*: *fun-eq-iff*)

lifting-update *ℐ.lifting*
lifting-forget *ℐ.lifting*

end

lemma *stop-ℐ-bot* [*simp*]: *stop-ℐ* ⊥ = ⊥
  by(*simp only*: *bot-ℐ-def stop-ℐ-uniform*)(*simp*)

lemma *WT-gpv-stop* [*simp*, *WT-intro*]: *stop-ℐ ℐ* ⊢*g gpv-stop gpv* √ **if** *ℐ* ⊢*g gpv* √
  using *that* **by**(*coinduction arbitrary*: *gpv*)(*auto 4 3 dest*: *WT-gpvD*)

lemma *expectation-gpv-stop*:
  fixes *fail* **and** *gpv* :: (*'a*, *'b*, *'c*) *gpv*
  assumes *WT*: *ℐ* ⊢*g gpv* √
  and *fail*: *fail* ≤ *c*
  shows *expectation-gpv fail* (*stop-ℐ ℐ*) (*λ-. c*) (*gpv-stop gpv*) = *expectation-gpv fail ℐ* (*λ-. c*) *gpv* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *expectation-gpv fail* (*stop-ℐ ℐ*) (*λ-. c*) (*gpv-stop gpv*) ≤ *expectation-gpv fail*

I (λ-. *c*) *gpv*
using *WT*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step f g*)
then show *?case*
apply(*simp add*: *pmf-map-spmf-None measure-spmf-return-spmf nn-integral-return*)
apply(*rule disjI2 nn-integral-mono-AE*)+
apply(*auto split*!: *generat.split simp add*: *image-image dest*: *WT-gpvD intro*!: *le-infI2 INF-mono*)
done
qed

define *stop* :: (*'a option*, *'b*, *'c option*) *gpv* ⇒ - **where** *stop* = *expectation-gpv fail* (*stop-I I*) (λ-. *c*)
show *?rhs* ≤ *stop* (*gpv-stop gpv*) **using** *WT*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv'*)
have *expectation-gpv' gpv'* ≤ *c* **if** *I* ⊢*g gpv'* √ **for** *gpv'*
using *expectation-gpv-const-le*[*of I gpv' fail c*] *fail step.hyps*(*1*)[*of gpv'*] *that*
by(*simp add*: *max-def split*: *if-split-asm*)
then show *?case* **using** *step* **unfolding** *stop-def*
apply(*subst expectation-gpv.simps*)
apply(*simp add*: *pmf-map-spmf-None*)
apply(*rule disjI2 nn-integral-mono-AE*)+
apply(*clarsimp split*!: *generat.split simp add*: *image-image*)
subgoal by(*auto 4 3 simp add*: *in-outs-I-iff-responses-I dest*: *WT-gpv-ContD intro*: *INF-lower2*)
subgoal by(*auto intro*!: *INF-mono rev-bexI dest*: *WT-gpvD*)
done
qed
qed

lemma *pgen-lossless-gpv-stop*:
fixes *fail* **and** *gpv* :: (*'a*, *'b*, *'c*) *gpv*
assumes *WT*: *I* ⊢*g gpv* √
and *fail*: *fail* ≤ *1*
shows *pgen-lossless-gpv fail* (*stop-I I*) (*gpv-stop gpv*) = *pgen-lossless-gpv fail I gpv*
by(*simp add*: *pgen-lossless-gpv-def expectation-gpv-stop assms*)

lemma *pfinite-gpv-stop* [*simp*]:
pfinite-gpv (*stop-I I*) (*gpv-stop gpv*) ⟷ *pfinite-gpv I gpv* **if** *I* ⊢*g gpv* √
using *that* **by**(*simp add*: *pgen-lossless-gpv-stop*)

lemma *plossless-gpv-stop* [*simp*]:
plossless-gpv (*stop-$\mathcal{I}$ $\mathcal{I}$*) (*gpv-stop gpv*) $\longleftrightarrow$ *plossless-gpv $\mathcal{I}$ gpv* **if** $\mathcal{I}$ $\vdash$*g gpv* $\surd$
using *that* **by**(*simp add*: *pgen-lossless-gpv-stop*)

lemma *results-gpv-stop-SomeD*: *Some x* $\in$ *results-gpv* (*stop-$\mathcal{I}$ $\mathcal{I}$*) (*gpv-stop gpv*) $\Longrightarrow$ *x* $\in$ *results-gpv $\mathcal{I}$ gpv*
by(*induction gpv'*$\equiv$*gpv-stop gpv arbitrary*: *gpv rule*: *results-gpv.induct*)
(*auto 4 3 intro*: *results-gpv.intros split*: *if-split-asm*)

lemma *Some-in-results'-gpv-gpv-stopD*: *Some xy* $\in$ *results'-gpv* (*gpv-stop gpv*) $\Longrightarrow$ *xy* $\in$ *results'-gpv gpv*
unfolding *results-gpv-$\mathcal{I}$-full*[*symmetric*]
by(*rule results-gpv-stop-SomeD*) *simp*

2.3 term *exception-$\mathcal{I}$*

datatype *'s exception* = *Fault* | *OK* (*ok*: *'s*)

lemma *inj-on-OK* [*simp*]: *inj-on OK A*
by(*auto simp add*: *inj-on-def*)

function *join-exception* :: *'a exception* $\Rightarrow$ *'b exception* $\Rightarrow$ (*'a* $\times$ *'b*) *exception* **where**
join-exception Fault - = Fault
| *join-exception - Fault = Fault*
| *join-exception* (*OK a*) (*OK b*) = *OK* (*a*, *b*)
by *pat-completeness auto*
termination by *lexicographic-order*

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

fun *option-of-exception* :: *'a exception* $\Rightarrow$ *'a option* **where**
option-of-exception Fault = None
| *option-of-exception* (*OK x*) = *Some x*

fun *exception-of-option* :: *'a option* $\Rightarrow$ *'a exception* **where**
exception-of-option None = Fault
| *exception-of-option* (*Some x*) = *OK x*

lemma *option-of-exception-exception-of-option* [*simp*]: *option-of-exception* (*exception-of-option x*) = *x*
by(*cases x*) *simp-all*

lemma *exception-of-option-option-of-exception* [*simp*]: *exception-of-option* (*option-of-exception x*) = *x*

by(*cases x*) *simp-all*

lemma *case-exception-of-option* [*simp*]: *case-exception f g* (*exception-of-option x*) = *case-option f g x*
by(*simp split*: *exception.split option.split*)

lemma *case-option-of-exception* [*simp*]: *case-option f g* (*option-of-exception x*) = *case-exception f g x*
by(*simp split*: *exception.split option.split*)

lemma *surj-exception-of-option* [*simp*]: *surj exception-of-option*
by(*rule surjI*[**where** *f*=*option-of-exception*])(*simp*)

lemma *surj-option-of-exception* [*simp*]: *surj option-of-exception*
by(*rule surjI*[**where** *f*=*exception-of-option*])(*simp*)

lemma *case-map-exception* [*simp*]: *case-exception f g* (*map-exception h x*) = *case-exception f* (*g* ∘ *h*) *x*
by(*simp split*: *exception.split*)

definition *exception-ℐ* :: (′*a*, ′*b*) ℐ ⇒ (′*a*, ′*b exception*) ℐ **where**
exception-ℐ ℐ = *map-ℐ id exception-of-option* (*stop-ℐ ℐ*)

lemma *outs-exception-ℐ* [*simp*]: *outs-ℐ* (*exception-ℐ ℐ*) = *outs-ℐ ℐ*
by(*simp add*: *exception-ℐ-def*)

lemma *responses-exception-ℐ* [*simp*]:
responses-ℐ (*exception-ℐ ℐ*) *x* = (*if x* ∈ *outs-ℐ ℐ then insert Fault* (*OK* ‘ *responses-ℐ ℐ x*) *else* {})
by(*simp add*: *exception-ℐ-def image-image*)

lemma *map-ℐ-full* [*simp*]: *map-ℐ f g ℐ-full* = *ℐ-uniform UNIV* (*range g*)
unfolding *ℐ-uniform-UNIV*[*symmetric*] *map-ℐ-ℐ-uniform* **by** *simp*

lemma *exception-ℐ-full* [*simp*]: *exception-ℐ ℐ-full* = *ℐ-full*
unfolding *exception-ℐ-def* **by** *simp*

lemma *exception-ℐ-uniform* [*simp*]:
exception-ℐ (*ℐ-uniform A B*) = (*if B* = {} *then* ⊥ *else ℐ-uniform A* (*insert Fault* (*OK* ‘ *B*)))
by(*simp add*: *exception-ℐ-def image-image*)

lemma *option-of-exception-ℐ* [*simp*]: *map-ℐ id option-of-exception* (*exception-ℐ ℐ*) = *stop-ℐ ℐ*
by(*simp add*: *exception-ℐ-def o-def id-def*[*symmetric*])

lemma *exception-of-option-ℐ* [*simp*]: *map-ℐ id exception-of-option* (*stop-ℐ ℐ*) = *exception-ℐ ℐ*
by(*simp add*: *exception-ℐ-def*)

2.4 inline

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

context
fixes *gpv* :: (*'a*, *'call*, *'ret*) *gpv*
assumes *gpv*: *plossless-gpv* $\mathcal{I}$ *gpv* $\mathcal{I}$ ⊢*g gpv* √
begin

lemma *lossless-spmf-inline1*:
assumes *lossless*: ⋀*s x*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *lossless-spmf* (*the-gpv* (*callee s x*))
and *I*: *I s*
shows *lossless-spmf* (*inline1 callee gpv s*)
proof −
have *1* = *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *gpv* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have ... ≤ *weight-spmf* (*inline1 callee gpv s*) **using** *gpv*(*2*) *I*
proof(*induction arbitrary*: *gpv s rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv'*)
{ **fix** *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
with *step.prems* **have** *out*: *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*auto dest*: *WT-gpvD*)
from *out*[*unfolded in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$] **obtain** *input* **where** *input*: *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* **by** *auto*
from *out* **have** (∏ *r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) = ∫ ⁺ *x*. (∏ *r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) ∂*measure-spmf* (*the-gpv* (*callee s out*))
using *lossless* ‹*I s*› **by**(*simp add*: *lossless-spmf-def measure-spmf.emeasure-eq-measure*)
also have ... ≤ ∫ ⁺ *generat*. (*case generat of Pure* (*r*, *s'*) ⇒ *weight-spmf* (*inline1 callee* (*c r*) *s'*) | - ⇒ *1*) ∂*measure-spmf* (*the-gpv* (*callee s out*))
apply(*intro nn-integral-mono-AE*)
apply(*clarsimp split*!: *generat.split*)
subgoal *Pure*
apply(*rule INF-lower2*)
apply(*fastforce dest*: *results-callee*[*OF out* ‹*I s*›, *THEN subsetD*, *OF results-gpv.Pure*])
apply(*rule step.IH*)
apply(*fastforce intro*: *WT-gpvD*[*OF step.prems*(*1*) *IO*] *dest*: *results-callee*[*OF out* ‹*I s*›, *THEN subsetD*, *OF results-gpv.Pure*])
apply(*fastforce dest*: *results-callee*[*OF out* ‹*I s*›, *THEN subsetD*, *OF results-gpv.Pure*])
done
subgoal *IO*
apply(*rule INF-lower2*[*OF input*])
apply(*rule order-trans*)
apply(*rule step.hyps*)
apply(*rule order-trans*)
apply(*rule expectation-gpv-const-le*)

```
          apply(rule WT-gpvD[OF step.prems(1) IO])
          apply(simp-all add: input)
         done
       done
      finally have (∏ r∈responses-ℐ ℐ out. expectation-gpv′ (c r)) ≤ ... .}
    then show ?case using step.prems
      apply(subst inline1.simps)
      apply(simp add: measure-spmf.emeasure-eq-measure[symmetric])
      apply(simp add: measure-spmf-bind)
      apply(subst emeasure-bind[where N=count-space UNIV])
         apply(simp add: space-measure-spmf)
        apply(simp add: o-def)
       apply(simp)
      apply(rule nn-integral-mono-AE)
         apply(clarsimp split!: generat.split simp add: measure-spmf-return-spmf
space-measure-spmf)
      apply(simp add: measure-spmf-bind)
      apply(subst emeasure-bind[where N=count-space UNIV])
         apply(simp add: space-measure-spmf)
        apply(simp add: o-def)
       apply(simp)
      apply (simp add: measure-spmf.emeasure-eq-measure)
      apply(subst generat.case-distrib[where h=λx. measure (measure-spmf x) -])
     apply(simp add: split-def measure-spmf-return-spmf space-measure-spmf mea-
sure-return cong del: generat.case-cong)
      done
  qed
  finally show ?thesis using weight-spmf-le-1[of inline1 callee gpv s] by(simp add:
lossless-spmf-def)
qed

end

end

lemma (in raw-converter-invariant) inline1-try-gpv:
  defines inline1′ ≡ inline1
  assumes WT: ℐ ⊢g gpv √
    and pfinite: pfinite-gpv ℐ gpv
    and f: ⋀s. I s ⟹ f (x, s) = z
    and lossless: ⋀s x. ⟦ x ∈ outs-ℐ ℐ; I s ⟧ ⟹ colossless-gpv ℐ′ (callee s x)
    and I: I s
  shows map-spmf (map-sum f id) (inline1 callee (try-gpv gpv (Done x)) s) =
   try-spmf (map-spmf (map-sum f (λ(out, c, rpv). (out, c, λinput. try-gpv (rpv
input) (Done x)))) (inline1′ callee gpv s)) (return-spmf (Inl z))
  (is ?lhs = ?rhs)
proof –
  have le: ord-spmf (=) ?lhs ?rhs using WT I
  proof(induction arbitrary: gpv s rule: inline1-fixp-induct)
```

```
    case adm show ?case by simp
    case bottom show ?case by simp
    case (step inline1'')
    show ?case using step.prems unfolding inline1'-def
      apply(subst inline1.simps)
      apply(simp add: bind-map-spmf map-bind-spmf o-def)
      apply(simp add: try-spmf-def)
      apply(subst bind-spmf-pmf-assoc)
      apply(simp add: bind-map-pmf)
      apply(subst (3) bind-spmf-def)
      apply(simp add: bind-assoc-pmf)
      apply(rule rel-pmf-bindI[where R=eq-onp (λx. x ∈ set-pmf (the-gpv gpv))])
       apply(rule pmf.rel-refl-strong)
       apply(simp add: eq-onp-def)
        apply(clarsimp simp add: eq-onp-def bind-return-pmf f split!: option.split
generat.split)
      subgoal for out c
        apply(simp add: in-set-spmf[symmetric] bind-map-pmf map-bind-spmf)
        apply(subst option.case-distrib[where h=return-pmf, symmetric, abs-def])
        apply(fold map-pmf-def)
        apply(simp add: bind-spmf-def map-bind-pmf)
       apply(rule rel-pmf-bindI[where R=eq-onp (λx. x ∈ set-pmf (the-gpv (callee
s out)))])
         apply(rule pmf.rel-refl-strong)
         apply(simp add: eq-onp-def)
           apply(simp add: in-set-spmf[symmetric] bind-map-pmf map-bind-spmf
eq-onp-def split!: option.split generat.split)
        apply(rule spmf.leq-trans)
         apply(rule step.IH[unfolded inline1'-def])
        subgoal
         by(auto dest: results-callee[THEN subsetD, OF - - results-gpv.Pure, rotated
−1] WT-gpvD)
        subgoal
         by(auto dest: results-callee[THEN subsetD, OF - - results-gpv.Pure, rotated
−1] WT-gpvD)
        apply(simp add: try-spmf-def)
        apply(subst option.case-distrib[where h=return-pmf, symmetric, abs-def])
        apply(fold map-pmf-def)
        apply simp
        done
      done
  qed
  have lossless-spmf ?lhs using I
    apply simp
    apply(rule lossless-spmf-inline1)
       apply(rule plossless-gpv-try-gpvI)
        apply(rule pfinite)
       apply simp
      apply(rule WT-gpv-try-gpvI)
```

apply(*rule WT*)
apply *simp*
apply(*rule colossless-gpv-lossless-spmfD*[*OF lossless*])
apply *simp-all*
done
from *ord-spmf-lossless-spmfD1*[*OF le this*] **show** *?thesis* **by**(*simp add*: *spmf-rel-eq*)
qed

lemma (**in** *raw-converter-invariant*) *inline-try-gpv*:
assumes *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *pfinite*: *pfinite-gpv* $\mathcal{I}$ *gpv*
and *f*: $\bigwedge s.\ I\ s \Longrightarrow f\ (x,\ s) = z$
and *lossless*: $\bigwedge s\ x.\ [\![\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; $I\ s\ ]\!] \Longrightarrow$ *colossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *eq-*$\mathcal{I}$*-gpv* (=) $\mathcal{I}'$ (*map-gpv f id* (*inline callee* (*try-gpv gpv* (*Done x*)) *s*))
(*try-gpv* (*map-gpv f id* (*inline callee gpv s*)) (*Done z*))
(**is** *eq-*$\mathcal{I}$*-gpv - - ?lhs ?rhs*)
using *WT pfinite I*
proof(*coinduction arbitrary*: *gpv s rule*: *eq-*$\mathcal{I}$*-gpv-coinduct-bind*)
case (*eq-*$\mathcal{I}$*-gpv gpv s*)
show *?case TYPE*((*'ret* × *'s*) *option*) *TYPE*((*'ret* × *'s*) *option*) (**is** *rel-spmf*
(*eq-*$\mathcal{I}$*-generat - - ?X*) *?lhs ?rhs*)
proof −
have *?lhs* = *map-spmf*
(λ*x. case x of Inl rs* ⇒ *Pure rs* | *Inr* (*out*, *oracle*, *rpv*) ⇒ *IO out* (λ*input.*
map-gpv f id (*bind-gpv* (*try-gpv* (*map-gpv Some id* (*oracle input*)) (*Done*
None)) (λ*xy. case xy of None* ⇒ *Fail* | *Some* (*x*, *y*) ⇒ *inline callee* (*rpv x*) *y*))))
(*map-spmf* (*map-sum f id*) (*inline1 callee* (*TRY gpv ELSE Done x*) *s*))
(**is** - = *map-spmf ?f ?lhs2*)
by(*auto simp add*: *gpv.map-sel inline-sel spmf.map-comp o-def bind-gpv-try-gpv-map-Some*
intro!: *map-spmf-cong*[*OF refl*] *split*: *sum.split*)
also from *eq-*$\mathcal{I}$*-gpv*
have *?lhs2* = *TRY map-spmf* (*map-sum f* (λ(*out*, *c*, *rpv*). (*out*, *c*, λ*input. TRY*
rpv input ELSE Done x))) (*inline1 callee gpv s*) *ELSE return-spmf* (*Inl z*)
by(*intro inline1-try-gpv*)(*auto intro*: *f lossless*)
also have . . . = *map-spmf* (λ*y. case y of None* ⇒ *Inl z* | *Some x'* ⇒ *map-sum*
f (λ(*out*, *c*, *rpv*). (*out*, *c*, λ*input. try-gpv* (*rpv input*) (*Done x*))) *x'*)
(*try-spmf* (*map-spmf Some* (*inline1 callee gpv s*)) (*return-spmf None*))
(**is** - = *?lhs3*) **by**(*simp add*: *map-try-spmf spmf.map-comp o-def*)
also have *?rhs* = *map-spmf* (λ*y. case y of None* ⇒ *Pure z* | *Some* (*Inl x*) ⇒
Pure (*f x*)
| *Some* (*Inr* (*out*, *oracle*, *rpv*)) ⇒ *IO out* (λ*input. try-gpv* (*map-gpv f id*
(*bind-gpv* (*oracle input*) (λ(*x*, *y*). *inline callee* (*rpv x*) *y*))) (*Done z*)))
(*try-spmf* (*map-spmf Some* (*inline1 callee gpv s*)) (*return-spmf None*))
by(*auto simp add*: *gpv.map-sel inline-sel spmf.map-comp o-def generat.map-comp*
spmf-rel-map map-try-spmf intro!: *try-spmf-cong map-spmf-cong split*: *sum.split*)
moreover have *rel-spmf* (*eq-*$\mathcal{I}$*-generat* (=) $\mathcal{I}'$ *?X*) (*map-spmf ?f ?lhs3*) . . .
apply(*clarsimp simp add*: *gpv.map-sel inline-sel spmf.map-comp o-def gen-*
erat.map-comp spmf-rel-map intro!: *rel-spmf-reflI*)

apply(*erule disjE*)
subgoal
apply(*clarsimp split*!: *generat.split sum.split simp add*: *map-gpv-id-bind-gpv*)
apply(*subst* (*3*) *try-gpv-bind-gpv*)
apply(*rule conjI*)
apply(*erule WT-gpv-inline1*[*OF - eq-I-gpv*(*1*,*3*)])
apply(*rule strip*)+
apply(*rule disjI2*)+
subgoal for *out rpv rpv′ input*
apply(*rule exI*)
apply(*rule exI*)
apply(*rule exI*[**where** *x*=λ*x y*. *x* = *y* ∧ *y* ∈ *results-gpv I′* (*TRY map-gpv Some id* (*rpv input*) *ELSE Done None*)])
apply(*rule exI conjI refl*)+
apply(*rule eq-I-gpv-reflI*)
apply(*simp add*: *eq-onp-def*)
apply(*rule WT-intro*)
apply *simp*
apply(*erule* (*1*) *WT-gpv-inline1*[*OF - eq-I-gpv*(*1*,*3*)])
apply *simp*
apply(*rule rel-funI*)
apply(*clarsimp simp add*: *eq-onp-def split*: *if-split-asm*)
subgoal
apply(*rule exI conjI refl*)+
apply(*drule* (*2*) *WT-gpv-inline1*(*3*)[*OF - eq-I-gpv*(*1*,*3*)])
apply *simp*
apply(*frule* (*2*) *WT-gpv-inline1*(*3*)[*OF - eq-I-gpv*(*1*,*3*)])
apply(*drule* (*2*) *inline1-in-sub-gpvs*[*OF - - - eq-I-gpv*(*1*,*3*)])
apply *clarsimp*
apply(*erule pfinite-gpv-sub-gpvs*[*OF eq-I-gpv*(*2*) *- eq-I-gpv*(*1*)])
done
subgoal
apply(*erule disjE*; *clarsimp*)
apply(*rule exI conjI refl*)+
apply(*drule* (*2*) *WT-gpv-inline1*(*3*)[*OF - eq-I-gpv*(*1*,*3*)])
apply *simp*
apply(*frule* (*2*) *WT-gpv-inline1*(*3*)[*OF - eq-I-gpv*(*1*,*3*)])
apply(*drule* (*2*) *inline1-in-sub-gpvs*[*OF - - - eq-I-gpv*(*1*,*3*)])
apply *clarsimp*
apply(*erule pfinite-gpv-sub-gpvs*[*OF eq-I-gpv*(*2*) *- eq-I-gpv*(*1*)])
apply(*erule notE*)
apply(*drule inline1-in-sub-gpvs-callee*[*OF - eq-I-gpv*(*1*,*3*)])
apply *clarify*
apply(*drule* (*1*) *bspec*)
apply(*erule colossless-gpv-sub-gpvs*[*rotated*])
apply(*rule lossless*; *simp*)
done
done
done

subgoal by(*clarsimp split*: *if-split-asm*)
done
ultimately show *?thesis* **by**(*simp only*:)
qed
qed

definition *cr-prod2* :: $'a \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'b \Rightarrow 'a \times 'c \Rightarrow bool$ **where**
cr-prod2 x A = (λb (*a*, *c*). *A b c* $\wedge$ *x* = *a*)

lemma *cr-prod2-simps* [*simp*]: *cr-prod2 x A a* (*b*, *c*) $\longleftrightarrow$ *A a c* $\wedge$ *x* = *b*
by(*simp add*: *cr-prod2-def*)

lemma *cr-prod2I*: *A a b* $\Longrightarrow$ *cr-prod2 x A a* (*x*, *b*) **by** *simp*

lemma *cr-prod2-Grp*: *cr-prod2 x* (*BNF-Def.Grp A f*) = *BNF-Def.Grp A* (λb. (*x*, *f b*))
by(*auto simp add*: *Grp-def fun-eq-iff*)

lemma *extend-state-oracle-transfer'*: **includes** *lifting-syntax* **shows**
((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) ===> *cr-prod2 s S* ===> *C* ===> *rel-spmf* (*rel-prod R* (*cr-prod2 s S*))) (λ*oracle*. *oracle*) *extend-state-oracle*
unfolding *extend-state-oracle-def*[*abs-def*]
apply(*rule rel-funI*)+
apply *clarsimp*
apply(*drule* (*1*) *rel-funD*)+
apply(*auto simp add*: *spmf-rel-map split-def dest*: *rel-funD intro*: *rel-spmf-mono*)
done

lemma *exec-gpv-extend-state-oracle*:
exec-gpv (*extend-state-oracle callee*) *gpv* (*s*, *s'*) =
map-spmf (λ(*x*, *s''*). (*x*, (*s*, *s''*))) (*exec-gpv callee gpv s'*)
using *exec-gpv-parametric'*[*THEN rel-funD*, *OF extend-state-oracle-transfer'*[*THEN rel-funD*], *of* (=) (=) (=) *callee callee* (=) *s*]
unfolding *relator-eq rel-gpv''-eq*
apply(*clarsimp simp add*: *rel-fun-def*)
apply(*unfold eq-alt cr-prod2-Grp prod.rel-Grp option.rel-Grp pmf.rel-Grp*)
apply(*simp add*: *Grp-def map-prod-def*)
apply(*blast intro*: *sym*)
done

3 Material for Constructive Crypto

lemma *WT-resource-$\mathcal{I}$-uniform-UNIV* [*simp*]: *$\mathcal{I}$-uniform A UNIV* $\vdash$*res res* $\surd$
by(*coinduction arbitrary*: *res*) *auto*

lemma *WT-converter-of-callee-invar*:
assumes *WT*: $\bigwedge s\ q.\ \llbracket\ q \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ \rrbracket \Longrightarrow \mathcal{I}' \vdash g\ callee\ s\ q\ \surd$
and *res*: $\bigwedge s\ q\ r\ s'.\ \llbracket\ (r,\ s') \in results\text{-}gpv\ \mathcal{I}'\ (callee\ s\ q);\ q \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ \ \rrbracket \Longrightarrow r \in responses\text{-}\mathcal{I}\ \mathcal{I}\ q \wedge I\ s'$
and *I*: *I s*
shows $\mathcal{I},\ \mathcal{I}' \vdash_C\ converter\text{-}of\text{-}callee\ callee\ s\ \surd$
using *I* **by**(*coinduction arbitrary*: *s*)(*auto simp add*: *WT res*)

lemma *eq-$\mathcal{I}$-gpv-eq-OO*:
assumes *eq-$\mathcal{I}$-gpv* (=) $\mathcal{I}$ *gpv gpv′ eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv′ gpv″*
shows *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv″*
using *eq-$\mathcal{I}$-gpv-relcompp*[*THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*, *OF relcomppI*, *OF assms*]
by(*simp add*: *eq-OO*)

lemma *eq-$\mathcal{I}$-gpv-eq-OO2*:
assumes *eq-$\mathcal{I}$-gpv* (=) $\mathcal{I}$ *gpv″ gpv′ eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv′*
shows *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv gpv″*
using *eq-$\mathcal{I}$-gpv-relcompp*[**where** *A′*=*conversep* (=), *THEN fun-cong*, *THEN fun-cong*, *THEN iffD2*, *OF relcomppI*, *OF assms*(*2*)] *assms*(*1*)
unfolding *eq-$\mathcal{I}$-gpv-conversep* **by**(*simp add*: *OO-eq*)

lemma *eq-$\mathcal{I}$-gpv-try-gpv-cong*:
assumes *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv1 gpv1′*
and *eq-$\mathcal{I}$-gpv A $\mathcal{I}$ gpv2 gpv2′*
shows *eq-$\mathcal{I}$-gpv A $\mathcal{I}$* (*try-gpv gpv1 gpv2*) (*try-gpv gpv1′ gpv2′*)
using *assms*(*1*)
apply(*coinduction arbitrary*: *gpv1 gpv1′*)
using *assms*(*2*)
apply(*fastforce simp add*: *spmf-rel-map intro*!: *rel-spmf-try-spmf dest*: *eq-$\mathcal{I}$-gpvD elim*!: *rel-spmf-mono-strong eq-$\mathcal{I}$-generat.cases*)
done

lemma *eq-$\mathcal{I}$-gpv-map-gpv′*:
assumes *eq-$\mathcal{I}$-gpv* (*BNF-Def.vimage2p f f′ A*) (*map-$\mathcal{I}$ g h $\mathcal{I}$*) *gpv1 gpv2*
shows *eq-$\mathcal{I}$-gpv A $\mathcal{I}$* (*map-gpv′ f g h gpv1*) (*map-gpv′ f′ g h gpv2*)
using *assms*
proof(*coinduction arbitrary*: *gpv1 gpv2*)
case *eq-$\mathcal{I}$-gpv*
from *this*[*THEN eq-$\mathcal{I}$-gpvD*] **show** *?case*
apply(*simp add*: *spmf-rel-map*)
apply(*erule rel-spmf-mono*)
apply(*auto 4 4 simp add*: *BNF-Def.vimage2p-def elim*!: *eq-$\mathcal{I}$-generat.cases*)
done
qed

lemma *eq-$\mathcal{I}$-converter-map-converter*:
assumes *map-$\mathcal{I}$* (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$, *map-$\mathcal{I}$ f′ g′ $\mathcal{I}'$* $\vdash_C$ *conv1* $\sim$ *conv2*

and *inj f surj g*
shows $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *map-converter f g f′ g′ conv1* ∼ *map-converter f g f′ g′ conv2*
using *assms*(*1*)
proof(*coinduction arbitrary*: *conv1 conv2*)
case *eq-*$\mathcal{I}$*-converter*
from *this*(*2*) **have** *f q* ∈ *outs-*$\mathcal{I}$ (*map-*$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$)
using *assms*(*2*) **by** *simp*
from *eq-*$\mathcal{I}$*-converter*(*1*)[*THEN eq-*$\mathcal{I}$*-converterD*, *OF this*] **show** *?case* **using** *assms*(*2*,*3*)
apply *simp*
apply(*rule eq-*$\mathcal{I}$*-gpv-map-gpv′*)
apply(*simp add*: *BNF-Def.vimage2p-def prod.rel-map*)
apply(*erule eq-*$\mathcal{I}$*-gpv-mono′*)
apply(*auto 4 4 simp add*: *eq-onp-def surj-f-inv-f*)
done
qed

lemma *resource-of-oracle-run-resource*: *resource-of-oracle run-resource res* = *res*
by(*coinduction arbitrary*: *res*)(*auto simp add*: *rel-fun-def spmf-rel-map intro*!: *rel-spmf-reflI*)

lemma *connect-map-gpv′*:
connect (*map-gpv′ f g h adv*) *res* = *map-spmf f* (*connect adv* (*map-resource g h res*))
unfolding *connect-def*
by(*subst* (*3*) *resource-of-oracle-run-resource*[*symmetric*])
(*simp add*: *exec-gpv-map-gpv′ map-resource-resource-of-oracle spmf.map-comp exec-gpv-resource-of-oracle*)

primcorec *fail-resource* :: (*′a*, *′b*) *resource* **where**
run-resource fail-resource = (λ-. *return-pmf None*)

lemma *WT-fail-resource* [*WT-intro*]: $\mathcal{I}$ ⊢*res fail-resource* √
by(*rule WT-resourceI*) *simp*

context fixes *y* :: *′b* **begin**

primcorec *const-resource* :: (*′a*, *′b*) *resource* **where**
run-resource const-resource = (λ-. *map-spmf* (*map-prod id* (λ-. *const-resource*)) (*return-spmf* (*y*, ())))

end

lemma *const-resource-sel* [*simp*]: *run-resource* (*const-resource y*) = (λ-. *return-spmf* (*y*, *const-resource y*))
by *simp*

declare *const-resource.sel* [*simp del*]

lemma *lossless-const-resource* [*simp*]: *lossless-resource* $\mathcal{I}$ (*const-resource y*)
by(*coinduction*) *simp*

lemma *WT-const-resource* [*simp*]:
$\mathcal{I}$ ⊢*res const-resource y* √ ⟷ (∀ *x*∈*outs-*$\mathcal{I}$ $\mathcal{I}$. *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x*) (**is** *?lhs* ⟷ *?rhs*)
proof(*intro iffI ballI*)
show *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* **if** *?lhs* **and** *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **for** *x* **using** *WT-resourceD*[*OF that*] **by** *auto*
show *?lhs* **if** *?rhs* **using** *that* **by**(*coinduction*)(*auto*)
qed

context fixes *y* :: *'b* **begin**

primcorec *const-converter* :: (*'a*, *'b*, *'c*, *'d*) *converter* **where**
run-converter const-converter = (λ-. *map-gpv* (*map-prod id* (λ-. *const-converter*)) *id* (*Done* (*y*, ())))

end

lemma *const-converter-sel* [*simp*]: *run-converter* (*const-converter y*) = (λ-. *Done* (*y*, *const-converter y*))
by *simp*

lemma *attach-const-converter* [*simp*]: *attach* (*const-converter y*) *res* = *const-resource y*
by(*coinduction*)(*simp add*: *rel-fun-def*)

declare *const-converter.sel* [*simp del*]

lemma *comp-const-converter* [*simp*]: *comp-converter* (*const-converter x*) *conv* = *const-converter x*
by(*coinduction*)(*simp add*: *rel-fun-def*)

lemma *interaction-bounded-const-converter* [*simp*, *interaction-bound*]:
interaction-any-bounded-converter (*const-converter Fault*) *bound*
by(*coinduction*) *simp*

primcorec *merge-exception-converter* :: (*'a*, (*'b* + *'c*) *exception*, *'a*, *'b exception* + *'c exception*) *converter* **where**
run-converter merge-exception-converter =
(λ*x*. *map-gpv* (*map-prod id* (λ*conv*. *case conv of None* ⇒ *merge-exception-converter* | *Some conv'* ⇒ *conv'*)) *id* (
Pause x (λ*y*. *Done* (*case merge-exception y of Fault* ⇒ (*Fault*, *Some* (*const-converter Fault*))
| *OK y'* ⇒ (*OK y'*, *None*)))))

lemma *merge-exception-converter-sel* [*simp*]:

run-converter merge-exception-converter x =
Pause x (*λy*. *Done* (*case merge-exception y of Fault* ⇒ (*Fault*, *const-converter Fault*) | *OK y′* ⇒ (*OK y′*, *merge-exception-converter*)))
by(*simp add*: *o-def fun-eq-iff split*: *exception.split*)

declare *merge-exception-converter.sel*[*simp del*]

lemma *plossless-const-converter*[*simp*]: *plossless-converter* $\mathcal{I}$ $\mathcal{I}'$ (*const-converter x*)
by(*coinduction*) *auto*

lemma *plossless-merge-exception-converter* [*simp*]:
plossless-converter (*exception-*$\mathcal{I}$ ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$)) (*exception-*$\mathcal{I}$ $\mathcal{I}$ $\oplus_{\mathcal{I}}$ *exception-*$\mathcal{I}$ $\mathcal{I}'$) *merge-exception-converter*
by(*coinduction*) *auto*

lemma *WT-const-converter* [*WT-intro*, *simp*]:
$\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *const-converter x* $\surd$ **if** ∀ *q* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$. *x* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *q*
by(*coinduction*)(*auto simp add*: *that*)

lemma *WT-merge-exception-converter* [*WT-intro*, *simp*]:
exception-$\mathcal{I}$ ($\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2'$), *exception-*$\mathcal{I}$ $\mathcal{I}1'$ $\oplus_{\mathcal{I}}$ *exception-*$\mathcal{I}$ $\mathcal{I}2'$ $\vdash_C$ *merge-exception-converter* $\surd$
by(*coinduction*) *auto*

lemma *inline-left-gpv-merge-exception-converter*:
bind-gpv (*inline run-converter* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*left-gpv gpv*))) *merge-exception-converter*) (*λ*(*x*, *conv′*). *case x of None* ⇒ *Fail* | *Some x′* ⇒ *Done* (*x*, *conv′*)) =
bind-gpv (*left-gpv* (*map-gpv′ id id option-of-exception* (*gpv-stop gpv*))) (*λx*. *case x of None* ⇒ *Fail* | *Some x′* ⇒ *Done* (*x*, *merge-exception-converter*))
apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
apply(*simp add*: *bind-gpv.sel inline-sel map-bind-spmf bind-map-spmf del*: *bind-gpv-sel′*)
apply(*subst inline1-unfold*)
apply(*clarsimp simp add*: *bind-map-spmf intro*!: *rel-spmf-bind-reflI simp add*: *generat.map-comp case-map-generat o-def split*!: *generat.split intro*!: *rel-funI*)
subgoal for *gpv out c input* **by**(*cases input*; *auto split*!: *exception.split*)
done

lemma *inline-right-gpv-merge-exception-converter*:
bind-gpv (*inline run-converter* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*right-gpv gpv*))) *merge-exception-converter*) (*λ*(*x*, *conv′*). *case x of None* ⇒ *Fail* | *Some x′* ⇒ *Done* (*x*, *conv′*)) =
bind-gpv (*right-gpv* (*map-gpv′ id id option-of-exception* (*gpv-stop gpv*))) (*λx*. *case x of None* ⇒ *Fail* | *Some x′* ⇒ *Done* (*x*, *merge-exception-converter*))
apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
apply(*simp add*: *bind-gpv.sel inline-sel map-bind-spmf bind-map-spmf del*: *bind-gpv-sel′*)
apply(*subst inline1-unfold*)
apply(*clarsimp simp add*: *bind-map-spmf intro*!: *rel-spmf-bind-reflI simp add*:

generat.map-comp case-map-generat o-def split!: *generat.split intro*!: *rel-funI*)
subgoal for *gpv out c input* **by**(*cases input*; *auto split*!: *exception.split*)
done

3.1 *Constructive-Cryptography.Wiring*

abbreviation (*input*)
id-wiring :: ('*a*, '*b*, '*a*, '*b*) *wiring* (‹1_w›)
where
id-wiring ≡ (*id*, *id*)

definition
swap-lassocr$_w$ :: ('*a* + '*b* + '*c*, '*d* + '*e* + '*f*, '*b* + '*a* + '*c*, '*e* + '*d* + '*f*) *wiring*
where
swap-lassocr$_w$ ≡ *rassocl*$_w$ $\circ_w$ ((*swap*$_w$ $|_w$ 1_w) $\circ_w$ *lassocr*$_w$)

schematic-goal
wiring-swap-lassocr[*wiring-intro*]: *wiring ?𝓘1 ?𝓘2 swap-lassocr swap-lassocr*$_w$
unfolding *swap-lassocr-def swap-lassocr*$_w$*-def*
by(*rule wiring-intro*)+

definition
parallel-wiring$_w$ :: (('*a* + '*b*) + ('*c* + '*d*), ('*e* + '*f*) + ('*g* + '*h*),
('*a* + '*c*) + ('*b* + '*d*), ('*e* + '*g*) + ('*f* + '*h*)) *wiring*
where
parallel-wiring$_w$ ≡ *lassocr*$_w$ $\circ_w$ ((1_w $|_w$ *swap-lassocr*$_w$) $\circ_w$ *rassocl*$_w$)

schematic-goal
wiring-parallel-wiring[*wiring-intro*]: *wiring ?𝓘1 ?𝓘2 parallel-wiring parallel-wiring*$_w$
unfolding *parallel-wiring-def parallel-wiring*$_w$*-def*
by(*rule wiring-intro*)+

lemma *lassocr-inverse*: *rassocl*$_C$ ⊙ *lassocr*$_C$ = 1_C
unfolding *rassocl*$_C$*-def lassocr*$_C$*-def*
apply(*simp add*: *comp-converter-map1-out comp-converter-map-converter2 comp-converter-id-right*)
apply(*subst map-converter-id-move-right*)
apply(*simp add*: *o-def id-def*[*symmetric*])
done

lemma *rassocl-inverse*: *lassocr*$_C$ ⊙ *rassocl*$_C$ = 1_C
unfolding *rassocl*$_C$*-def lassocr*$_C$*-def*
apply(*simp add*: *comp-converter-map1-out comp-converter-map-converter2 comp-converter-id-right*)
apply(*subst map-converter-id-move-right*)
apply(*simp add*: *o-def id-def*[*symmetric*])
done

lemma *swap-sum-swap-sum* [*simp*]: *swap-sum* (*swap-sum x*) = *x*
by(*cases x*) *simp-all*

lemma *inj-on-lsumr* [*simp*]: *inj-on lsumr A*
  by(*auto simp add*: *inj-on-def elim*: *lsumr.elims*)

lemma *inj-on-rsuml* [*simp*]: *inj-on rsuml A*
  by(*auto simp add*: *inj-on-def elim*: *rsuml.elims*)

lemma *bij-lsumr* [*simp*]: *bij lsumr*
  by(*rule o-bij*[**where** *g*=*rsuml*]) *auto*

lemma *bij-swap-sum* [*simp*]: *bij swap-sum*
  by(*rule o-bij*[**where** *g*=*swap-sum*]) *auto*

lemma *bij-rsuml* [*simp*]: *bij rsuml*
  by(*rule o-bij*[**where** *g*=*lsumr*]) *auto*

lemma *bij-lassocr-swap-sum* [*simp*]: *bij lassocr-swap-sum*
  unfolding *lassocr-swap-sum-def*
  by(*simp add*: *bij-comp*)

lemma *inj-lassocr-swap-sum* [*simp*]: *inj lassocr-swap-sum*
  by(*simp add*: *bij-is-inj*)

lemma *inv-rsuml* [*simp*]: *inv-into UNIV rsuml* = *lsumr*
  by(*rule inj-imp-inv-eq*) *auto*

lemma *inv-lsumr* [*simp*]: *inv-into UNIV lsumr* = *rsuml*
  by(*rule inj-imp-inv-eq*) *auto*

lemma *lassocr-swap-sum-inverse* [*simp*]: *lassocr-swap-sum* (*lassocr-swap-sum x*) =
x
 by(*simp add*: *lassocr-swap-sum-def sum.map-comp o-def id-def*[*symmetric*] *sum.map-id*)

lemma *inv-lassocr-swap-sum* [*simp*]: *inv-into UNIV lassocr-swap-sum* = *lassocr-swap-sum*
  by(*rule inj-imp-inv-eq*)(*simp-all add*: *sum.map-comp sum.inj-map bij-def surj-iff*
sum.map-id)

lemma *swap-inverse*: $swap_C \odot swap_C = 1_C$
  unfolding $swap_C$*-def*
 apply(*simp add*: *comp-converter-map1-out comp-converter-map-converter2 comp-converter-id-right*)
  apply(*subst map-converter-id-move-right*)
  apply(*simp add*: *o-def id-def*[*symmetric*])
  done

lemma *swap-lassocr-inverse*: $\mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3), \mathcal{I}1 \oplus_\mathcal{I} (\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3) \vdash_C$ *swap-lassocr*
$\odot$ *swap-lassocr* $\sim 1_C$
  (**is** *?*$\mathcal{I}$,- $\vdash_C$ *?lhs* $\sim$ -)
proof −
  have *?lhs* = $(rassocl_C \odot (swap_C \mid_= 1_C)) \odot (lassocr_C \odot rassocl_C) \odot ((swap_C \mid_=$

1_C) ⊙ $lassocr_C$)
by(*simp add*: *swap-lassocr-def comp-converter-assoc*)
also have ... = $rassocl_C$ ⊙ (($swap_C$ ⊙ $swap_C$) $|_=$ 1_C) ⊙ $lassocr_C$
unfolding *rassocl-inverse comp-converter-id-left*
by(*simp add*: *parallel-converter2-comp1-out comp-converter-assoc*)
also have *?I*,*?I* $\vdash_C$ ... ∼ $rassocl_C$ ⊙ 1_C ⊙ $lassocr_C$ **unfolding** *swap-inverse*
by(*rule eq-I-converter-reflI eq-I-comp-cong WT-intro parallel-converter2-id-id*)+
also have $rassocl_C$ ⊙ 1_C ⊙ $lassocr_C$ = 1_C **by**(*simp add*: *comp-converter-id-left lassocr-inverse*)
finally show *?thesis* **.**
qed

lemma *parallel-wiring-inverse*:
($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) $\oplus_{\mathcal{I}}$ ($\mathcal{I}3 \oplus_{\mathcal{I}} \mathcal{I}4$),($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) $\oplus_{\mathcal{I}}$ ($\mathcal{I}3 \oplus_{\mathcal{I}} \mathcal{I}4$) $\vdash_C$ *parallel-wiring* ⊙ *parallel-wiring* ∼ 1_C
(**is** *?I*, - $\vdash_C$ *?lhs* ∼ -)
proof −
have *?lhs* = ($lassocr_C$ ⊙ (1_C $|_=$ *swap-lassocr*)) ⊙ ($rassocl_C$ ⊙ $lassocr_C$) ⊙ ((1_C $|_=$ *swap-lassocr*) ⊙ $rassocl_C$)
by(*simp add*: *parallel-wiring-def comp-converter-assoc*)
also have ... = ($lassocr_C$ ⊙ (1_C $|_=$ *swap-lassocr*)) ⊙ (1_C $|_=$ *swap-lassocr*) ⊙ $rassocl_C$
by(*simp add*: *lassocr-inverse comp-converter-id-left*)
also have ... = $lassocr_C$ ⊙ (1_C $|_=$ (*swap-lassocr* ⊙ *swap-lassocr*)) ⊙ $rassocl_C$
by(*simp add*: *parallel-converter2-comp2-out comp-converter-assoc*)
also have *?I*,*?I* $\vdash_C$ ... ∼ $lassocr_C$ ⊙ (1_C $|_=$ 1_C) ⊙ $rassocl_C$
by(*rule eq-I-converter-reflI eq-I-comp-cong parallel-converter2-eq-I-cong WT-intro swap-lassocr-inverse*)+
also have *?I*,*?I* $\vdash_C$ $lassocr_C$ ⊙ (1_C $|_=$ 1_C) ⊙ $rassocl_C$ ∼ $lassocr_C$ ⊙ 1_C ⊙ $rassocl_C$
by(*rule eq-I-converter-reflI eq-I-comp-cong parallel-converter2-id-id WT-intro*)+
also have $lassocr_C$ ⊙ 1_C ⊙ $rassocl_C$ = 1_C **by**(*simp add*: *comp-converter-id-left rassocl-inverse*)
finally show *?thesis* **.**
qed

— Analogous to *attach-wiring* in Wiring.thy
definition
attach-wiring-right ::
('a, 'b, 'c, 'd) *wiring* ⇒
('s ⇒ 'e ⇒ ('f × 's, 'a, 'b) *gpv*) ⇒ ('s ⇒ 'e ⇒ ('f × 's, 'c, 'd) *gpv*)
where
attach-wiring-right = (λ(f, g). *map-fun id* (*map-fun id* (*map-gpv' id f g*)))

lemma
attach-wiring-right-simps:
attach-wiring-right (f, g) = *map-fun id* (*map-fun id* (*map-gpv' id f g*))
by(*simp add*: *attach-wiring-right-def*)

lemma
comp-converter-of-callee-wiring:
assumes *wiring*: *wiring* $\mathcal{I}2$ $\mathcal{I}3$ *conv w*
and *WT*: $\mathcal{I}1$, $\mathcal{I}2 \vdash_C$ *CNV callee s* $\surd$
shows $\mathcal{I}1$, $\mathcal{I}3 \vdash_C$ *CNV callee s* $\odot$ *conv* $\sim$ *CNV* (*attach-wiring-right w callee*) *s*
using *wiring*
proof *cases*
case (*wiring f g*)
from - *wiring*(*2*) **have** $\mathcal{I}1$,$\mathcal{I}3 \vdash_C$ *CNV callee s* $\odot$ *conv* $\sim$ *CNV callee s* $\odot$ *map-converter id id f g* 1_C
by(*rule eq-$\mathcal{I}$-comp-cong*)(*rule eq-$\mathcal{I}$-converter-reflI*[*OF WT*])
also have *CNV callee s* $\odot$ *map-converter id id f g* 1_C = *map-converter id id f g* (*CNV callee s*)
by(*subst comp-converter-map-converter2*)(*simp add*: *comp-converter-id-right*)
also have ... = *CNV* (*attach-wiring-right w callee*) *s*
by(*simp add*: *map-converter-of-callee attach-wiring-right-simps wiring*(*1*) *prod.map-id0*)
finally show *?thesis* **.**
qed

lemma *attach-wiring-right-comp-wiring*:
attach-wiring-right (*w1* $\circ_w$ *w2*) *callee* = *attach-wiring-right w2* (*attach-wiring-right w1 callee*)
by(*simp add*: *attach-wiring-right-def comp-wiring-def split-def map-fun-def o-def map-gpv′-comp id-def fun-eq-iff*)

lemma *attach-wiring-comp-wiring*:
attach-wiring (*w1* $\circ_w$ *w2*) *callee* = *attach-wiring w1* (*attach-wiring w2 callee*)
unfolding *attach-wiring-def comp-wiring-def*
by (*simp add*: *split-def map-fun-def o-def map-gpv-conv-map-gpv′ map-gpv′-comp id-def map-prod-def*)

3.2 Probabilistic finite converter

coinductive *pfinite-converter* :: ($'a$, $'b$) $\mathcal{I}$ $\Rightarrow$ ($'c$, $'d$) $\mathcal{I}$ $\Rightarrow$ ($'a$, $'b$, $'c$, $'d$) *converter* $\Rightarrow$ *bool*
for $\mathcal{I}$ $\mathcal{I}'$ **where**
pfinite-converterI: *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* **if**
$\bigwedge a$. $a \in$ *outs-$\mathcal{I}$* $\mathcal{I}$ $\Longrightarrow$ *pfinite-gpv* $\mathcal{I}'$ (*run-converter conv a*)
$\bigwedge a$ *b conv′*. ⟦ $a \in$ *outs-$\mathcal{I}$* $\mathcal{I}$; (*b*, *conv′*) $\in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) ⟧ $\Longrightarrow$ *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv′*

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

using *assms*(*1*) **by**(*rule pfinite-converter.coinduct*)(*auto dest*: *step*)

lemma *pfinite-converterD*:
⟦ *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*; *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ *pfinite-gpv* $\mathcal{I}'$ (*run-converter conv a*) ∧
(∀ (*b*, *conv'*) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*). *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv'*)
by(*auto elim*: *pfinite-converter.cases*)

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

lemma *pfinite-converter-mono*:
assumes ∗: *pfinite-converter* $\mathcal{I}1$ $\mathcal{I}2$ *conv*
and *le*: *outs-*$\mathcal{I}$ $\mathcal{I}1'$ ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}1$ $\mathcal{I}2 \leq \mathcal{I}2'$
and *WT*: $\mathcal{I}1$, $\mathcal{I}2 \vdash_C$ *conv* √
shows *pfinite-converter* $\mathcal{I}1'$ $\mathcal{I}2'$ *conv*
using ∗ *WT*
apply(*coinduction arbitrary*: *conv*)
apply(*drule pfinite-converterD*)
apply(*erule le*(*1*)[*THEN subsetD*])
apply(*drule WT-converterD'*)
apply(*erule le*(*1*)[*THEN subsetD*])
using *le*(*2*)[*THEN responses-*$\mathcal{I}$*-mono*]
by(*auto intro*: *pfinite-gpv-mono*[*OF - le*(*2*)] *results-gpv-mono*[*OF le*(*2*), *THEN subsetD*] *dest*: *bspec*)

context *raw-converter-invariant* **begin**
lemma *pfinite-converter-of-callee*:
assumes *step*: ⋀*x s*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *pfinite-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ (*converter-of-callee callee s*)
using *I*
by(*coinduction arbitrary*: *s*)(*auto 4 3 simp add*: *step dest*: *results-callee*)
end

lemma *raw-converter-invariant-run-pfinite-converter*:
raw-converter-invariant $\mathcal{I}$ $\mathcal{I}'$ *run-converter* (λ*conv*. *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* ∧ $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *conv* √)
by(*unfold-locales*)(*auto dest*: *WT-converterD pfinite-converterD*)

interpretation *run-pfinite-converter*: *raw-converter-invariant*
$\mathcal{I}$ $\mathcal{I}'$ *run-converter* λ*conv*. *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* ∧ $\mathcal{I}$,$\mathcal{I}' \vdash_C$ *conv* √ **for** $\mathcal{I}$ $\mathcal{I}'$
by(*rule raw-converter-invariant-run-pfinite-converter*)

named-theorems *pfinite-intro Introduction rules for probabilistic finiteness*

lemma *pfinite-id-converter* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$ $\mathcal{I}$ *id-converter*
by(*coinduction*) *simp*

lemma *pfinite-fail-converter* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *fail-converter*
by *coinduction simp*

lemma *pfinite-parallel-converter2* [*pfinite-intro*]:
pfinite-converter ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) ($\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2'$) (*conv1* $|_=$ *conv2*)
if *pfinite-converter* $\mathcal{I}1$ $\mathcal{I}1'$ *conv1* *pfinite-converter* $\mathcal{I}2$ $\mathcal{I}2'$ *conv2*
using *that* **by**(*coinduction arbitrary*: *conv1 conv2*)(*fastforce dest*: *pfinite-converterD*)

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

lemma *expectation-gpv-1-le-inline*:
defines *expectation-gpv2* ≡ *expectation-gpv 1* $\mathcal{I}'$
assumes *callee*: ⋀*s x*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *pfinite-gpv* $\mathcal{I}'$ (*callee s x*)
and *WT-gpv*: $\mathcal{I}$ ⊢*g gpv* √
and *I*: *I s*
and *f-le-1*: ⋀*x*. *f x* ≤ *1*
shows *expectation-gpv 1* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*)
using *WT-gpv I*
proof(*induction arbitrary*: *gpv s rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv'*)
have (∫$^+$ *x*. (*case x of Pure a* ⇒ *f a* | *IO out c* ⇒ ⨅*r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) ∂*measure-spmf* (*the-gpv gpv*)) + *1* ∗ *ennreal* (*pmf* (*the-gpv gpv*) *None*) =
(∑$^+$ *x*. *pmf* (*the-gpv gpv*) *x* ∗ (*case x of Some* (*Pure a*) ⇒ *f a* | *Some* (*IO out c*) ⇒ ⨅*r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*) | *None* ⇒ *1*))
apply(*simp add*: *nn-integral-measure-spmf-conv-measure-pmf nn-integral-restrict-space nn-integral-measure-pmf*)
apply(*subst* (*2*) *nn-integral-disjoint-pair-countspace*[**where** *B*=*range Some* **and** *C*={*None*}, *simplified*, *folded UNIV-option-conv*, *simplified*])
apply(*auto simp add*: *mult.commute intro*!: *nn-integral-cong split*: *split-indicator*)
done
also have ... ≤ (∑$^+$ *x*. *pmf* (*the-gpv gpv*) *x* ∗ (*case x of None* ⇒ *1* | *Some* (*Pure a*) ⇒ *f a* | *Some* (*IO out c*) ⇒
(∑$^+$ *x*. *ennreal* (*pmf* (*the-gpv* (*callee s out*) ≫= *case-generat* (λ(*x*, *y*). *inline1 callee* (*c x*) *y*) (λ*out rpv'*. *return-spmf* (*Inr* (*out*, *rpv'*, *c*)))) *x*) ∗
(*case x of None* ⇒ *1* | *Some* (*Inl* (*a*, *s*)) ⇒ *f a*
| *Some* (*Inr* (*r*, *rpv*, *rpv'*)) ⇒ ⨅*x*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *r*. *expectation-gpv 1* $\mathcal{I}'$ (λ(*x*, *s'*). *expectation-gpv 1* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*rpv' x*) *s'*)) (*rpv x*)))))
(**is** *nn-integral - ?lhs* ≤ *nn-integral - ?rhs*)
proof(*rule nn-integral-mono*)
fix *x* :: (*'a*, *'call*, (*'a*, *'call*, *'ret*) *rpv*) *generat option*
consider (*None*) *x* = *None* | (*Pure*) *a* **where** *x* = *Some* (*Pure a*)
| (*IO*) *out c* **where** *x* = *Some* (*IO out c*) *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
| (*outside*) *out c* **where** *x* = *Some* (*IO out c*) *IO out c* ∉ *set-spmf* (*the-gpv gpv*)

by (*metis dest-IO.elims not-None-eq*)
then show *?lhs x* ≤ *?rhs x*
proof *cases*
case *None* **then show** *?thesis* **by** *simp*
next
case *Pure* **then show** *?thesis* **by** *simp*
next
case (*IO out c*)
with *step.prems* **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*auto dest*: *WT-gpvD*)
then obtain *response* **where** *resp*: *response* ∈ *responses-ℐ ℐ out* **unfolding** *in-outs-ℐ-iff-responses-ℐ* **by** *blast*
with *out step.prems IO* **have** *WT-resp* [*WT-intro*]: *ℐ* ⊢g *c response* √ **by**(*auto dest*: *WT-gpvD*)
have *exp-resp*: *expectation-gpv′* (*c response*) ≤ *1*
using *step.hyps*[*of c response*] *expectation-gpv-mono*[*of 1 1 f* λ-. *1 ℐ c response*] *expectation-gpv-const-le*[*OF WT-resp*, *of 1 1*]
by(*simp add*: *le-fun-def f-le-1*)

have (⨅*r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) =
(∫⁺ *generat*. (⨅*r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ∂*measure-spmf* (*the-gpv* (*callee s out*))) +
(⨅*r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ∗ (*1* − *ennreal* (*weight-spmf* (*the-gpv* (*callee s out*))))
by(*simp add*: *measure-spmf.emeasure-eq-measure add-mult-distrib2*[*symmetric*] *semiring-class.distrib-left*[*symmetric*] *add-diff-inverse-ennreal weight-spmf-le-1*)
also have … ≤ (∫⁺ *generat*. (⨅*r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ∂*measure-spmf* (*the-gpv* (*callee s out*))) +
1 ∗ *ennreal* (*pmf* (*the-gpv* (*callee s out*)) *None*) **unfolding** *pmf-None-eq-weight-spmf*
by(*intro add-mono mult-mono order-refl INF-lower2*[*OF resp*])(*auto simp add*: *ennreal-minus*[*symmetric*] *weight-spmf-le-1 exp-resp*)
also have … = (∑⁺ *z*. *ennreal* (*pmf* (*the-gpv* (*callee s out*)) *z*) ∗ (*case z of None* ⇒ *1* | *Some generat* ⇒ (⨅*r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*))))
apply(*simp add*: *nn-integral-measure-spmf-conv-measure-pmf nn-integral-restrict-space nn-integral-measure-pmf del*: *nn-integral-const*)
apply(*subst* (*2*) *nn-integral-disjoint-pair-countspace*[**where** *B*=*range Some* **and** *C*={*None*}, *simplified*, *folded UNIV-option-conv*, *simplified*])
apply(*auto simp add*: *mult.commute intro*!: *nn-integral-cong split*: *split-indicator*)
done
also have … ≤ (∑⁺ *z*. *ennreal* (*pmf* (*the-gpv* (*callee s out*)) *z*) ∗
(*case z of None* ⇒ *1* | *Some* (*IO out′ rpv′*) ⇒ ⨅*x*∈*responses-ℐ ℐ′ out′*. *expectation-gpv 1 ℐ′* (λ(*x*, *s′*). *expectation-gpv 1 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*c x*) *s′*)) (*rpv′ x*)
| *Some* (*Pure* (*r*, *s′*)) ⇒ (∑⁺ *x*. *ennreal* (*pmf* (*inline1 callee* (*c r*) *s′*) *x*) ∗ (*case x of None* ⇒ *1* | *Some* (*Inl* (*a*, *s*)) ⇒ *f a* | *Some* (*Inr* (*out′*, *rpv*, *rpv′*)) ⇒
⨅*x*∈*responses-ℐ ℐ′ out′*. *expectation-gpv 1 ℐ′* (λ(*x*, *s′*). *expectation-gpv 1 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv′ x*) *s′*)) (*rpv x*)))))
(**is** *nn-integral - ?lhs2* ≤ *nn-integral - ?rhs2*)
proof(*intro nn-integral-mono*)
fix *z* :: (*′ret* × *′s*, *′call′*, (*′ret* × *′s*, *′call′*, *′ret′*) *rpv*) *generat option*

consider (*None*) $z = None$ | (*Pure*) x' s' **where** $z = Some\ (Pure\ (x', s'))$ $Pure\ (x', s') \in$ *set-spmf* (*the-gpv* (*callee s out*))
| (IO') out' c' **where** $z = Some\ (IO\ out'\ c')$ $IO\ out'\ c' \in$ *set-spmf* (*the-gpv* (*callee s out*))
| (*outside*) *generat* **where** $z =$ *Some generat generat* $\notin$ *set-spmf* (*the-gpv* (*callee s out*))
by (*metis dest-IO.elims not-Some-eq old.prod.exhaust*)
then show *?lhs2 z* $\leq$ *?rhs2 z*
proof *cases*
case *None* **then show** *?thesis* **by** *simp*
next
case *Pure*
hence $(x', s') \in$ *results-gpv* $\mathcal{I}'$ (*callee s out*) **by**(*simp add*: *results-gpv.Pure*)
with *results-callee step.prems out* **have** *x*: $x' \in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *out* **and** s': *I* s' **by** *auto*
with *IO out step.prems* **have** *WT-c* [*WT-intro*]: $\mathcal{I} \vdash g\ c\ x'\ \surd$ **by**(*auto dest*: *WT-gpvD*)
from *x* **have** (*INF* $r \in$*responses-$\mathcal{I}$* $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) $\leq$ *expectation-gpv'* ($c\ x'$) **by**(*rule INF-lower*)
also have ... $\leq$ *expectation-gpv2* ($\lambda(x, s)$. *f x*) (*inline callee* ($c\ x'$) s') **using** *WT-c* s' **by**(*rule step.IH*)
also have ... $= \int^+ xx$. (*case xx of Inl* (*x*, -) $\Rightarrow$ *f x*
| *Inr* (out', $callee'$, *rpv*) $\Rightarrow$ *INF* $r' \in$*responses-$\mathcal{I}$* $\mathcal{I}'$ out'. *expectation-gpv 1* $\mathcal{I}'$ ($\lambda(r, s')$. *expectation-gpv 1* $\mathcal{I}'$ ($\lambda(x, s)$. *f x*) (*inline callee* (*rpv r*) s')) ($callee'$ r'))
∂*measure-spmf* (*inline1 callee* ($c\ x'$) s') + *ennreal* (*pmf* (*the-gpv* (*inline callee* ($c\ x'$) s')) *None*)
unfolding *expectation-gpv2-def*
by(*subst expectation-gpv.simps*)(*auto simp add*: *inline-sel split-def o-def intro*!: *nn-integral-cong split*: *generat.split sum.split*)
also have ... $= (\sum^+ xx$. *ennreal* (*pmf* (*inline1 callee* ($c\ x'$) s') *xx*) $*$ (*case xx of None* $\Rightarrow$ *1* | *Some* (*Inl* (*x*, -)) $\Rightarrow$ *f x*
| *Some* (*Inr* (out', $callee'$, *rpv*)) $\Rightarrow$ *INF* $r' \in$*responses-$\mathcal{I}$* $\mathcal{I}'$ out'. *expectation-gpv 1* $\mathcal{I}'$ ($\lambda(r, s')$. *expectation-gpv 1* $\mathcal{I}'$ ($\lambda(x, s)$. *f x*) (*inline callee* (*rpv r*) s')) ($callee'$ r')))
apply(*subst inline-sel*)
apply(*simp add*: *nn-integral-measure-spmf-conv-measure-pmf nn-integral-restrict-space nn-integral-measure-pmf pmf-map-spmf-None del*: *nn-integral-const*)
apply(*subst* (*2*) *nn-integral-disjoint-pair-countspace*[**where** *B*=*range Some* **and** *C*={*None*}, *simplified*, *folded UNIV-option-conv*, *simplified*])
apply(*auto simp add*: *mult.commute intro*!: *nn-integral-cong split*: *split-indicator*)
done
finally show *?thesis* **using** *Pure* **by**(*simp add*: *mult-mono*)
next
case IO'
then have out': $out' \in$ *outs-$\mathcal{I}$* $\mathcal{I}'$ **using** *WT-callee out step.prems* **by**(*auto dest*: *WT-gpvD*)
have (*INF* $r \in$*responses-$\mathcal{I}$* $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) $\leq$ *min* (*INF* (*r*,

s′)∈(⋃ *r*′∈*responses-ℐ ℐ*′ *out*′. *results-gpv ℐ*′ (*c*′ *r*′)). *expectation-gpv*′ (*c r*)) *1*
using *IO*′ *results-callee*[*OF out*, *of s*] *step.prems* **by**(*intro INF-mono min.boundedI*)(*auto intro*: *results-gpv.IO intro*!: *INF-lower2*[*OF resp*] *exp-resp*)
also have ... ≤ (*INF r*′∈*responses-ℐ ℐ*′ *out*′. *min* (*INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*c*′ *r*′). *expectation-gpv*′ (*c r*)) *1*)
using *resp out*′ **unfolding** *inf-min*[*symmetric*] *in-outs-ℐ-iff-responses-ℐ*
by(*subst INF-inf-const2*)(*auto simp add*: *INF-UNION*)
also have ... ≤ (*INF r*′∈*responses-ℐ ℐ*′ *out*′. *expectation-gpv 1 ℐ*′ (λ(*r*′, *s*′). *expectation-gpv 1 ℐ*′ (λ(*x*, *s*). *f x*) (*inline callee* (*c r*′) *s*′)) (*c*′ *r*′))
(**is** - ≤ (*INF r*′∈-. *?r r*′))
proof(*rule INF-mono*, *rule bexI*)
fix *r*′
assume *r*′: *r*′ ∈ *responses-ℐ ℐ*′ *out*′
have *fin*: *pfinite-gpv ℐ*′ (*c*′ *r*′) **using** *callee*[*OF out*, *of s*] *IO*′ *r*′ *WT-callee*[*OF out*, *of s*] *step.prems* **by**(*auto dest*: *pfinite-gpv-ContD*)
have *min* (*INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*c*′ *r*′). *expectation-gpv*′ (*c r*)) *1* ≤ *min* (*INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*c*′ *r*′). *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c r*) *s*′)) *1*
using *IO IO*′ *step.prems out results-callee*[*OF out*, *of s*] *r*′
by(*intro min.mono*)(*auto intro*!: *INF-mono rev-bexI step.IH dest*: *WT-gpv-ContD intro*: *results-gpv.IO*)
also have ... ≤ *?r r*′ **unfolding** *expectation-gpv2-def* **using** *fin* **by**(*rule pfinite-INF-le-expectation-gpv*)
finally show *min* (*INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*c*′ *r*′). *expectation-gpv*′ (*c r*)) *1* ≤ ... .
qed
finally show *?thesis* **using** *IO*′ **by**(*simp add*: *mult-mono*)
next
case *outside* **then show** *?thesis* **by**(*simp add*: *in-set-spmf-iff-spmf*)
qed
qed
also have ... = (∑⁺ *z*. ∑⁺ *x*.
ennreal (*pmf* (*the-gpv* (*callee s out*)) *z*) ∗
ennreal (*pmf* (*case z of None* ⇒ *return-pmf None* | *Some* (*Pure* (*x*, *xb*)) ⇒ *inline1 callee* (*c x*) *xb* | *Some* (*IO out rpv*′) ⇒ *return-spmf* (*Inr* (*out*, *rpv*′, *c*))) *x*) ∗
(*case x of None* ⇒ *1* | *Some* (*Inl* (*a*, *s*)) ⇒ *f a* | *Some* (*Inr* (*out*, *rpv*, *rpv*′)) ⇒ ∏ *x*∈*responses-ℐ ℐ*′ *out*.
expectation-gpv 1 ℐ′ (λ(*x*, *s*′). *expectation-gpv 1 ℐ*′ (λ(*x*, *s*). *f x*) (*inline callee* (*rpv*′ *x*) *s*′)) (*rpv x*)))
(**is** ... = (∑⁺ *z*. ∑⁺ *x*. *?f x z*))
by(*auto intro*!: *nn-integral-cong split*!: *option.split generat.split simp add*: *mult.assoc nn-integral-cmult ennreal-indicator*)
also have (∑⁺ *z*. ∑⁺ *x*. *?f x z*) = (∑⁺ *x*. ∑⁺ *z*. *?f x z*)
by(*subst nn-integral-fst-count-space*[**where** *f*=*case-prod* -, *simplified*])(*simp add*: *nn-integral-snd-count-space*[*symmetric*])
also have ... = (∑⁺ *x*.
ennreal (*pmf* (*the-gpv* (*callee s out*) ≫= *case-generat* (λ(*x*, *y*). *inline1 callee* (*c x*) *y*) (λ*out rpv*′. *return-spmf* (*Inr* (*out*, *rpv*′, *c*)))) *x*) ∗

(*case x of None ⇒ 1* | *Some* (*Inl* (*a*, *s*)) ⇒ *f a* | *Some* (*Inr* (*r*, *rpv*, *rpv'*)) ⇒
∏*x*∈*responses-ℐ ℐ' r*. *expectation-gpv 1 ℐ'* (λ(*x*, *s'*). *expectation-gpv 1 ℐ'* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv' x*) *s'*)) (*rpv x*)))
by(*simp add*: *bind-spmf-def ennreal-pmf-bind nn-integral-multc*[*symmetric*] *nn-integral-measure-pmf*)
finally show *?thesis* **using** *IO* **by**(*auto intro*!: *mult-mono*)
next
case *outside* **then show** *?thesis* **by**(*simp add*: *in-set-spmf-iff-spmf*)
qed
qed
also have ... = (∑⁺ *y*. ∑⁺ *x*.
ennreal (*pmf* (*the-gpv gpv*) *y*) ∗
ennreal (*case y of None* ⇒ *pmf* (*return-pmf None*) *x* | *Some* (*Pure xa*) ⇒ *pmf* (*return-spmf* (*Inl* (*xa*, *s*))) *x*
| *Some* (*IO out rpv*) ⇒
pmf (*bind-spmf* (*the-gpv* (*callee s out*)) (λ*generat'* ⇒ *case generat' of Pure* (*x*, *y*) ⇒ *inline1 callee* (*rpv x*) *y* | *IO out rpv'* ⇒ *return-spmf* (*Inr* (*out*, *rpv'*, *rpv*)))) *x*) ∗
(*case x of None* ⇒ *1* | *Some* (*Inl* (*a*, *s*)) ⇒ *f a*
| *Some* (*Inr* (*out*, *rpv*, *rpv'*)) ⇒ ∏*x*∈*responses-ℐ ℐ' out*. *expectation-gpv 1 ℐ'* (λ(*x*, *s'*). *expectation-gpv 1 ℐ'* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv' x*) *s'*)) (*rpv x*)))
(**is** - = (∑⁺ *y*. ∑⁺ *x*. *?f x y*))
by(*auto intro*!: *nn-integral-cong split*!: *option.split generat.split simp add*: *nn-integral-cmult mult.assoc ennreal-indicator*)
also have (∑⁺ *y*. ∑⁺ *x*. *?f x y*) = (∑⁺ *x*. ∑⁺ *y*. *?f x y*)
by(*subst nn-integral-fst-count-space*[**where** *f*=*case-prod* -, *simplified*])(*simp add*: *nn-integral-snd-count-space*[*symmetric*])
also have ... = (∑⁺ *x*. (*pmf* (*inline1 callee gpv s*) *x*) ∗ (*case x of None* ⇒ *1* | *Some* (*Inl* (*a*, *s*)) ⇒ *f a* |
Some (*Inr* (*out*, *rpv*, *rpv'*)) ⇒ ∏*x*∈*responses-ℐ ℐ' out*. *expectation-gpv 1 ℐ'* (λ(*x*, *s'*). *expectation-gpv 1 ℐ'* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv' x*) *s'*)) (*rpv x*)))
by(*rewrite* **in** - = ⌑ *inline1.simps*)
(*auto simp add*: *bind-spmf-def ennreal-pmf-bind nn-integral-multc*[*symmetric*] *nn-integral-measure-pmf intro*!: *nn-integral-cong split*: *option.split generat.split*)
also have ... = (∫⁺ *res*. (*case res of Inl* (*a*, *s*) ⇒ *f a*
| *Inr* (*out*, *rpv*, *rpv'*) ⇒ ∏*x*∈*responses-ℐ ℐ' out*. *expectation-gpv 1 ℐ'* (λ(*x*, *s'*). *expectation-gpv 1 ℐ'* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv' x*) *s'*)) (*rpv x*))
∂*measure-spmf* (*inline1 callee gpv s*) +
ennreal (*pmf* (*inline1 callee gpv s*) *None*))
apply(*simp add*: *nn-integral-measure-spmf-conv-measure-pmf nn-integral-restrict-space nn-integral-measure-pmf*)
apply(*subst nn-integral-disjoint-pair-countspace*[**where** *B*=*range Some* **and** *C*={*None*}, *simplified*, *folded UNIV-option-conv*, *simplified*])
apply(*auto simp add*: *mult.commute intro*!: *nn-integral-cong split*: *split-indicator*)
done
also have ... = *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*) **unfolding** *expectation-gpv2-def*
by(*rewrite* **in** - = ⌑ *expectation-gpv.simps*, *subst* (*1 2*) *inline-sel*)

(*simp add*: *o-def pmf-map-spmf-None sum.case-distrib*[**where** *h*=*case-generat* - -] *split-def cong*: *sum.case-cong*)
finally show *?case* **.**
qed

lemma *pfinite-inline*:
assumes *fin*: *pfinite-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ [\![\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow$ *pfinite-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *pfinite-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
unfolding *pgen-lossless-gpv-def*
proof(*rule antisym*)
have *WT'*: $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$ **using** *WT I* **by**(*rule WT-gpv-inline-invar*)
from *expectation-gpv-const-le*[*OF WT'*, *of 1 1*]
show *expectation-gpv 1* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*) ≤ *1* **by**(*simp add*: *max-def*)

have *1* = *expectation-gpv 1* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *fin* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have ... ≤ *expectation-gpv 1* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*)
by(*rule expectation-gpv-1-le-inline*[*unfolded split-def*]; *rule callee I WT WT-callee order-refl*)
finally show *1* ≤ ... **.**
qed

end

lemma *pfinite-comp-converter* [*pfinite-intro*]:
pfinite-converter $\mathcal{I}1\ \mathcal{I}3$ (*conv1* ⊙ *conv2*)
if *pfinite-converter* $\mathcal{I}1\ \mathcal{I}2$ *conv1 pfinite-converter* $\mathcal{I}2\ \mathcal{I}3$ *conv2* $\mathcal{I}1,\mathcal{I}2 \vdash_C$ *conv1* $\surd$ $\mathcal{I}2,\mathcal{I}3 \vdash_C$ *conv2* $\surd$
using *that*
proof(*coinduction arbitrary*: *conv1 conv2*)
case *pfinite-converter*
have *conv1*: *pfinite-gpv* $\mathcal{I}2$ (*run-converter conv1 a*)
using *pfinite-converter*(*1*, *5*) **by**(*simp add*: *pfinite-converterD*)
have *conv2*: $\mathcal{I}2 \vdash g$ *run-converter conv1 a* $\surd$
using *pfinite-converter*(*3*, *5*) **by**(*simp add*: *WT-converterD*)
have *?pfinite* **using** *pfinite-converter*(*2*,*4*,*5*)
by(*auto intro*!:*run-pfinite-converter.pfinite-inline*[*OF conv1*] *dest*: *pfinite-converterD intro*: *conv2*)
moreover have *?step* (**is** ∀(*b*, *conv'*)∈*?res*. *?P b conv'* ∨ -)
proof(*clarify*)
fix *b conv''*
assume (*b*, *conv''*) ∈ *?res*
then obtain *conv1' conv2'* **where** [*simp*]: *conv''* = *comp-converter conv1' conv2'*
and *inline*: ((*b*, *conv1'*), *conv2'*) ∈ *results-gpv* $\mathcal{I}3$ (*inline run-converter* (*run-converter conv1 a*) *conv2*)
by *auto*

from *run-pfinite-converter.results-gpv-inline*[*OF inline conv2*] *pfinite-converter*(*2*,*4*)
have *run*: (*b*, *conv1*′) ∈ *results-gpv* $\mathcal{I}$*2* (*run-converter conv1 a*)
and ∗: *pfinite-converter* $\mathcal{I}$*2* $\mathcal{I}$*3* *conv2*′ $\mathcal{I}$*2*, $\mathcal{I}$*3* $\vdash_C$ *conv2*′ √ **by** *auto*
with *WT-converterD*(*2*)[*OF pfinite-converter*(*3*,*5*) *run*] *pfinite-converterD*[*THEN conjunct2*, *rule-format*, *OF pfinite-converter*(*1*,*5*) *run*]
show *?P b conv*′′ **by** *auto*
qed
ultimately show *?case* **..**
qed

lemma *pfinite-map-converter* [*pfinite-intro*]:
pfinite-converter $\mathcal{I}$ $\mathcal{I}$′ (*map-converter f g f*′ *g*′ *conv*) **if**
∗: *pfinite-converter* (*map-*$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$) (*map-*$\mathcal{I}$ *f*′ *g*′ $\mathcal{I}$′) *conv*
and *f*: *inj f* **and** *g*: *surj g*
using ∗
proof(*coinduction arbitrary*: *conv*)
case (*pfinite-converter a conv*)
with *f* **have** *a*: *inv-into UNIV f* (*f a*) ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by** *simp*
with *pfinite-converterD*[*OF* ‹*pfinite-converter - - conv*›, *of f a*] **have** *?pfinite* **by** *simp*
moreover have *?step*
proof(*safe*)
fix *r conv*′
assume (*r*, *conv*′) ∈ *results-gpv* $\mathcal{I}$′ (*run-converter* (*map-converter f g f*′ *g*′ *conv*) *a*)
then obtain *r*′ *conv*′′
where *results*: (*r*′, *conv*′′) ∈ *results-gpv* (*map-*$\mathcal{I}$ *f*′ *g*′ $\mathcal{I}$′) (*run-converter conv* (*f a*))
and *r*: *r* = *g r*′
and *conv*′: *conv*′ = *map-converter f g f*′ *g*′ *conv*′′
by *auto*
from *pfinite-converterD*[*OF* ‹*pfinite-converter - - conv*›, *THEN conjunct2*, *rule-format*, *OF - results*] *a r conv*′
show ∃ *conv*. *conv*′ = *map-converter f g f*′ *g*′ *conv* ∧
pfinite-converter (*map-*$\mathcal{I}$ (*inv-into UNIV f*) (*inv-into UNIV g*) $\mathcal{I}$) (*map-*$\mathcal{I}$ *f*′ *g*′ $\mathcal{I}$′) *conv*
by *auto*
qed
ultimately show *?case* **..**
qed

lemma *pfinite-lassocr*$_C$ [*pfinite-intro*]: *pfinite-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*$_C$
by(*coinduction*)(*auto simp add*: *lassocr*$_C$*-def*)

lemma *pfinite-rassocl*$_C$ [*pfinite-intro*]: *pfinite-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*$_C$
by(*coinduction*)(*auto simp add*: *rassocl*$_C$*-def*)

lemma *pfinite-swap*$_C$ [*pfinite-intro*]: *pfinite-converter* ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) ($\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}1$) *swap*$_C$
by(*coinduction*)(*auto simp add*: *swap*$_C$*-def*)

lemma *pfinite-swap-lassocr* [*pfinite-intro*]: *pfinite-converter* ($\mathcal{I}1 \oplus_\mathcal{I}$ ($\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}3$)) ($\mathcal{I}2 \oplus_\mathcal{I}$ ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3$)) *swap-lassocr*
unfolding *swap-lassocr-def* **by**(*rule pfinite-intro WT-intro*)+

lemma *pfinite-swap-rassocl* [*pfinite-intro*]: *pfinite-converter* (($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I} \mathcal{I}3$) (($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3$) $\oplus_\mathcal{I} \mathcal{I}2$) *swap-rassocl*
unfolding *swap-rassocl-def* **by**(*rule pfinite-intro WT-intro*)+

lemma *pfinite-parallel-wiring* [*pfinite-intro*]:
pfinite-converter (($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\oplus_\mathcal{I}$ ($\mathcal{I}3 \oplus_\mathcal{I} \mathcal{I}4$)) (($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}3$) $\oplus_\mathcal{I}$ ($\mathcal{I}2 \oplus_\mathcal{I} \mathcal{I}4$)) *parallel-wiring*
unfolding *parallel-wiring-def* **by**(*rule pfinite-intro WT-intro*)+

lemma *pfinite-parallel-converter* [*pfinite-intro*]:
pfinite-converter ($\mathcal{I}1 \oplus_\mathcal{I} \mathcal{I}2$) $\mathcal{I}3$ (*conv1* $|_\propto$ *conv2*)
if *pfinite-converter* $\mathcal{I}1$ $\mathcal{I}3$ *conv1* **and** *pfinite-converter* $\mathcal{I}2$ $\mathcal{I}3$ *conv2*
using *that* **by**(*coinduction arbitrary*: *conv1 conv2*)(*fastforce dest*: *pfinite-converterD*)

lemma *pfinite-converter-of-resource* [*simp*, *pfinite-intro*]: *pfinite-converter* $\mathcal{I}1$ $\mathcal{I}2$ (*converter-of-resource res*)
by(*coinduction arbitrary*: *res*) *auto*

3.3 colossless converter

coinductive *colossless-converter* :: ($'a$, $'b$) $\mathcal{I} \Rightarrow$ ($'c$, $'d$) $\mathcal{I} \Rightarrow$ ($'a$, $'b$, $'c$, $'d$) *converter* $\Rightarrow$ *bool*
for $\mathcal{I}$ $\mathcal{I}'$ **where**
colossless-converterI:
colossless-converter $\mathcal{I}$ $\mathcal{I}'$ *conv* **if**
$\bigwedge a.\ a \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *colossless-gpv* $\mathcal{I}'$ (*run-converter conv a*)
$\bigwedge a\ b\ conv'.$ $[\![\ a \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; (b, $conv'$) $\in$ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*) $]\!]$ $\Longrightarrow$ *colossless-converter* $\mathcal{I}$ $\mathcal{I}'$ $conv'$

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

lemma *colossless-converterD*:

⟦ *colossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*; *a* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧
⟹ *colossless-gpv* $\mathcal{I}'$ (*run-converter conv a*) ∧
(∀ (*b*, *conv*′) ∈ *results-gpv* $\mathcal{I}'$ (*run-converter conv a*). *colossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*′)
by(*auto elim*: *colossless-converter.cases*)

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

lemma *raw-converter-invariant-run-colossless-converter*: *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ *run-converter* (λ*conv*. *colossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* ∧ $\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *conv* √)
by(*unfold-locales*)(*auto dest*: *WT-converterD colossless-converterD*)

interpretation *run-colossless-converter*: *raw-converter-invariant*
$\mathcal{I}$ $\mathcal{I}'$ *run-converter* λ*conv*. *colossless-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv* ∧ $\mathcal{I}$,$\mathcal{I}'$ $\vdash_C$ *conv* √ **for** $\mathcal{I}$ $\mathcal{I}'$
by(*rule raw-converter-invariant-run-colossless-converter*)

lemma *colossless-const-converter* [*simp*]: *colossless-converter* $\mathcal{I}$ $\mathcal{I}'$ (*const-converter x*)
by(*coinduction*)(*auto*)

3.4 trace equivalence

lemma *distinguish-trace-eq*:
assumes *distinguish*: ⋀*distinguisher*. $\mathcal{I}$ ⊢*g distinguisher* √ ⟹ *connect distinguisher res* = *connect distinguisher res*′
shows *outs-*$\mathcal{I}$ $\mathcal{I}$ $\vdash_R$ *res* ≈ *res*′
using *assms* **by**(*rule distinguish-trace-eq*)(*auto intro*: *WT-fail-resource*)

lemma *attach-trace-eq*′:
assumes *eq*: *outs-*$\mathcal{I}$ $\mathcal{I}$ $\vdash_R$ *res1* ≈ *res2*
and *WT1* [*WT-intro*]: $\mathcal{I}$ ⊢*res res1* √
and *WT2* [*WT-intro*]: $\mathcal{I}$ ⊢*res res2* √
and *WT-conv* [*WT-intro*]: $\mathcal{I}'$,$\mathcal{I}$ $\vdash_C$ *conv* √
shows *outs-*$\mathcal{I}$ $\mathcal{I}'$ $\vdash_R$ *conv* ⊳ *res1* ≈ *conv* ⊳ *res2*
proof(*rule distinguish-trace-eq*)
fix $\mathcal{D}$ :: (′*c*, ′*d*) *distinguisher*
assume [*WT-intro*]: $\mathcal{I}'$ ⊢*g* $\mathcal{D}$ √
have *connect* (*absorb* $\mathcal{D}$ *conv*) *res1* = *connect* (*absorb* $\mathcal{D}$ *conv*) *res2* **using** *eq*
by(*rule connect-cong-trace*)(*rule WT-intro* | *fold WT-gpv-iff-outs-gpv*)+
then show *connect* $\mathcal{D}$ (*conv* ⊳ *res1*) = *connect* $\mathcal{D}$ (*conv* ⊳ *res2*) **by**(*simp add*: *distinguish-attach*)
qed

lemma *trace-callee-eq-trans* [*trans*]:
⟦ *trace-callee-eq callee1 callee2 A p q*; *trace-callee-eq callee2 callee3 A q r* ⟧
⟹ *trace-callee-eq callee1 callee3 A p r*
by(*simp add*: *trace-callee-eq-def*)

lemma *trace-eq′-parallel-resource*:
  fixes *res1* :: (′*a*, ′*b*) *resource* **and** *res2* :: (′*c*, ′*d*) *resource*
  assumes *1*: *trace-eq′ A res1 res1′*
    and *2*: *trace-eq′ B res2 res2′*
  shows *trace-eq′* (*A* <+> *B*) (*res1* ∥ *res2*) (*res1′* ∥ *res2′*)
proof −
  let *?ℐ* = *ℐ-uniform A* (*UNIV* :: ′*b set*) $\oplus_{\mathcal{I}}$ *ℐ-uniform B* (*UNIV* :: ′*d set*)
  have *trace-eq′* (*outs-ℐ ?ℐ*) (*res1* ∥ *res2*) (*res1′* ∥ *res2*)
   apply(*subst* (*1 2*) *attach-converter-of-resource-conv-parallel-resource2*[*symmetric*])
    apply(*rule attach-trace-eq′*[**where** *?ℐ* = *ℐ-uniform A UNIV*]; *auto simp add*: *1 intro*: *WT-intro WT-resource-ℐ-uniform-UNIV*)
    done
  also have *trace-eq′* (*outs-ℐ ?ℐ*) (*res1′* ∥ *res2*) (*res1′* ∥ *res2′*)
   apply(*subst* (*1 2*) *attach-converter-of-resource-conv-parallel-resource*[*symmetric*])
    apply(*rule attach-trace-eq′*[**where** *?ℐ* = *ℐ-uniform B UNIV*]; *auto simp add*: *2 intro*: *WT-intro WT-resource-ℐ-uniform-UNIV*)
    done
  finally show *?thesis* **by** *simp*
qed

proposition *trace-callee-eq-coinduct* [*consumes 1*, *case-names step 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 res′ b s′′ s′*. ⟦ *S p q*; *a* ∈ *A*; *res* ∈ *set-spmf p*; (*b*, *s′′*) ∈ *set-spmf* (*callee1 res 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(*rule trace-callee-eqI*)
  fix *xs* :: (′*a* × ′*b*) *list* **and** *z*
  assume *xs*: *set xs* ⊆ *A* × *UNIV* **and** *z*: *z* ∈ *A*

  from *start* **show** *trace-callee callee1 p xs z* = *trace-callee callee2 q xs z* **using** *xs*
  proof(*induction xs arbitrary*: *p q*)
    case *Nil*
    then show *?case* **using** *z* **by**(*simp add*: *step*)
  next
    case (*Cons xy xs*)
    obtain *x y* **where** *xy* [*simp*]: *xy* = (*x*, *y*) **by**(*cases xy*)
    have *trace-callee callee1 p* (*xy* # *xs*) *z* =
      trace-callee callee1 (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee1 s x*)) *y*) *xs z*
      by(*simp add*: *bind-map-spmf split-def o-def*)
    also have … = *trace-callee callee2* (*cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s x*)) *y*) *xs z*
    proof(*cases* ∃ *s* ∈ *set-spmf q*. ∃ *s′*. (*y*, *s′*) ∈ *set-spmf* (*callee2 s x*))

case *True*
then obtain s s' **where** *ss'*: $s \in$ *set-spmf q* $(y, s') \in$ *set-spmf* (*callee2 s x*)
by *blast*
from *Cons* **have** $x \in A$ **by** *simp*
from *ss' step*[*THEN arg-cong*[**where** *f=set-spmf*], *OF* ‹*S p q*› *this*] **obtain** *ss ss'*
where *ss* $\in$ *set-spmf p* (*y*, *ss'*) $\in$ *set-spmf* (*callee1 ss x*)
by(*clarsimp simp add*: *bind-UNION*) *force*
from *sim*[*OF* ‹*S p q*› - *this ss'*] **show** *?thesis* **using** *Cons.prems* **by** (*auto intro*: *Cons.IH*)
next
case *False*
then have *cond-spmf-fst* (*bind-spmf q* (λ*s*. *callee2 s x*)) *y* = *return-pmf None*
by(*auto simp add*: *bind-eq-return-pmf-None*)
moreover from *step*[*OF* ‹*S p q*›, *of x*, *THEN arg-cong*[**where** *f=set-spmf*], *THEN eq-refl*] *Cons.prems False*
have *cond-spmf-fst* (*bind-spmf p* (λ*s*. *callee1 s x*)) *y* = *return-pmf None*
by(*clarsimp simp add*: *bind-eq-return-pmf-None*)(*rule ccontr*; *fastforce*)
ultimately show *?thesis* **by**(*simp del*: *cond-spmf-fst-eq-return-None*)
qed
also have ... = *trace-callee callee2 q* (*xy* # *xs*) *z*
by(*simp add*: *split-def o-def*)
finally show *?case* **.**
qed
qed

proposition *trace-callee-eq-coinduct-strong* [*consumes 1*, *case-names step sim*, *case-conclusion step lhs rhs*, *case-conclusion sim sim eq*]:
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 res' b s'' s'*. ⟦ *S p q*; *a* ∈ *A*; *res* ∈ *set-spmf p*; (*b*, *s''*) ∈ *set-spmf* (*callee1 res 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*) ∨
trace-callee-eq callee1 callee2 A (*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 −
from *start* **have** *S p q* ∨ *trace-callee-eq callee1 callee2 A p q* **by** *simp*
thus *?thesis*
apply(*rule trace-callee-eq-coinduct*)
apply(*erule disjE*)
apply(*erule* (*1*) *step*)
apply(*drule trace-callee-eqD*[**where** *xs*=[]]; *simp*)
apply(*erule disjE*)
apply(*erule* (*5*) *sim*)

apply(*rule disjI2*)
apply(*rule trace-callee-eqI*)
apply(*drule trace-callee-eqD*[**where** *xs*=(-, -) # -])
apply *simp-all*
done
qed

lemma *trace-callee-return-pmf-None* [*simp*]:
trace-callee-eq callee1 callee2 A (*return-pmf None*) (*return-pmf None*)
by(*rule trace-callee-eqI*) *simp*

lemma *trace-callee-eq-sym* [*sym*]: *trace-callee-eq callee1 callee2 A p q* $\Longrightarrow$ *trace-callee-eq callee2 callee1 A q p*
by(*simp add*: *trace-callee-eq-def*)

lemma *eq-resource-on-imp-trace-eq*: $A \vdash_R$ *res1* $\approx$ *res2* **if** $A \vdash_R$ *res1* $\sim$ *res2*
proof −
have *outs-ℐ* (*ℐ-uniform A UNIV* :: ('*a*, '*b*) ℐ) $\vdash_R$ *res1* $\approx$ *res2* **using** *that*
by −(*rule distinguish-trace-eq*[*OF connect-eq-resource-cong*], *simp*+)
thus *?thesis* **by** *simp*
qed

lemma *advantage-nonneg*: *0* $\leq$ *advantage* $\mathcal{A}$ *res1 res2*
by(*simp add*: *advantage-def*)

lemma *comp-converter-of-resource-conv-parallel-converter*:
(*converter-of-resource res* $|_\propto$ 1_C) $\odot$ *conv* = *converter-of-resource res* $|_\propto$ *conv*
by(*coinduction arbitrary*: *res conv*)
(*auto 4 3 simp add*: *rel-fun-def gpv.map-comp map-lift-spmf spmf-rel-map split-def map-gpv-conv-bind*[*symmetric*] *id-def*[*symmetric*] *gpv.rel-map split*!: *sum.split intro*!: *rel-spmf-reflI gpv.rel-refl-strong*)

lemma *comp-converter-of-resource-conv-parallel-converter2*:
(1_C $|_\propto$ *converter-of-resource res*) $\odot$ *conv* = *conv* $|_\propto$ *converter-of-resource res*
by(*coinduction arbitrary*: *res conv*)
(*auto 4 3 simp add*: *rel-fun-def gpv.map-comp map-lift-spmf spmf-rel-map split-def map-gpv-conv-bind*[*symmetric*] *id-def*[*symmetric*] *gpv.rel-map split*!: *sum.split intro*!: *rel-spmf-reflI gpv.rel-refl-strong*)

lemma *parallel-converter-map-converter*:
map-converter f g f' g' conv1 $|_\propto$ *map-converter f'' g'' f' g' conv2* =
map-converter (*map-sum f f''*) (*map-sum g g''*) *f' g'* (*conv1* $|_\propto$ *conv2*)
using *parallel-callee-parametric*[
where *A*=*conversep* (*BNF-Def.Grp UNIV f*) **and** *B*=*BNF-Def.Grp UNIV g* **and** *C*=*BNF-Def.Grp UNIV f'* **and** *R*=*conversep* (*BNF-Def.Grp UNIV g'*) **and** *A'*=*conversep* (*BNF-Def.Grp UNIV f''*) **and** *B'*=*BNF-Def.Grp UNIV g''*,
unfolded rel-converter-Grp sum.rel-conversep sum.rel-Grp,
simplified,
unfolded rel-converter-Grp]

by(*simp add*: *rel-fun-def Grp-def*)

lemma *map-converter-parallel-converter-out2*:
conv1 $|_\propto$ *map-converter f g id id conv2* = *map-converter* (*map-sum id f*) (*map-sum id g*) *id id* (*conv1* $|_\propto$ *conv2*)
by(*rule parallel-converter-map-converter*[**where** *f*=*id* **and** *g*=*id* **and** *f'*=*id* **and** *g'*=*id*, *simplified*])

lemma *parallel-converter-assoc2*:
parallel-converter conv1 (*parallel-converter conv2 conv3*) =
map-converter lsumr rsuml id id (*parallel-converter* (*parallel-converter conv1 conv2*) *conv3*)
by(*coinduction arbitrary*: *conv1 conv2 conv3*)
(*auto 4 5 intro*!: *rel-funI gpv.rel-refl-strong split*: *sum.split simp add*: *gpv.rel-map map-gpv'-id map-gpv-conv-map-gpv'*[*symmetric*])

lemma *parallel-converter-of-resource*:
converter-of-resource res1 $|_\propto$ *converter-of-resource res2* = *converter-of-resource* (*res1* ∥ *res2*)
by(*coinduction arbitrary*: *res1 res2*)
(*auto 4 3 simp add*: *rel-fun-def map-lift-spmf spmf-rel-map intro*!: *rel-spmf-reflI split*!: *sum.split*)

lemma *map-Inr-parallel-converter*:
map-converter Inr f g h (*conv1* $|_\propto$ *conv2*) = *map-converter id* (*f* ∘ *Inr*) *g h conv2*
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *map-converter Inr f id id* (*map-converter id id g h conv1* $|_\propto$ *map-converter id id g h conv2*)
by(*simp add*: *parallel-converter-map-converter sum.map-id0*)
also have *map-converter Inr f id id* (*conv1* $|_\propto$ *conv2*) = *map-converter id* (*f* ∘ *Inr*) *id id conv2* **for** *conv1 conv2*
by(*coinduction arbitrary*: *conv2*)
(*auto simp add*: *rel-fun-def map-gpv-conv-map-gpv'*[*symmetric*] *gpv.rel-map intro*!: *gpv.rel-refl-strong*)
also have *map-converter id* (*f* ∘ *Inr*) *id id* (*map-converter id id g h conv2*) = *?rhs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *map-Inl-parallel-converter*:
map-converter Inl f g h (*conv1* $|_\propto$ *conv2*) = *map-converter id* (*f* ∘ *Inl*) *g h conv1*
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *map-converter Inl f id id* (*map-converter id id g h conv1* $|_\propto$ *map-converter id id g h conv2*)
by(*simp add*: *parallel-converter-map-converter sum.map-id0*)
also have *map-converter Inl f id id* (*conv1* $|_\propto$ *conv2*) = *map-converter id* (*f* ∘ *Inl*) *id id conv1* **for** *conv1 conv2*

by(*coinduction arbitrary*: *conv1*)
(*auto simp add*: *rel-fun-def map-gpv-conv-map-gpv′*[*symmetric*] *gpv.rel-map intro*!: *gpv.rel-refl-strong*)
also have *map-converter id* ($f \circ$ *Inl*) *id id* (*map-converter id id g h conv1*) = *?rhs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *left-interface-parallel-converter*:
left-interface (*conv1* $|_\propto$ *conv2*) = *left-interface conv1* $|_\propto$ *left-interface conv2*
by(*coinduction arbitrary*: *conv1 conv2*)
(*auto 4 3 split*!: *sum.split simp add*: *rel-fun-def gpv.rel-map left-gpv-map*[**where** *h*=*id*] *sum.map-id0 intro*!: *gpv.rel-refl-strong*)

lemma *right-interface-parallel-converter*:
right-interface (*conv1* $|_\propto$ *conv2*) = *right-interface conv1* $|_\propto$ *right-interface conv2*
by(*coinduction arbitrary*: *conv1 conv2*)
(*auto 4 3 split*!: *sum.split simp add*: *rel-fun-def gpv.rel-map right-gpv-map*[**where** *h*=*id*] *sum.map-id0 intro*!: *gpv.rel-refl-strong*)

lemma *left-interface-converter-of-resource* [*simp*]:
left-interface (*converter-of-resource res*) = *converter-of-resource res*
by(*coinduction arbitrary*: *res*)(*auto simp add*: *rel-fun-def map-lift-spmf spmf-rel-map intro*!: *rel-spmf-reflI*)

lemma *right-interface-converter-of-resource* [*simp*]:
right-interface (*converter-of-resource res*) = *converter-of-resource res*
by(*coinduction arbitrary*: *res*)(*auto simp add*: *rel-fun-def map-lift-spmf spmf-rel-map intro*!: *rel-spmf-reflI*)

lemma *parallel-converter-swap*: *map-converter swap-sum swap-sum id id* (*conv1* $|_\propto$ *conv2*) = *conv2* $|_\propto$ *conv1*
by(*coinduction arbitrary*: *conv1 conv2*)
(*auto 4 3 split*!: *sum.split simp add*: *rel-fun-def map-gpv-conv-map-gpv′*[*symmetric*] *gpv.rel-map intro*!: *gpv.rel-refl-strong*)

lemma *eq-$\mathcal{I}$-converter-map-converter′*:
assumes $\mathcal{I}''$, *map-$\mathcal{I}$* f' g' $\mathcal{I}'$ $\vdash_C$ *conv1* $\sim$ *conv2*
and f ‘ *outs-$\mathcal{I}$* $\mathcal{I} \subseteq$ *outs-$\mathcal{I}$* $\mathcal{I}''$
and $\forall$ $q \in$*outs-$\mathcal{I}$* $\mathcal{I}$. g ‘ *responses-$\mathcal{I}$* $\mathcal{I}''$ (f q) $\subseteq$ *responses-$\mathcal{I}$* $\mathcal{I}$ q
shows $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *map-converter f g f′ g′ conv1* $\sim$ *map-converter f g f′ g′ conv2*
using *assms*(*1*)
proof(*coinduction arbitrary*: *conv1 conv2*)
case *eq-$\mathcal{I}$-converter*
from *this*(*2*) **have** f $q \in$ *outs-$\mathcal{I}$* $\mathcal{I}''$ **using** *assms*(*2*) **by** *auto*
from *eq-$\mathcal{I}$-converter*(*1*)[*THEN eq-$\mathcal{I}$-converterD*, *OF this*] *eq-$\mathcal{I}$-converter*(*2*)
show *?case*
apply *simp*
apply(*rule eq-$\mathcal{I}$-gpv-map-gpv′*)

apply(*simp add*: *BNF-Def.vimage2p-def prod.rel-map*)
apply(*erule eq-$\mathcal{I}$-gpv-mono'*)
using *assms*(*3*)
apply(*auto 4 4 simp add*: *eq-onp-def*)
done
qed

lemma *parallel-converter-eq-$\mathcal{I}$-cong*:
⟦ $\mathcal{I}$*1*, $\mathcal{I}$ $\vdash_C$ *conv1* ∼ *conv1'*; $\mathcal{I}$*2*, $\mathcal{I}$ $\vdash_C$ *conv2* ∼ *conv2'* ⟧
⟹ $\mathcal{I}$*1* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*2*, $\mathcal{I}$ $\vdash_C$ *parallel-converter conv1 conv2* ∼ *parallel-converter conv1' conv2'*
by(*coinduction arbitrary*: *conv1 conv2 conv1' conv2'*)
(*fastforce dest*: *eq-$\mathcal{I}$-converterD elim*!: *eq-$\mathcal{I}$-gpv-mono' simp add*: *eq-onp-def*)

— Helper lemmas for simplyfing *exec-gpv*
lemma
exec-gpv-parallel-oracle-right:
exec-gpv (*oracle1* $\ddagger_O$ *oracle2*) (*right-gpv gpv*) *s* = *exec-gpv* (†*oracle2*) *gpv s*
unfolding *spmf-rel-eq*[*symmetric*]
apply (*rule rel-spmf-mono*)
by (*rule exec-gpv-parametric'*[**where** *S*=(=) **and** *A*=(=) **and** *CALL*=λ*l r*. *l* = *Inr r* **and** *R*=λ*l r*. *l* = *Inr r* , *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
(*auto simp add*: *prod.rel-eq extend-state-oracle-def parallel-oracle-def split-def spmf-rel-map1 spmf-rel-map2 map-prod-def rel-spmf-reflI right-gpv-Inr-transfer intro*!:*rel-funI*)

lemma
exec-gpv-parallel-oracle-left:
exec-gpv (*oracle1* $\ddagger_O$ *oracle2*) (*left-gpv gpv*) *s* = *exec-gpv* (*oracle1*†) *gpv s* (**is** *?L* = *?R*)
unfolding *spmf-rel-eq*[*symmetric*]
apply (*rule rel-spmf-mono*)
by (*rule exec-gpv-parametric'*[**where** *S*=(=) **and** *A*=(=) **and** *CALL*=λ*l r*. *l* = *Inl r* **and** *R*=λ*l r*. *l* = *Inl r* , *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
(*auto simp add*: *prod.rel-eq extend-state-oracle2-def parallel-oracle-def split-def spmf-rel-map1 spmf-rel-map2 map-prod-def rel-spmf-reflI left-gpv-Inl-transfer intro*!:*rel-funI*)

end
theory *Observe-Failure* **imports**
More-CC
begin

declare [[*show-variants*]]

context fixes *oracle* :: (*'s*, *'in*, *'out*) *oracle'* **begin**

fun *obsf-oracle* :: (*'s exception*, *'in*, *'out exception*) *oracle'* **where**
obsf-oracle Fault x = *return-spmf* (*Fault*, *Fault*)

| *obsf-oracle* (*OK s*) *x* = *TRY map-spmf* (*map-prod OK OK*) (*oracle s x*) *ELSE return-spmf* (*Fault*, *Fault*)

end

type-synonym ('*a*, '*b*) *resource-obsf* = ('*a*, '*b exception*) *resource*

translations
(*type*) ('*a*, '*b*) *resource-obsf* <= (*type*) ('*a*, '*b exception*) *resource*

primcorec *obsf-resource* :: ('*in*, '*out*) *resource* ⇒ ('*in*, '*out*) *resource-obsf* **where**
run-resource (*obsf-resource res*) = (λ*x*.
map-spmf (*map-prod id obsf-resource*)
(*map-spmf* (*map-prod id* (λ*resF*. *case resF of OK res*' ⇒ *res*' | *Fault* ⇒ *fail-resource*))
(*TRY map-spmf* (*map-prod OK OK*) (*run-resource res x*) *ELSE return-spmf* (*Fault*, *Fault*))))

lemma *obsf-resource-sel*:
run-resource (*obsf-resource res*) *x* =
map-spmf (*map-prod id* (λ*resF*. *obsf-resource* (*case resF of OK res*' ⇒ *res*' | *Fault* ⇒ *fail-resource*)))
(*TRY map-spmf* (*map-prod OK OK*) (*run-resource res x*) *ELSE return-spmf* (*Fault*, *Fault*))
by(*simp add*: *spmf.map-comp prod.map-comp o-def id-def*)

declare *obsf-resource.simps* [*simp del*]

lemma *obsf-resource-exception* [*simp*]: *obsf-resource fail-resource* = *const-resource Fault*
by *coinduction*(*simp add*: *rel-fun-def obsf-resource-sel*)

lemma *obsf-resource-sel2* [*simp*]:
run-resource (*obsf-resource res*) *x* =
try-spmf (*map-spmf* (*map-prod OK obsf-resource*) (*run-resource res x*)) (*return-spmf* (*Fault*, *const-resource Fault*))
by(*simp add*: *map-try-spmf spmf.map-comp o-def prod.map-comp obsf-resource-sel*)

lemma *lossless-obsf-resource* [*simp*]: *lossless-resource* ℐ (*obsf-resource res*)
by(*coinduction arbitrary*: *res*) *auto*

lemma *WT-obsf-resource* [*WT-intro*, *simp*]: *exception-*ℐ ℐ ⊢*res obsf-resource res* √ **if** ℐ ⊢*res res* √
using *that* **by**(*coinduction arbitrary*: *res*)(*auto dest*: *WT-resourceD split*: *if-split-asm*)

type-synonym ('*a*, '*b*) *distinguisher-obsf* = (*bool*, '*a*, '*b exception*) *gpv*

translations

(*type*) ('*a*, '*b*) *distinguisher-obsf* <= (*type*) (*bool*, '*a*, '*b exception*) *gpv*

abbreviation *connect-obsf* :: ('*a*, '*b*) *distinguisher-obsf* ⇒ ('*a*, '*b*) *resource-obsf* ⇒ *bool spmf* **where**
connect-obsf == *connect*

definition *obsf-distinguisher* :: ('*a*, '*b*) *distinguisher* ⇒ ('*a*, '*b*) *distinguisher-obsf* **where**
obsf-distinguisher $\mathcal{D}$ = *map-gpv'* (λ*x*. *x* = *Some True*) *id option-of-exception* (*gpv-stop* $\mathcal{D}$)

lemma *WT-obsf-distinguisher* [*WT-intro*]:
exception-$\mathcal{I}$ $\mathcal{I}$ ⊢*g obsf-distinguisher* $\mathcal{A}$ √ **if** [*WT-intro*]: $\mathcal{I}$ ⊢*g* $\mathcal{A}$ √
unfolding *obsf-distinguisher-def* **by**(*rule WT-intro*|*simp*)+

lemma *interaction-bounded-by-obsf-distinguisher* [*interaction-bound*]:
interaction-bounded-by consider (*obsf-distinguisher* $\mathcal{A}$) *bound*
if [*interaction-bound*]: *interaction-bounded-by consider* $\mathcal{A}$ *bound*
unfolding *obsf-distinguisher-def* **by**(*rule interaction-bound*|*simp*)+

lemma *plossless-obsf-distinguisher* [*simp*]:
plossless-gpv (*exception-*$\mathcal{I}$ $\mathcal{I}$) (*obsf-distinguisher* $\mathcal{A}$)
if *plossless-gpv* $\mathcal{I}$ $\mathcal{A}$ $\mathcal{I}$ ⊢*g* $\mathcal{A}$ √
using *that* **unfolding** *obsf-distinguisher-def* **by**(*simp*)

type-synonym ('*a*, '*b*, '*c*, '*d*) *converter-obsf* = ('*a*, '*b exception*, '*c*, '*d exception*) *converter*

translations
(*type*) ('*a*, '*b*, '*c*, '*d*) *converter-obsf* <= (*type*) ('*a*, '*b exception*, '*c*, '*d exception*) *converter*

primcorec *obsf-converter* :: ('*a*, '*b*, '*c*, '*d*) *converter* ⇒ ('*a*, '*b*, '*c*, '*d*) *converter-obsf* **where**
run-converter (*obsf-converter conv*) = (λ*x*.
map-gpv (*map-prod id obsf-converter*) *id*
(*map-gpv* (λ*convF*. *case convF of Fault* ⇒ (*Fault*, *fail-converter*) | *OK* (*a*, *conv'*) ⇒ (*OK a*, *conv'*)) *id*
(*try-gpv* (*map-gpv' exception-of-option id option-of-exception* (*gpv-stop* (*run-converter conv x*))) (*Done Fault*))))

lemma *obsf-converter-exception* [*simp*]: *obsf-converter fail-converter* = *const-converter Fault*
by(*coinduction*)(*simp add*: *rel-fun-def*)

lemma *obsf-converter-sel* [*simp*]:
run-converter (*obsf-converter conv*) *x* =
TRY map-gpv' (λ*y*. *case y of None* ⇒ (*Fault*, *const-converter Fault*) | *Some*(*x*,

conv′) ⇒ (*OK x*, *obsf-converter conv′*)) *id option-of-exception*
(*gpv-stop* (*run-converter conv x*))
ELSE Done (*Fault*, *const-converter Fault*)
by(*simp add*: *map-try-gpv*)
(*simp add*: *map-gpv-conv-map-gpv′ map-gpv′-comp o-def option.case-distrib*[**where** *h*=*map-prod - -*] *split-def id-def cong del*: *option.case-cong*)

declare *obsf-converter.sel* [*simp del*]

lemma *exec-gpv-obsf-resource*:
defines *exec-gpv1* ≡ *exec-gpv*
and *exec-gpv2* ≡ *exec-gpv*
shows
exec-gpv1 run-resource (*map-gpv′ id id option-of-exception* (*gpv-stop gpv*)) (*obsf-resource res*) ↿ {(*Some x*, *y*)|*x y*. *True*} =
map-spmf (*map-prod Some obsf-resource*) (*exec-gpv2 run-resource gpv res*)
(**is** *?lhs* = *?rhs*)
proof(*rule spmf.leq-antisym*)
show *ord-spmf* (=) *?lhs ?rhs* **unfolding** *exec-gpv1-def*
proof(*induction arbitrary*: *gpv res rule*: *exec-gpv-fixp-induct-strong*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv′*)
show *?case* **unfolding** *exec-gpv2-def*
apply(*subst exec-gpv.simps*)
apply(*clarsimp simp add*: *map-bind-spmf bind-map-spmf restrict-bind-spmf o-def try-spmf-def intro*!: *ord-spmf-bind-reflI split*!: *generat.split*)
apply(*clarsimp simp add*: *bind-map-pmf bind-spmf-def bind-assoc-pmf bind-return-pmf spmf.leq-trans*[*OF restrict-spmf-mono*, *OF step.hyps*] *id-def step.IH*[*simplified*, *simplified id-def exec-gpv2-def*] *intro*!: *rel-pmf-bind-reflI split*!: *option.split*)
done
qed

show *ord-spmf* (=) *?rhs ?lhs* **unfolding** *exec-gpv2-def*
proof(*induction arbitrary*: *gpv res rule*: *exec-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv′*)
show *?case* **unfolding** *exec-gpv1-def*
apply(*subst exec-gpv.simps*)
apply(*clarsimp simp add*: *bind-map-spmf map-bind-spmf restrict-bind-spmf o-def try-spmf-def intro*!: *ord-spmf-bind-reflI split*!: *generat.split*)
apply(*clarsimp simp add*: *bind-spmf-def bind-assoc-pmf bind-map-pmf map-bind-pmf bind-return-pmf id-def step.IH*[*simplified*, *simplified id-def exec-gpv1-def*] *intro*!: *rel-pmf-bind-reflI split*!: *option.split*)
done
qed
qed

lemma *obsf-attach*:
assumes *pfinite*: *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv*
and *WT*: $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* $\surd$
and *WT-resource*: $\mathcal{I}'$ $\vdash$*res res* $\surd$
shows *outs-*$\mathcal{I}$ $\mathcal{I}$ $\vdash_R$ *attach* (*obsf-converter conv*) (*obsf-resource res*) $\sim$ *obsf-resource* (*attach conv res*)
using *assms*
proof(*coinduction arbitrary*: *conv res*)
case (*eq-resource-on out conv res*)
then show *?case* (**is** *rel-spmf ?X ?lhs ?rhs*)
proof −
have *?lhs* = *map-spmf* (λ((*b*, *conv'*), *res'*). (*b*, *conv'* ⊳ *res'*))
(*exec-gpv run-resource*
(*TRY map-gpv'* (*case-option* (*Fault*, *const-converter Fault*) (λ(*x*, *conv'*). (*OK* *x*, *obsf-converter conv'*))) *id option-of-exception* (*gpv-stop* (*run-converter conv out*)) *ELSE Done* (*Fault*, *const-converter Fault*))
(*obsf-resource res*))
(**is** - = *map-spmf ?attach* (*exec-gpv* - (*TRY ?gpv ELSE* -) -)) **by**(*clarsimp*)
also have ... = *TRY map-spmf ?attach* (*exec-gpv run-resource ?gpv* (*obsf-resource res*)) *ELSE return-spmf* (*Fault*, *const-resource Fault*)
by(*rule run-lossless-resource.exec-gpv-try-gpv*[**where** $\mathcal{I}$=*exception-*$\mathcal{I}$ $\mathcal{I}'$])
(*use eq-resource-on* **in** ‹*auto intro*!: *WT-gpv-map-gpv' WT-gpv-stop pfinite-gpv-stop*[*THEN iffD2*] *dest*: *WT-converterD pfinite-converterD lossless-resourceD*›)
also have ... = *TRY map-spmf* (λ(*rc*, *res'*). *case rc of None* ⇒ (*Fault*, *const-resource Fault*) | *Some* (*x*, *conv'*) ⇒ (*OK x*, *obsf-converter conv'* ⊳ *res'*))
((*exec-gpv run-resource* (*map-gpv' id id option-of-exception* (*gpv-stop* (*run-converter conv out*))) (*obsf-resource res*)) ↾ {(*Some x*, *y*)|*x y*. *True*})
ELSE return-spmf (*Fault*, *const-resource Fault*) (**is** - = *TRY map-spmf* *?f* - *ELSE ?z*)
by(*subst map-gpv'-id12*)(*clarsimp simp add*: *map-gpv'-map-gpv-swap exec-gpv-map-gpv-id try-spmf-def restrict-spmf-def bind-map-pmf intro*!: *bind-pmf-cong*[*OF refl*] *split*: *option.split*)
also have ... = *TRY map-spmf ?f* (*map-spmf* (*map-prod Some obsf-resource*) (*exec-gpv run-resource* (*run-converter conv out*) *res*)) *ELSE ?z*
by(*simp only*: *exec-gpv-obsf-resource*)
also have *rel-spmf ?X* ... *?rhs* **using** *eq-resource-on*
by(*auto simp add*: *spmf.map-comp o-def spmf-rel-map intro*!: *rel-spmf-try-spmf rel-spmf-reflI*)
(*auto intro*!: *exI dest*: *run-resource.in-set-spmf-exec-gpv-into-results-gpv WT-converterD pfinite-converterD run-resource.exec-gpv-invariant*)
finally show *?case* **.**
qed
qed

lemma *colossless-obsf-converter* [*simp*]:
colossless-converter (*exception-*$\mathcal{I}$ $\mathcal{I}$) $\mathcal{I}'$ (*obsf-converter conv*)
by(*coinduction arbitrary*: *conv*)(*auto split*: *option.split-asm*)

lemma *WT-obsf-converter* [*WT-intro*]:
exception-$\mathcal{I}$ $\mathcal{I}$, *exception-$\mathcal{I}$ $\mathcal{I}'$* $\vdash_C$ *obsf-converter conv* $\surd$ **if** $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv* $\surd$
using *that*
by(*coinduction arbitrary*: *conv*)(*auto 4 3 dest*: *WT-converterD results-gpv-stop-SomeD split*!: *option.splits intro*!: *WT-intro*)

lemma *inline1-gpv-stop-obsf-converter*:
defines *inline1a* ≡ *inline1*
and *inline1b* ≡ *inline1*
shows *bind-spmf* (*inline1a run-converter* (*map-gpv′ id id option-of-exception* (*gpv-stop gpv*)) (*obsf-converter conv*))
(*λxy. case xy of Inl* (*None*, *conv′*) ⇒ *return-pmf None* | *Inl* (*Some x*, *conv′*) ⇒ *return-spmf* (*Inl* (*x*, *conv′*)) | *Inr y* ⇒ *return-spmf* (*Inr y*)) =
map-spmf (*map-sum* (*apsnd obsf-converter*)
(*apsnd* (*map-prod* (*λrpv input. case input of Fault* ⇒ *Done* (*Fault*, *const-converter Fault*) | *OK input′* ⇒
map-gpv′ (*λres. case res of None* ⇒ (*Fault*, *const-converter Fault*) | *Some* (*x*, *conv′*) ⇒ (*OK x*, *obsf-converter conv′*)) *id option-of-exception* (*try-gpv* (*gpv-stop* (*rpv input′*)) (*Done None*)))
(*λrpv input. case input of Fault* ⇒ *Done None* | *OK input′* ⇒ *map-gpv′ id id option-of-exception* (*gpv-stop* (*rpv input′*))))))
(*inline1b run-converter gpv conv*)
(**is** *?lhs* = *?rhs*)
proof(*rule spmf.leq-antisym*)
show *ord-spmf* (=) *?lhs ?rhs* **unfolding** *inline1a-def*
proof(*induction arbitrary*: *gpv conv rule*: *inline1-fixp-induct-strong*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1′*)
show *?case* **unfolding** *inline1b-def*
apply(*subst inline1-unfold*)
apply(*clarsimp simp add*: *map-spmf-bind-spmf bind-map-spmf spmf.map-comp o-def generat.map-comp intro*!: *ord-spmf-bind-reflI split*!: *generat.split*)
apply(*clarsimp simp add*: *bind-spmf-def try-spmf-def bind-assoc-pmf bind-map-pmf bind-return-pmf intro*!: *rel-pmf-bind-reflI split*!: *option.split*)
subgoal unfolding *bind-spmf-def*[*symmetric*]
by(*rule ord-spmf-bindI*[*OF step.hyps*, *THEN spmf.leq-trans*])
(*auto split*!: *option.split intro*!: *ord-spmf-bindI*[*OF step.hyps*, *THEN spmf.leq-trans*] *ord-spmf-reflI*)
subgoal unfolding *bind-spmf-def*[*symmetric*]
by(*clarsimp simp add*: *in-set-spmf*[*symmetric*] *inline1b-def split*!: *generat.split intro*!: *step.IH*[*THEN spmf.leq-trans*])
(*auto simp add*: *fun-eq-iff map′-try-gpv split*: *exception.split*)
done
qed

show *ord-spmf* (=) *?rhs ?lhs* **unfolding** *inline1b-def*
proof(*induction arbitrary*: *gpv conv rule*: *inline1-fixp-induct-strong*)

case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1'*)
show *?case* **unfolding** *inline1a-def*
apply(*subst inline1-unfold*)
apply(*clarsimp simp add*: *map-spmf-bind-spmf bind-map-spmf spmf.map-comp o-def generat.map-comp intro*!: *ord-spmf-bind-reflI split*!: *generat.split*)
apply(*clarsimp simp add*: *bind-spmf-def try-spmf-def bind-assoc-pmf bind-map-pmf bind-return-pmf intro*!: *rel-pmf-bind-reflI split*!: *option.split*)
apply(*clarsimp simp add*: *bind-spmf-def*[*symmetric*] *in-set-spmf*[*symmetric*] *inline1a-def id-def*[*symmetric*] *split*!: *generat.split intro*!: *step.IH*[*THEN spmf.leq-trans*])
apply(*auto simp add*: *fun-eq-iff map'-try-gpv split*: *exception.split*)
done
qed
qed

lemma *inline-gpv-stop-obsf-converter*:
bind-gpv (*inline run-converter* (*map-gpv' id id option-of-exception* (*gpv-stop gpv*)) (*obsf-converter conv*)) (λ(*x*, *conv'*). *case x of None* ⇒ *Fail* | *Some x'* ⇒ *Done* (*x*, *conv'*)) =
bind-gpv (*map-gpv' id id option-of-exception* (*gpv-stop* (*inline run-converter gpv conv*))) (λ*x*. *case x of None* ⇒ *Fail* | *Some* (*x'*, *conv*) ⇒ *Done* (*Some x'*, *obsf-converter conv*))
proof(*coinduction arbitrary*: *gpv conv rule*: *gpv-coinduct-bind*)
case (*Eq-gpv gpv conv*)
show *?case TYPE*(*'c* × (*'b*, *'c*, *'d*, *'e*) *converter*) *TYPE*(*'c* × (*'b*, *'c*, *'d*, *'e*) *converter*) (**is** *rel-spmf ?X ?lhs ?rhs*)
proof −
have *?lhs* = *map-spmf* (λ*xyz*. *case xyz of Inl* (*x*, *conv*) ⇒ *Pure* (*Some x*, *conv*) | *Inr* (*out*, *rpv*, *rpv'*) ⇒ *IO out* (λ*input*. *bind-gpv* (*bind-gpv* (*rpv input*) (λ(*x*, *y*). *inline run-converter* (*rpv' x*) *y*)) (λ(*x*, *conv'*). *case x of None* ⇒ *Fail* | *Some x'* ⇒ *Done* (*x*, *conv'*))))
(*bind-spmf* (*inline1 run-converter* (*map-gpv' id id option-of-exception* (*gpv-stop gpv*)) (*obsf-converter conv*))
(λ*xy*. *case xy of Inl* (*None*, *conv'*) ⇒ *return-pmf None* | *Inl* (*Some x*, *conv'*) ⇒ *return-spmf* (*Inl* (*x*, *conv'*)) | *Inr y* ⇒ *return-spmf* (*Inr y*)))
(**is** - = *map-spmf ?f* -)
by(*auto simp del*: *bind-gpv-sel' simp add*: *bind-gpv.sel map-bind-spmf inline-sel bind-map-spmf o-def intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *sum.split option.split*)
also have ... = *map-spmf ?f* (*map-spmf* (*map-sum* (*apsnd obsf-converter*) (*apsnd* (*map-prod* (λ*rpv*. *case-exception* (*Done* (*Fault*, *const-converter Fault*))
(λ*input'*. *map-gpv'* (*case-option* (*Fault*, *const-converter Fault*) (λ(*x*, *conv'*). (*OK x*, *obsf-converter conv'*))) *id option-of-exception* (*TRY gpv-stop* (*rpv input'*) *ELSE Done None*)))
(λ*rpv*. *case-exception* (*Done None*) (λ*input'*. *map-gpv' id id option-of-exception* (*gpv-stop* (*rpv input'*)))))))
(*inline1 run-converter gpv conv*))
by(*simp only*: *inline1-gpv-stop-obsf-converter*)
also have ... = *bind-spmf* (*inline1 run-converter gpv conv*) (λ*y*. *return-spmf*

(*?f* (*map-sum* (*apsnd obsf-converter*)
(*apsnd* (*map-prod* (*λrpv. case-exception* (*Done* (*Fault*, *const-converter Fault*)) (*λinput′. map-gpv′* (*case-option* (*Fault*, *const-converter Fault*) (*λ*(*x*, *conv′*). (*OK x*, *obsf-converter conv′*))) *id option-of-exception* (*TRY gpv-stop* (*rpv input′*) *ELSE Done None*)))
(*λrpv. case-exception* (*Done None*) (*λinput′. map-gpv′ id id option-of-exception* (*gpv-stop* (*rpv input′*))))))
y)))
by(*simp add*: *map-spmf-conv-bind-spmf*)
also have *rel-spmf ?X* ... (*bind-spmf* (*inline1 run-converter gpv conv*)
(*λx. map-spmf* (*map-generat id id* ((∘) (*case-sum id* (*λr. bind-gpv r* (*case-option Fail* (*λ*(*x′*, *conv*). *Done* (*Some x′*, *obsf-converter conv*)))))))
(*case map-generat id id* (*map-fun option-of-exception* (*map-gpv′ id id option-of-exception*))
(*map-generat Some id* (*λrpv. case-option* (*Done None*) (*λinput′. gpv-stop* (*rpv input′*)))
(*case x of Inl x* ⇒ *Pure x*
| *Inr* (*out*, *oracle*, *rpv*) ⇒ *IO out* (*λinput. bind-gpv* (*oracle input*) (*λ*(*x*, *y*). *inline run-converter* (*rpv x*) *y*)))) *of*
Pure x ⇒
map-spmf (*map-generat id id* ((∘) *Inl*)) (*the-gpv* (*case x of None* ⇒ *Fail* | *Some* (*x′*, *conv*) ⇒ *Done* (*Some x′*, *obsf-converter conv*)))
| *IO out c* ⇒ *return-spmf* (*IO out* (*λinput. Inr* (*c input*))))))
(**is** *rel-spmf - - ?rhs2* **is** *rel-spmf -* (*bind-spmf - ?L*) (*bind-spmf - ?R*))
proof(*rule rel-spmf-bind-reflI*)
fix *x* :: *′a* × (*′b*, *′c*, *′d*, *′e*) *converter* + *′d* × (*′c* × (*′b*, *′c*, *′d*, *′e*) *converter*, *′d*, *′e*) *rpv* × (*′a*, *′b*, *′c*) *rpv*
assume *x*: *x* ∈ *set-spmf* (*inline1 run-converter gpv conv*)
consider (*Inl*) *a conv′* **where** *x* = *Inl* (*a*, *conv′*) | (*Inr*) *out rpv rpv′* **where** *x* = *Inr* (*out*, *rpv*, *rpv′*) **by**(*cases x*) *auto*
then show *rel-spmf ?X* (*?L x*) (*?R x*)
proof *cases*
case *Inr*
have ∃(*gpv2* :: (*′c* × (*′b*, *′c*, *′d*, *′e*) *converter*, *′d*, *′e exception*) *gpv*) (*gpv2′* :: (*′c* × (*′b*, *′c*, *′d*, *′e*) *converter*, *′d*, *′e exception*) *gpv*) *f f′*.
bind-gpv (*map-gpv′* (*case-option* (*Fault*, *const-converter Fault*) (*λp.* (*OK* (*fst p*), *obsf-converter* (*snd p*)))) *id option-of-exception* (*TRY gpv-stop* (*rpv input*) *ELSE Done None*))
(*λx. case fst x of Fault* ⇒ *Fail* | *OK xa* ⇒ *bind-gpv* (*inline run-converter* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*rpv′ xa*))) (*snd x*)) (*λp. case fst p of None* ⇒ *Fail* | *Some x′* ⇒ *Done* (*Some x′*, *snd p*))) =
bind-gpv gpv2 f ∧
bind-gpv (*map-gpv′ id id option-of-exception* (*gpv-stop* (*rpv input*))) (*case-option Fail* (*λx. bind-gpv* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*inline run-converter* (*rpv′* (*fst x*)) (*snd x*)))) (*case-option Fail* (*λp. Done* (*Some* (*fst p*), *obsf-converter* (*snd p*)))))) =
bind-gpv gpv2′ f′ ∧
rel-gpv (*λx y.* ∃ *gpv conv. f x* = *bind-gpv* (*inline run-converter* (*map-gpv′ id id option-of-exception* (*gpv-stop gpv*)) (*obsf-converter conv*)) (*λp. case fst p of*

None ⇒ *Fail* | *Some x′* ⇒ *Done* (*Some x′*, *snd p*)) ∧
f′ y = *bind-gpv* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*inline run-converter gpv conv*))) (*case-option Fail* (λ*p*. *Done* (*Some* (*fst p*), *obsf-converter* (*snd p*)))))
(=) *gpv2 gpv2′*
(**is** ∃ *gpv2 gpv2′ f f′*. *?lhs1 input* = - ∧ *?rhs1 input* = - ∧ *rel-gpv* (*?X f f′*) - - -) **for** *input*
proof(*intro exI conjI*)
let *?gpv2* = *bind-gpv* (*map-gpv′ id id option-of-exception* (*TRY gpv-stop* (*rpv input*) *ELSE Done None*)) (λ*x*. *case x of None* ⇒ *Fail* | *Some x* ⇒ *Done x*)
let *?gpv2′* = *bind-gpv* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*rpv input*))) (λ*x*. *case x of None* ⇒ *Fail* | *Some x* ⇒ *Done x*)
let *?f* = λ*x*. *bind-gpv* (*inline run-converter* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*rpv′* (*fst x*)))) (*obsf-converter* (*snd x*))) (λ*p*. *case fst p of None* ⇒ *Fail* | *Some x′* ⇒ *Done* (*Some x′*, *snd p*))
let *?f′* = λ*x*. *bind-gpv* (*map-gpv′ id id option-of-exception* (*gpv-stop* (*inline run-converter* (*rpv′* (*fst x*)) (*snd x*)))) (*case-option Fail* (λ*p*. *Done* (*Some* (*fst p*), *obsf-converter* (*snd p*))))
show *?lhs1 input* = *bind-gpv ?gpv2 ?f*
by(*subst map-gpv′-id12*[*THEN trans*, *OF map-gpv′-map-gpv-swap*])
(*auto simp add*: *bind-map-gpv o-def bind-gpv-assoc intro*!: *bind-gpv-cong split*!: *option.split*)
show *?rhs1 input* = *bind-gpv ?gpv2′ ?f′*
by(*auto simp add*: *bind-gpv-assoc id-def*[*symmetric*] *intro*!: *bind-gpv-cong split*!: *option.split*)
show *rel-gpv* (*?X ?f ?f′*) (=) *?gpv2 ?gpv2′* **using** *Inr x*
by(*auto simp add*: *map′-try-gpv id-def*[*symmetric*] *bind-try-Done-Fail intro*: *gpv.rel-refl-strong*)
qed
then show *?thesis* **using** *Inr*
by(*clarsimp split*!: *sum.split exception.split simp add*: *rel-fun-def bind-gpv-assoc split-def map-gpv′-bind-gpv exception.case-distrib*[**where** *h*=λ*x*. *bind-gpv* (*inline - x -*) -] *option.case-distrib*[**where** *h*=λ*x*. *bind-gpv* (*map-gpv′* - - - *x*) -] *cong*: *exception.case-cong option.case-cong*)
qed *simp*
qed
moreover have *?rhs2* = *?rhs*
by(*simp del*: *bind-gpv-sel′ add*: *bind-gpv.sel map-bind-spmf inline-sel bind-map-spmf o-def*)
ultimately show *?thesis* **by**(*simp only*:)
qed
qed

lemma *obsf-comp-converter*:
assumes *WT*: $\mathcal{I}$, $\mathcal{I}'$ $\vdash_C$ *conv1* √ $\mathcal{I}'$, $\mathcal{I}''$ $\vdash_C$ *conv2* √
and *pfinite1*: *pfinite-converter* $\mathcal{I}$ $\mathcal{I}'$ *conv1*
shows *exception-*$\mathcal{I}$ $\mathcal{I}$, *exception-*$\mathcal{I}$ $\mathcal{I}''$ $\vdash_C$ *obsf-converter* (*comp-converter conv1 conv2*) ∼ *comp-converter* (*obsf-converter conv1*) (*obsf-converter conv2*)
using *WT pfinite1* **supply** *eq-*$\mathcal{I}$*-gpv-map-gpv*[*simp del*]

proof(*coinduction arbitrary*: *conv1 conv2*)
  case *eq-$\mathcal{I}$-converter*
  show *?case* (**is** *eq-$\mathcal{I}$-gpv ?X - ?lhs ?rhs*)
  proof −
    have *eq-$\mathcal{I}$-gpv* (=) (*exception-$\mathcal{I}$ $\mathcal{I}''$*) *?rhs* (*TRY map-gpv* (λ((*b*, *conv1'*), *conv2'*).
(*b*, *conv1'* $\odot$ *conv2'*)) *id*
      (*inline run-converter*
        (*map-gpv'*
          (*case-option* (*Fault*, *const-converter Fault*)
            (λ(*x*, *conv'*). (*OK x*, *obsf-converter conv'*)))
          id option-of-exception (*gpv-stop* (*run-converter conv1 q*)))
        (*obsf-converter conv2*)) *ELSE Done* (*Fault*, *const-converter Fault*))
      (**is** *eq-$\mathcal{I}$-gpv - - - ?rhs2* **is** *eq-$\mathcal{I}$-gpv - - -* (*try-gpv* (*map-gpv ?f - ?inline*) *?else*))
      using *eq-$\mathcal{I}$-converter*
      apply *simp*
      apply(*rule run-colossless-converter.inline-try-gpv*[**where** *$\mathcal{I}$=exception-$\mathcal{I}$ $\mathcal{I}'$*])
      apply(*auto intro*!: *WT-intro pfinite-gpv-stop*[*THEN iffD2*] *dest*: *WT-converterD*
pfinite-converterD colossless-converterD)
      done
    term *bind-gpv* (*inline run-converter* (*map-gpv' id id option-of-exception* (*gpv-stop*
(*run-converter conv1 q*))) (*obsf-converter conv2*))
      (λ(*x*, *conv'*). *case x of None* $\Rightarrow$ *Fail* | *Some x'* $\Rightarrow$ *Done* (*x*, *conv'*))

    also have *?rhs2* = *try-gpv* (*map-gpv ?f id*
      (*map-gpv* (λ(*xo*, *conv'*). *case xo of None* $\Rightarrow$ ((*Fault*, *const-converter Fault*),
conv') | *Some* (*x*, *conv*) $\Rightarrow$ ((*OK x*, *obsf-converter conv*), *conv'*)) *id*
      (*bind-gpv* (*inline run-converter* (*map-gpv' id id option-of-exception* (*gpv-stop*
(*run-converter conv1 q*))) (*obsf-converter conv2*))
        (λ(*x*, *conv'*). *case x of None* $\Rightarrow$ *Fail* | *Some x'* $\Rightarrow$ *Done* (*x*, *conv'*)))))
      ?else
      apply(*simp add*: *map-gpv-bind-gpv gpv.map-id*)
      apply(*subst try-gpv-bind-gpv*)
      apply(*simp add*: *split-def option.case-distrib*[**where** *h=map-gpv - -*] *option.case-distrib*[**where** *h=fst*] *option.case-distrib*[**where** *h=λx. try-gpv x -*] *cong*
del: *option.case-cong*)
      apply(*subst option.case-distrib*[**where** *h=Done*, *symmetric*, *abs-def*])+
      apply(*fold map-gpv-conv-bind*)
      apply(*simp add*: *map-try-gpv gpv.map-comp o-def*)
      apply(*rule try-gpv-cong*)
      apply(*subst map-gpv'-id12*)
      apply(*subst map-gpv'-map-gpv-swap*)
      apply(*simp add*: *inline-map-gpv gpv.map-comp id-def*[*symmetric*])
      apply(*rule gpv.map-cong*[*OF refl*])
      apply(*auto split*: *option.split*)
      done
    also have ... = *try-gpv* (*map-gpv ?f id*
      (*map-gpv* (λ(*xo*, *conv'*). *case xo of None* $\Rightarrow$ ((*Fault*, *const-converter Fault*),
conv') | *Some* (*x*, *conv*) $\Rightarrow$ ((*OK x*, *obsf-converter conv*), *conv'*)) *id*
(*bind-gpv*

(*map-gpv′ id id option-of-exception*
(*gpv-stop* (*inline run-converter* (*run-converter conv1 q*) *conv2*)))
(*case-option Fail*
(λ(*x′*, *conv*).
Done
(*Some x′*,
obsf-converter
conv)))))) *?else*
by(*simp only*: *inline-gpv-stop-obsf-converter*)
also have *eq-*$\mathcal{I}$*-gpv ?X* (*exception-*$\mathcal{I}$ $\mathcal{I}''$) *?lhs* ... **using** *eq-*$\mathcal{I}$*-converter*
apply *simp*
apply(*simp add*: *map-gpv-bind-gpv gpv.map-id*)
apply(*subst try-gpv-bind-gpv*)
apply(*simp add*: *split-def option.case-distrib*[**where** *h*=*map-gpv - -*] *option.case-distrib*[**where** *h*=*fst*] *option.case-distrib*[**where** *h*=λ*x*. *try-gpv x -*] *cong del*: *option.case-cong*)
apply(*subst option.case-distrib*[**where** *h*=*Done*, *symmetric*, *abs-def*])+
apply(*fold map-gpv-conv-bind*)
apply(*simp add*: *map-try-gpv gpv.map-comp o-def*)
apply(*rule eq-*$\mathcal{I}$*-gpv-try-gpv-cong*)
apply(*subst map-gpv′-id12*)
apply(*subst map-gpv′-map-gpv-swap*)
apply(*simp add*: *eq-*$\mathcal{I}$*-gpv-map-gpv id-def*[*symmetric*])
apply(*subst map-gpv-conv-map-gpv′*)
apply(*subst gpv-stop-map′*)
apply(*subst option.map-id0*)
apply(*subst map-gpv-conv-map-gpv′*[*symmetric*])
apply(*subst map-gpv′-map-gpv-swap*)
apply(*simp add*: *eq-*$\mathcal{I}$*-gpv-map-gpv id-def*[*symmetric*])
apply(*rule eq-*$\mathcal{I}$*-gpv-reflI*)
apply(*clarsimp split*!: *option.split simp add*: *eq-onp-def*)
apply(*erule notE*)
apply(*rule eq-*$\mathcal{I}$*-converter-reflI*)
apply *simp*
apply(*drule results-gpv-stop-SomeD*)
apply(*rule conjI*)
apply(*rule imageI*)
apply(*drule run-converter.results-gpv-inline*)
apply(*erule* (*1*) *WT-converterD*)
apply *simp*
apply *clarsimp*
apply(*drule* (*2*) *WT-converterD-results*)
apply *simp*
apply(*rule disjI1*)
apply(*rule exI conjI refl*)+
apply(*drule run-converter.results-gpv-inline*)
apply(*erule* (*1*) *WT-converterD*)
apply *simp*
apply *clarsimp*

apply(*drule* (*2*) *WT-converterD-results*)
apply *simp*
apply(*drule run-converter.results-gpv-inline*)
apply(*erule* (*1*) *WT-converterD*)
apply *simp*
apply *clarsimp*
apply(*drule* (*1*) *pfinite-converterD*)
apply *blast*
apply(*rule WT-intro run-converter.WT-gpv-inline-invar*|*simp*)+
apply(*erule* (*1*) *WT-converterD*)
apply *simp*
apply(*simp add*: *eq-onp-def*)
apply(*rule disjI2*)
apply(*rule eq-$\mathcal{I}$-converter-reflI*)
apply *simp*
done
finally (*eq-$\mathcal{I}$-gpv-eq-OO2*) **show** *?thesis* **.**
qed
qed

lemma *resource-of-obsf-oracle-Fault* [*simp*]:
resource-of-oracle (*obsf-oracle oracle*) *Fault* = *const-resource Fault*
by(*coinduction*)(*auto simp add*: *rel-fun-def*)

lemma *obsf-resource-of-oracle* [*simp*]:
obsf-resource (*resource-of-oracle oracle s*) = *resource-of-oracle* (*obsf-oracle oracle*) (*OK s*)
by(*coinduction arbitrary*: *s rule*: *resource.coinduct-strong*)
(*auto 4 3 simp add*: *rel-fun-def map-try-spmf spmf-rel-map intro*!: *rel-spmf-try-spmf rel-spmf-reflI*)

lemma *trace-callee-eq-obsf-Fault* [*simp*]: $A \vdash_C$ *obsf-oracle callee1*(*Fault*) $\approx$ *obsf-oracle callee2*(*Fault*)
by(*coinduction rule*: *trace-callee-eq-coinduct*) *auto*

lemma *obsf-resource-eq-$\mathcal{I}$-cong*: $A \vdash_R$ *obsf-resource res1* $\sim$ *obsf-resource res2* **if** $A \vdash_R$ *res1* $\sim$ *res2*
using *that* **by**(*coinduction arbitrary*: *res1 res2*)(*fastforce intro*!: *rel-spmf-try-spmf simp add*: *spmf-rel-map elim*!: *rel-spmf-mono dest*: *eq-resource-onD*)

lemma *trace-callee-eq-obsf-oracleI*:
assumes *trace-callee-eq callee1 callee2 A p q*
shows *trace-callee-eq* (*obsf-oracle callee1*) (*obsf-oracle callee2*) *A* (*try-spmf* (*map-spmf OK p*) (*return-spmf Fault*)) (*try-spmf* (*map-spmf OK q*) (*return-spmf Fault*))
using *assms*
proof(*coinduction arbitrary*: *p q rule*: *trace-callee-eq-coinduct-strong*)
case (*step z p q*)
have *?lhs* = *map-pmf* (λx. *case x of None* $\Rightarrow$ *Some Fault* | *Some y* $\Rightarrow$ *Some* (*OK y*)) (*bind-spmf p* ($\lambda s'$. *map-spmf fst* (*callee1 s' z*)))

by(*auto simp add*: *bind-spmf-def try-spmf-def bind-assoc-pmf map-bind-pmf bind-map-pmf bind-return-pmf option.case-distrib*[**where** *h*=*map-pmf* -] *option.case-distrib*[**where** *h*=*return-pmf*, *symmetric*, *abs-def*] *map-pmf-def*[*symmetric*] *pmf.map-comp o-def intro*!: *bind-pmf-cong*[*OF refl*] *pmf.map-cong*[*OF refl*] *split*: *option.split*)
also have *bind-spmf p* (*λs'*. *map-spmf fst* (*callee1 s' z*)) = *bind-spmf q* (*λs'*. *map-spmf fst* (*callee2 s' z*))
using *step*(*1*)[*THEN trace-callee-eqD*[**where** *xs*=[] **and** *x*=*z*]] *step*(*2*) **by** *simp*
also have *map-pmf* (*λx*. *case x of None* ⇒ *Some Fault* | *Some y* ⇒ *Some* (*OK y*)) . . . = *?rhs*
by(*auto simp add*: *bind-spmf-def try-spmf-def bind-assoc-pmf map-bind-pmf bind-map-pmf bind-return-pmf option.case-distrib*[**where** *h*=*map-pmf* -] *option.case-distrib*[**where** *h*=*return-pmf*, *symmetric*, *abs-def*] *map-pmf-def*[*symmetric*] *pmf.map-comp o-def intro*!: *bind-pmf-cong*[*OF refl*] *pmf.map-cong*[*OF refl*] *split*: *option.split*)
finally show *?case* **.**
next
case (*sim x s1 s2 ye s1' s2' p q*)
have *eq1*: *bind-spmf* (*try-spmf* (*map-spmf OK p*) (*return-spmf Fault*)) (*λs*. *obsf-oracle callee1 s x*) =
try-spmf (*bind-spmf p* (*λs*. *map-spmf* (*map-prod OK OK*) (*callee1 s x*))) (*return-spmf* (*Fault*, *Fault*))
by(*auto simp add*: *bind-spmf-def try-spmf-def bind-assoc-pmf bind-map-pmf bind-return-pmf intro*!: *bind-pmf-cong*[*OF refl*] *split*: *option.split*)
have *eq2*: *bind-spmf* (*try-spmf* (*map-spmf OK q*) (*return-spmf Fault*)) (*λs*. *obsf-oracle callee2 s x*) =
try-spmf (*bind-spmf q* (*λs*. *map-spmf* (*map-prod OK OK*) (*callee2 s x*))) (*return-spmf* (*Fault*, *Fault*))
by(*auto simp add*: *bind-spmf-def try-spmf-def bind-assoc-pmf bind-map-pmf bind-return-pmf intro*!: *bind-pmf-cong*[*OF refl*] *split*: *option.split*)

show *?case*
proof(*cases ye*)
case [*simp*]: *Fault*
have *lossless-spmf* (*bind-spmf p* (*λs*. *map-spmf* (*map-prod OK OK*) (*callee1 s x*))) ⟷ *lossless-spmf* (*bind-spmf q* (*λs*. *map-spmf* (*map-prod OK OK*) (*callee2 s x*)))
using *sim*(*1*)[*THEN trace-callee-eqD*[**where** *xs*=[] **and** *x*=*x*], *THEN arg-cong*[**where** *f*=*lossless-spmf*]] *sim*(*2*) **by** *simp*
then have *?eq* **by**(*simp add*: *eq1 eq2*)(*subst* (*1 2*) *cond-spmf-fst-try2*, *auto*)
then show *?thesis* **..**
next
case [*simp*]: (*OK y*)
have *eq3*: *fst* ‘ *set-spmf* (*bind-spmf p* (*λs*. *callee1 s x*)) = *fst* ‘ *set-spmf* (*bind-spmf q* (*λs*. *callee2 s x*))
using *trace-callee-eqD*[*OF sim*(*1*) - *sim*(*2*), **where** *xs*=[], *THEN arg-cong*[**where** *f*=*set-spmf*]]
by(*auto simp add*: *bind-UNION image-UN del*: *equalityI*)
show *?thesis*
proof(*cases y* ∈ *fst* ‘ *set-spmf* (*bind-spmf p* (*λs*. *callee1 s x*)))
case *True*

have *eq4*: *cond-spmf-fst* (*bind-spmf p* (*λs. map-spmf* (*apfst OK*) (*callee1 s x*))) (*OK y*) = *cond-spmf-fst* (*bind-spmf p* (*λs. callee1 s x*)) *y*
cond-spmf-fst (*bind-spmf q* (*λs. map-spmf* (*apfst OK*) (*callee2 s x*))) (*OK y*) = *cond-spmf-fst* (*bind-spmf q* (*λs. callee2 s x*)) *y*
by(*fold map-bind-spmf*[*unfolded o-def*])(*simp-all add*: *cond-spmf-fst-map-inj*)
have *?sim*
unfolding *eq1 eq2*
apply(*subst* (*1 2*) *cond-spmf-fst-try1*)
apply *simp*
apply *simp*
apply(*rule exI*[**where** *x*=*cond-spmf-fst* (*bind-spmf p* (*λs. map-spmf* (*apfst OK*) (*callee1 s x*))) *ye*])
apply(*rule exI*[**where** *x*=*cond-spmf-fst* (*bind-spmf q* (*λs. map-spmf* (*apfst OK*) (*callee2 s x*))) *ye*])
apply(*subst* (*1 2*) *try-spmf-lossless*)
subgoal using *True* **unfolding** *eq3* **by**(*auto simp add*: *bind-UNION image-UN intro*!: *rev-bexI rev-image-eqI*)
subgoal using *True* **by**(*auto simp add*: *bind-UNION image-UN intro*!: *rev-bexI rev-image-eqI*)
apply(*simp add*: *map-cond-spmf-fst map-bind-spmf o-def spmf.map-comp map-prod-def split-def*)
apply(*simp add*: *eq4*)
apply(*rule trace-callee-eqI*)
subgoal for *xs z*
using *sim*(*1*)[*THEN trace-callee-eqD*[**where** *xs*=(*x*, *y*) # *xs* **and** *x*=*z*]] *sim*(*2*)
apply *simp*
done
done
then show *?thesis* **..**
next
case *False*
with *eq3* **have** $y \notin$ *fst* ‘ *set-spmf* (*bind-spmf q* (*λs. callee2 s x*)) **by** *auto*
with *False* **have** *?eq*
apply *simp*
apply(*subst* (*1 2*) *cond-spmf-fst-eq-return-None*[*THEN iffD2*])
apply(*auto simp add*: *bind-UNION split*: *if-split-asm intro*: *rev-image-eqI*)
done
then show *?thesis* **..**
qed
qed
qed

lemma *trace-callee-eq′-obsf-resourceI*:
assumes $A \vdash_C$ *callee1*(*s*) ≈ *callee2*(*s′*)
shows $A \vdash_C$ *obsf-oracle callee1*(*OK s*) ≈ *obsf-oracle callee2*(*OK s′*)
using *assms*[*THEN trace-callee-eq-obsf-oracleI*] **by** *simp*

lemma *trace-eq-obsf-resourceI*:

```
  assumes A ⊢R res1 ≈ res2
  shows A ⊢R obsf-resource res1 ≈ obsf-resource res2
  using assms
  apply(subst (2 4) resource-of-oracle-run-resource[symmetric])
  apply(subst (asm) (1 2) resource-of-oracle-run-resource[symmetric])
  apply(drule trace-callee-eq'-obsf-resourceI)
  apply simp
  apply(simp add: resource-of-oracle-run-resource)
  done

lemma spmf-run-obsf-oracle-obsf-distinguisher [rule-format]:
  defines eg1 ≡ exec-gpv and eg2 ≡ exec-gpv shows
   spmf (map-spmf fst (eg1 (obsf-oracle oracle) (obsf-distinguisher gpv) (OK s)))
True =
    spmf (map-spmf fst (eg2 oracle gpv s)) True
  (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs unfolding eg1-def
  proof(induction arbitrary: gpv s rule: exec-gpv-fixp-induct-strong)
    case adm show ?case by simp
    case bottom show ?case by simp
    case (step exec-gpv')
    show ?case unfolding eg2-def
      apply(subst exec-gpv.simps)
     apply(clarsimp simp add: obsf-distinguisher-def bind-map-spmf map-bind-spmf
o-def)
      apply(subst (1 2) spmf-bind)
      apply(rule Bochner-Integration.integral-mono)
        apply(rule measure-spmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
       apply(rule measure-spmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
      apply(clarsimp split: generat.split simp add: map-bind-spmf o-def)
      apply(simp add: try-spmf-def bind-spmf-pmf-assoc bind-map-pmf)
      apply(simp add: bind-spmf-def)
      apply(subst (1 2) pmf-bind)
      apply(rule Bochner-Integration.integral-mono)
        apply(rule measure-pmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
       apply(rule measure-pmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
      apply(clarsimp split!: option.split simp add: bind-return-pmf)
       apply(rule antisym)
        apply(rule order-trans)
         apply(rule step.hyps[THEN ord-spmf-map-spmfI, THEN ord-spmf-eq-leD])
        apply simp
       apply(simp)
      apply(rule step.IH[unfolded eg2-def obsf-distinguisher-def])
      done
```

```
  qed

  show ?rhs ≤ ?lhs unfolding eg2-def
  proof(induction arbitrary: gpv s rule: exec-gpv-fixp-induct-strong)
    case adm show ?case by simp
    case bottom show ?case by simp
    case (step exec-gpv')
    show ?case unfolding eg1-def
      apply(subst exec-gpv.simps)
     apply(clarsimp simp add: obsf-distinguisher-def bind-map-spmf map-bind-spmf
o-def)
      apply(subst (1 2) spmf-bind)
      apply(rule Bochner-Integration.integral-mono)
         apply(rule measure-spmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
        apply(rule measure-spmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
      apply(clarsimp split: generat.split simp add: map-bind-spmf o-def)
      apply(simp add: try-spmf-def bind-spmf-pmf-assoc bind-map-pmf)
      apply(simp add: bind-spmf-def)
      apply(subst (1 2) pmf-bind)
      apply(rule Bochner-Integration.integral-mono)
         apply(rule measure-pmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
        apply(rule measure-pmf.integrable-const-bound[where B=1]; simp add:
pmf-le-1)
      apply(clarsimp split!: option.split simp add: bind-return-pmf)
      apply(rule step.IH[unfolded eg1-def obsf-distinguisher-def])
      done
  qed
qed

lemma spmf-obsf-distinguisher-obsf-resource-True:
   spmf (connect-obsf (obsf-distinguisher 𝒜) (obsf-resource res)) True = spmf
(connect 𝒜 res) True
  unfolding connect-def
  apply(subst (2) resource-of-oracle-run-resource[symmetric])
 apply(simp add: exec-gpv-resource-of-oracle spmf.map-comp spmf-run-obsf-oracle-obsf-distinguisher)
  done

lemma advantage-obsf-distinguisher:
 advantage (obsf-distinguisher 𝒜) (obsf-resource ideal-resource) (obsf-resource real-resource)
=
   advantage 𝒜 ideal-resource real-resource
 unfolding advantage-def by(simp add: spmf-obsf-distinguisher-obsf-resource-True)

end
theory Fold-Spmf
  imports
```

More-CC
begin

primrec (*transfer*)
foldl-spmf :: (′*b* ⇒ ′*a* ⇒ ′*b spmf*) ⇒ ′*b spmf* ⇒ ′*a list* ⇒ ′*b spmf*
where
foldl-spmf-Nil: *foldl-spmf f p* [] = *p*
| *foldl-spmf-Cons*: *foldl-spmf f p* (*x* # *xs*) = *foldl-spmf f* (*bind-spmf p* (λ*a*. *f a x*)) *xs*

lemma *foldl-spmf-return-pmf-None* [*simp*]:
foldl-spmf f (*return-pmf None*) *xs* = *return-pmf None*
by(*induction xs*) *simp-all*

lemma *foldl-spmf-bind-spmf*: *foldl-spmf f* (*bind-spmf p g*) *xs* = *bind-spmf p* (λ*a*. *foldl-spmf f* (*g a*) *xs*)
by(*induction xs arbitrary*: *g*) *simp-all*

lemma *bind-foldl-spmf-return*:
bind-spmf p (λ*x*. *foldl-spmf f* (*return-spmf x*) *xs*) = *foldl-spmf f p xs*
by(*simp add*: *foldl-spmf-bind-spmf*[*symmetric*])

lemma *foldl-spmf-map* [*simp*]: *foldl-spmf f p* (*map g xs*) = *foldl-spmf* (*map-fun id* (*map-fun g id*) *f*) *p xs*
by(*induction xs arbitrary*: *p*) *simp-all*

lemma *foldl-spmf-identity* [*simp*]: *foldl-spmf* (λ*s x*. *return-spmf s*) *p xs* = *p*
by(*induction xs arbitrary*: *p*) *simp-all*

lemma *foldl-spmf-conv-foldl*:
foldl-spmf (λ*s x*. *return-spmf* (*f s x*)) *p xs* = *map-spmf* (λ*s*. *foldl f s xs*) *p*
by(*induction xs arbitrary*: *p*)(*simp-all add*: *map-spmf-conv-bind-spmf*[*symmetric*] *spmf.map-comp o-def*)

lemma *foldl-spmf-Cons*′:
foldl-spmf f (*return-spmf a*) (*x* # *xs*) = *bind-spmf* (*f a x*) (λ*a*′. *foldl-spmf f* (*return-spmf a*′) *xs*)
by(*simp add*: *bind-foldl-spmf-return*)

lemma *foldl-spmf-append*: *foldl-spmf f p* (*xs* @ *ys*) = *foldl-spmf f* (*foldl-spmf f p xs*) *ys*
by(*induction xs arbitrary*: *p*) *simp-all*

lemma
foldl-spmf-helper:
assumes ⋀*x*. *h* (*f x*) = *x*
assumes ⋀*x*. *f* (*h x*) = *x*
shows *foldl-spmf* (λ*a e*. *map-spmf h* (*g* (*f a*) *e*)) *acc es* =

map-spmf h (*foldl-spmf g* (*map-spmf f acc*) *es*)
using *assms* **proof** (*induction es arbitrary*: *acc*)
case (*Cons a es*)
then show *?case*
by (*simp add*: *spmf.map-comp map-bind-spmf bind-map-spmf o-def*)
qed (*simp add*: *map-spmf-conv-bind-spmf*)

lemma
foldl-spmf-helper2:
assumes $\bigwedge x\ y.\ p\ (f\ x\ y) = x$
assumes $\bigwedge x\ y.\ q\ (f\ x\ y) = y$
assumes $\bigwedge x.\ f\ (p\ x)\ (q\ x) = x$
shows *foldl-spmf* ($\lambda a\ e.$ *map-spmf* (f (p a)) (g (q a) e)) *acc es* =
bind-spmf acc ($\lambda acc'.$ *map-spmf* (f (p acc')) (*foldl-spmf g* (*return-spmf* (q acc'))
es))
proof (*induction es arbitrary*: *acc*)
note [*simp*] = *spmf.map-comp map-bind-spmf bind-map-spmf o-def*
case (*Cons e es*)
then show *?case*
apply (*simp add*: *map-spmf-conv-bind-spmf assms*)
apply (*subst bind-spmf-assoc*[*symmetric*])
by (*simp add*: *bind-foldl-spmf-return*)
qed (*simp add*: *assms*(*3*))

lemma *foldl-pair-constl*: *foldl* ($\lambda s\ e.$ *map-prod* (λ-. c) ($\lambda r.\ f\ r\ e$) s) (c, sr) l =
Pair c (*foldl* ($\lambda s\ e.\ f\ s\ e$) *sr l*)
by (*induction l arbitrary*: *sr*) (*auto simp add*: *map-prod-def split-def*)

lemma *foldl-spmf-pair-left*:
foldl-spmf (λ(l, r) e. *map-spmf* ($\lambda l'.$ (l', r)) (f l e)) (*return-spmf* (l, r)) *es* =
map-spmf ($\lambda l'.$ (l', r)) (*foldl-spmf f* (*return-spmf l*) *es*)
apply (*induction es arbitrary*: *l*)
apply *simp-all*
apply (*subst* (*2*) *map-spmf-conv-bind-spmf*)
apply (*subst foldl-spmf-bind-spmf*)
apply (*subst* (*2*) *bind-foldl-spmf-return*[*symmetric*])
by (*simp add*: *map-spmf-conv-bind-spmf*)

lemma *foldl-spmf-pair-left2*:
foldl-spmf (λ(l, -) e. *map-spmf* ($\lambda l'.$ (l', c')) (f l e)) (*return-spmf* (l, c)) *es* =
map-spmf ($\lambda l'.$ (l', *if es* = [] *then c else* c')) (*foldl-spmf f* (*return-spmf l*) *es*)
apply (*induction es arbitrary*: l c c')
apply *simp-all*
apply (*subst* (*2*) *map-spmf-conv-bind-spmf*)
apply (*subst foldl-spmf-bind-spmf*)
apply (*subst* (*2*) *bind-foldl-spmf-return*[*symmetric*])
by (*simp add*: *map-spmf-conv-bind-spmf*)

lemma *foldl-pair-constr*: *foldl* ($\lambda s\ e.$ *map-prod* ($\lambda l.\ f\ l\ e$) (λ-. c) s) (sl, c) l =

Pair (*foldl* (λ*s e*. *f s e*) *sl l*) *c*
by (*induction l arbitrary*: *sl*) (*auto simp add*: *map-prod-def split-def*)

lemma *foldl-spmf-pair-right*:
foldl-spmf (λ(*l*, *r*) *e*. *map-spmf* (λ*r*′. (*l*, *r*′)) (*f r e*)) (*return-spmf* (*l*, *r*)) *es* =
map-spmf (λ*r*′. (*l*, *r*′)) (*foldl-spmf f* (*return-spmf r*) *es*)
apply (*induction es arbitrary*: *r*)
apply *simp-all*
apply (*subst* (*2*) *map-spmf-conv-bind-spmf*)
apply (*subst foldl-spmf-bind-spmf*)
apply (*subst* (*2*) *bind-foldl-spmf-return*[*symmetric*])
by (*simp add*: *map-spmf-conv-bind-spmf*)

lemma *foldl-spmf-pair-right2*:
foldl-spmf (λ(-, *r*) *e*. *map-spmf* (λ*r*′. (*c*′, *r*′)) (*f r e*)) (*return-spmf* (*c*, *r*)) *es* =
map-spmf (λ*r*′. (*if es* = [] *then c else c*′, *r*′)) (*foldl-spmf f* (*return-spmf r*) *es*)
apply (*induction es arbitrary*: *r c c*′)
apply *simp-all*
apply (*subst* (*2*) *map-spmf-conv-bind-spmf*)
apply (*subst foldl-spmf-bind-spmf*)
apply (*subst* (*2*) *bind-foldl-spmf-return*[*symmetric*])
by (*auto simp add*: *map-spmf-conv-bind-spmf split-def*)

lemma *foldl-spmf-pair-right3*:
foldl-spmf (λ(*l*, *r*) *e*. *map-spmf* (*Pair* (*g e*)) (*f r e*)) (*return-spmf* (*l*, *r*)) *es* =
map-spmf (*Pair* (*if es* = [] *then l else g* (*last es*))) (*foldl-spmf f* (*return-spmf r*) *es*)
apply (*induction es arbitrary*: *r l*)
apply *simp-all*
apply (*subst* (*2*) *map-spmf-conv-bind-spmf*)
apply (*subst foldl-spmf-bind-spmf*)
apply (*subst* (*2*) *bind-foldl-spmf-return*[*symmetric*])
by (*clarsimp simp add*: *split-def map-bind-spmf o-def*)

lemma *foldl-pullout*: *bind-spmf f* (λ*x*. *bind-spmf* (*foldl-spmf g init* (*events x*)) (λ*y*. *h x y*)) =
bind-spmf (*bind-spmf f* (λ*x*. *foldl-spmf* (λ(*l*, *r*) *e*. *map-spmf* (*Pair l*) (*g r e*)) (*map-spmf* (*Pair x*) *init*) (*events x*)))
(λ(*x*, *y*). *h x y*) **for** *f g h init events*
apply (*simp add*: *foldl-spmf-helper2*[**where** *f*=*Pair* **and** *p*=*fst* **and** *q*=*snd*, *simplified*] *split-def*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*)
by (*subst bind-spmf-assoc*[*symmetric*]) (*auto simp add*: *bind-foldl-spmf-return*)

lemma *bind-foldl-spmf-pair-append*:
bind-spmf
(*foldl-spmf* (λ(*x*, *y*) *e*. *map-spmf* (*apfst* ((@) *x*)) (*f y e*)) (*return-spmf* (*a* @ *c*, *b*)) *es*)
(λ(*x*, *y*). *g x y*) =

bind-spmf
  (*foldl-spmf* (λ(*x*, *y*) *e*. *map-spmf* (*apfst* ((@) *x*)) (*f y e*)) (*return-spmf* (*c*, *b*)) *es*)
  (λ(*x*, *y*). *g* (*a* @ *x*) *y*)
apply (*induction es arbitrary*: *c b*)
apply (*simp-all add*: *split-def map-spmf-conv-bind-spmf apfst-def map-prod-def*)
apply (*subst* (*1 2*) *foldl-spmf-bind-spmf*)
by *simp*

lemma *foldl-spmf-chain*:
(*foldl-spmf* (λ(*oevents*, *s-event*) *event*. *map-spmf* (*map-prod* ((@) *oevents*) *id*) (*fff s-event event*)) (*return-spmf* ([], *s-event*)) *ievents*)
≫= (λ(*oevents*, *s-event′*). *foldl-spmf ggg* (*return-spmf s-core*) *oevents*
≫= (λ*s-core′*. *return-spmf* (*f s-core′ s-event′*))) =
foldl-spmf (λ(*s-event*, *s-core*) *event*. *fff s-event event* ≫= (λ(*oevents*, *s-event′*).
map-spmf (*Pair s-event′*) (*foldl-spmf ggg* (*return-spmf s-core*) *oevents*)))
(*return-spmf* (*s-event*, *s-core*)) *ievents*
≫= (λ(*s-event′*, *s-core′*). *return-spmf* (*f s-core′ s-event′*))
proof (*induction ievents arbitrary*: *s-event s-core*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons e es*)

show *?case*
apply (*subst* (*1 2*) *foldl-spmf-Cons′*)
apply (*simp add*: *split-def*)
apply (*subst map-spmf-conv-bind-spmf*)
apply *simp*
apply (*rule bind-spmf-cong*[*OF refl*])
apply (*subst* (*2*) *map-spmf-conv-bind-spmf*)
apply *simp*
apply (*subst Cons.IH*[*symmetric*, *simplified split-def*])
apply (*subst bind-commute-spmf*)
apply (*subst* (*2*) *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst map-bind-spmf*[*symmetric*, *simplified o-def*])
apply (*subst* (*1*) *foldl-spmf-bind-spmf*[*symmetric*])
apply (*subst* (*3*) *map-spmf-conv-bind-spmf*)
apply (*simp add*: *foldl-spmf-append*[*symmetric*] *map-prod-def split-def*)
subgoal for *x*
apply (*cases x*)
subgoal for *a b*
apply (*simp add*: *split-def*)
apply (*subst bind-foldl-spmf-pair-append*[**where** *c*=[] **and** *a*=*a* **and** *b*=*b* **and** *es*=*es*, *simplified apfst-def map-prod-def append-Nil2 split-def id-def*])
by *simp*
done
done
qed

— pauses
primrec *pauses* :: *'a list* ⇒ (*unit*, *'a*, *'b*) *gpv* **where**
pauses [] = *Done* ()
| *pauses* (*x* # *xs*) = *Pause x* (λ-. *pauses xs*)

lemma *WT-gpv-pauses* [*WT-intro*]:
$\mathcal{I}$ ⊢g *pauses xs* √ **if** *set xs* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$
using *that* **by**(*induction xs*) *auto*

lemma *exec-gpv-pauses*:
exec-gpv callee (*pauses xs*) *s* =
map-spmf (*Pair* ()) (*foldl-spmf* (*map-fun id* (*map-fun id* (*map-spmf snd*)) *callee*) (*return-spmf s*) *xs*)
by(*induction xs arbitrary*: *s*)(*simp-all add*: *split-def foldl-spmf-Cons' map-bind-spmf bind-map-spmf o-def del*: *foldl-spmf-Cons*)

end
theory *Fused-Resource* **imports**
Fold-Spmf
begin

context includes $\mathcal{I}$*.lifting* **begin**
lift-definition *e*$\mathcal{I}$ :: (*'a*, *'b*) $\mathcal{I}$ ⇒ (*'a*, *'b* × *'c*) $\mathcal{I}$ **is** λ$\mathcal{I}$ *x*. $\mathcal{I}$ *x* × *UNIV* **.**

lemma *outs-*$\mathcal{I}$*-e*$\mathcal{I}$[*simp*]: *outs-*$\mathcal{I}$ (*e*$\mathcal{I}$ $\mathcal{I}$) = *outs-*$\mathcal{I}$ $\mathcal{I}$
by *transfer simp*

lemma *responses-*$\mathcal{I}$*-e*$\mathcal{I}$ [*simp*]: *responses-*$\mathcal{I}$ (*e*$\mathcal{I}$ $\mathcal{I}$) *x* = *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* × *UNIV*
by *transfer simp*

lemma *e*$\mathcal{I}$*-map-*$\mathcal{I}$: *e*$\mathcal{I}$ (*map-*$\mathcal{I}$ *f g* $\mathcal{I}$) = *map-*$\mathcal{I}$ *f* (*apfst g*) (*e*$\mathcal{I}$ $\mathcal{I}$)
by *transfer*(*auto simp add*: *fun-eq-iff intro*: *rev-image-eqI*)

lemma *e*$\mathcal{I}$*-inverse* [*simp*]: *map-*$\mathcal{I}$ *id fst* (*e*$\mathcal{I}$ $\mathcal{I}$) = $\mathcal{I}$
by *transfer auto*
end
lifting-update $\mathcal{I}$*.lifting*
lifting-forget $\mathcal{I}$*.lifting*

4 Fused Resource

4.1 Event Oracles – they generate events

type-synonym
(*'state*, *'event*, *'input*, *'output*) *eoracle* = (*'state*, *'input*, *'output* × *'event list*)

oracle′

definition
parallel-eoracle ::
('*s1*, '*e1*, '*i1*, '*o1*) *eoracle* ⇒ ('*s2*, '*e2*, '*i2*, '*o2*) *eoracle* ⇒
('*s1* × '*s2*, '*e1* + '*e2*, '*i1* + '*i2*, '*o1* + '*o2*) *eoracle*
where
parallel-eoracle eoracle1 eoracle2 state ≡
comp
(*map-spmf*
(*map-prod*
(*case-sum*
(*map-prod Inl* (*map Inl*))
(*map-prod Inr* (*map Inr*)))
id))
(*parallel-oracle eoracle1 eoracle2 state*)

definition
plus-eoracle ::
('*s*, '*e1*, '*i1*, '*o1*) *eoracle* ⇒ ('*s*, '*e2*, '*i2*, '*o2*) *eoracle* ⇒
('*s*, '*e1* + '*e2*, '*i1* + '*i2*, '*o1* + '*o2*) *eoracle*
where
plus-eoracle eoracle1 eoracle2 state ≡
comp
(*map-spmf*
(*map-prod*
(*case-sum*
(*map-prod Inl* (*map Inl*))
(*map-prod Inr* (*map Inr*)))
id))
(*plus-oracle eoracle1 eoracle2 state*)

definition
translate-eoracle ::
('*s-event*, '*e1*, '*e2 list*) *oracle′* ⇒ ('*s-event* × '*s*, '*e1*, '*i*, '*o*) *eoracle* ⇒
('*s-event* × '*s*, '*e2*, '*i*, '*o*) *eoracle*
where
translate-eoracle translator eoracle state inp ≡
bind-spmf
(*eoracle state inp*)
(λ((*out*, *e-in*), *s*).
let conc = (λ(*es*, *st*) *e*. *map-spmf* (*map-prod* ((@) *es*) *id*) (*translator st e*))
in do {
(*e-out*, *s-event*) ← *foldl-spmf conc* (*return-spmf* ([], *fst s*)) *e-in*;
return-spmf ((*out*, *e-out*), *s-event*, *snd s*)
})

4.2 Event Handlers – they absorb (and silently handle) events

type-synonym
('*state*, '*event*) *handler* = '*state* ⇒ '*event* ⇒ '*state spmf*

fun
parallel-handler :: ('*s1*, '*e1*) *handler* ⇒ ('*s2*, '*e2*) *handler* ⇒ ('*s1* × '*s2*, '*e1* + '*e2*) *handler*
where
parallel-handler left - s (*Inl e1*) = *map-spmf* (λ*s1*'. (*s1*', *snd s*)) (*left* (*fst s*) *e1*)
| *parallel-handler - right s* (*Inr e2*) = *map-spmf* (λ*s2*'. (*fst s*, *s2*')) (*right* (*snd s*) *e2*)

definition
plus-handler :: ('*s*, '*e1*) *handler* ⇒ ('*s*, '*e2*) *handler* ⇒ ('*s*, '*e1* + '*e2*) *handler*
where
plus-handler left right s ≡ *case-sum* (*left s*) (*right s*)

lemma *parallel-handler-left*:
map-fun id (*map-fun Inl id*) (*parallel-handler left right*) =
(λ(*s-l*, *s-r*) *q*. *map-spmf* (λ*s-l*'. (*s-l*', *s-r*)) (*left s-l q*))
by *force*

lemma *parallel-handler-right*:
map-fun id (*map-fun Inr id*) (*parallel-handler left right*) =
(λ(*s-l*, *s-r*) *q*. *map-spmf* (λ*s-r*'. (*s-l*, *s-r*')) (*right s-r q*))
by *force*

lemma *in-set-spmf-parallel-handler*:
s' ∈ *set-spmf* (*parallel-handler left right s x*) ⟷
(*case x of Inl e* ⇒ *fst s*' ∈ *set-spmf* (*left* (*fst s*) *e*) ∧ *snd s*' = *snd s*
| *Inr e* ⇒ *snd s*' ∈ *set-spmf* (*right* (*snd s*) *e*) ∧ *fst s*' = *fst s*)
by(*cases s*; *cases s*'; *auto split*: *sum.split*)

4.3 Fused Resource Construction

codatatype
('*s-core*, '*event*, '*iadv-core*, '*iusr-core*, '*oadv-core*, '*ousr-core*) *core* =
Core
(*cpoke*: ('*s-core*, '*event*) *handler*)
(*cfunc-adv*: ('*s-core*, '*iadv-core*, '*oadv-core*) *oracle*')
(*cfunc-usr*: ('*s-core*, '*iusr-core*, '*ousr-core*) *oracle*')

declare *core.sel-transfer*[*transfer-rule del*]
declare *core.ctr-transfer*[*transfer-rule del*]
declare *core.case-transfer*[*transfer-rule del*]

context
includes *lifting-syntax*
begin

inductive
rel-core'::
('*s-core* ⇒ '*s-core'* ⇒ *bool*) ⇒
('*event* ⇒ '*event'* ⇒ *bool*) ⇒
('*iadv-core* ⇒ '*iadv-core'* ⇒ *bool*) ⇒
('*iusr-core* ⇒ '*iusr-core'* ⇒ *bool*) ⇒
('*oadv-core* ⇒ '*oadv-core'* ⇒ *bool*) ⇒
('*ousr-core* ⇒ '*ousr-core'* ⇒ *bool*) ⇒
('*s-core*, '*event*, '*iadv-core*, '*iusr-core*, '*oadv-core*, '*ousr-core*) *core* ⇒
('*s-core'*, '*event'*, '*iadv-core'*, '*iusr-core'*, '*oadv-core'*, '*ousr-core'*) *core* ⇒ *bool*
for *S E IA IU OA OU*
where *rel-core' S E IA IU OA OU* (*Core cpoke cfunc-adv cfunc-usr*) (*Core cpoke' cfunc-adv' cfunc-usr'*)
if
(*S* ===> *E* ===> *rel-spmf S*) *cpoke cpoke'* **and**
(*S* ===> *IA* ===> *rel-spmf* (*rel-prod OA S*)) *cfunc-adv cfunc-adv'* **and**
(*S* ===> *IU* ===> *rel-spmf* (*rel-prod OU S*)) *cfunc-usr cfunc-usr'*
for *cpoke cfunc-adv cfunc-usr*

inductive-simps
rel-core'-simps [*simp*]:
rel-core' S E IA IU OA OU (*Core cpoke' cfunc-adv' cfunc-usr'*) (*Core cpoke'' cfunc-adv'' cfunc-usr''*)

lemma
rel-core'-eq [*relator-eq*]:
rel-core' (=) (=) (=) (=) (=) (=) = (=)
apply(*intro ext*)
subgoal for *x y* **by**(*cases x*; *cases y*)(*auto simp add*: *fun-eq-iff rel-fun-def relator-eq*)
done

lemma
rel-core'-mono [*relator-mono*]:
rel-core' S E IA IU OA OU ≤ *rel-core' S E' IA' IU' OA' OU'*
if *E'* ≤ *E IA'* ≤ *IA IU'* ≤ *IU OA* ≤ *OA' OU* ≤ *OU'*
apply(*rule predicate2I*)
subgoal for *x y*
apply(*cases x*; *cases y*; *clarsimp*; *intro conjI*)
apply(*erule rel-fun-mono rel-spmf-mono prod.rel-mono*[*THEN predicate2D*, *rotated* −*1*] |
rule impI that order-refl | *erule that*[*THEN predicate2D*] | *erule rel-spmf-mono* | *assumption*)+
done
done

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

(*rel-core′ S E IA IU OA OU* ===> *S* ===> *E* ===> *rel-spmf S*) *cpoke cpoke*
by(*rule rel-funI*; *erule rel-core′.cases*; *simp*)

lemma
cfunc-adv-parametric [*transfer-rule*]:
(*rel-core′ S E IA IU OA OU* ===> *S* ===> *IA* ===> *rel-spmf* (*rel-prod OA S*)) *cfunc-adv cfunc-adv*
by(*rule rel-funI*; *erule rel-core′.cases*; *simp*)

lemma
cfunc-usr-parametric [*transfer-rule*]:
(*rel-core′ S E IA IU OA OU* ===> *S* ===> *IU* ===> *rel-spmf* (*rel-prod OU S*)) *cfunc-usr cfunc-usr*
by(*rule rel-funI*; *erule rel-core′.cases*; *simp*)

lemma
Core-parametric [*transfer-rule*]:
((*S* ===> *E* ===> *rel-spmf S*) ===> (*S* ===> *IA* ===> *rel-spmf* (*rel-prod OA S*)) ===> (*S* ===> *IU* ===> *rel-spmf* (*rel-prod OU S*))
===> *rel-core′ S E IA IU OA OU*) *Core Core*
by(*rule rel-funI*)+ *simp*

lemma
case-core-parametric [*transfer-rule*]:
(((*S* ===> *E* ===> *rel-spmf S*) ===>
(*S* ===> *IA* ===> *rel-spmf* (*rel-prod OA S*)) ===>
(*S* ===> *IU* ===> *rel-spmf* (*rel-prod OU S*)) ===> *X*) ===>
rel-core′ S E IA IU OA OU ===> *X*) *case-core case-core*
by(*rule rel-funI*)+(*auto 4 4 split*: *core.split dest*: *rel-funD*)

lemma
corec-core-parametric [*transfer-rule*]:
((*X* ===> *S* ===> *E* ===> *rel-spmf S*) ===>
(*X* ===> *S* ===> *IA* ===> *rel-spmf* (*rel-prod OA S*)) ===>
(*X* ===> *S* ===> *IU* ===> *rel-spmf* (*rel-prod OU S*)) ===>
X ===> *rel-core′ S E IA IU OA OU*) *corec-core corec-core*
by(*rule rel-funI*)+(*auto simp add*: *core.corec dest*: *rel-funD*)

primcorec *map-core′* ::
(*′event′* ⇒ *′event*) ⇒
(*′iadv-core′* ⇒ *′iadv-core*) ⇒
(*′iusr-core′* ⇒ *′iusr-core*) ⇒
(*′oadv-core* ⇒ *′oadv-core′*) ⇒
(*′ousr-core* ⇒ *′ousr-core′*) ⇒
(*′s-core*, *′event*, *′iadv-core*, *′iusr-core*, *′oadv-core*, *′ousr-core*) *core* ⇒
(*′s-core*, *′event′*, *′iadv-core′*, *′iusr-core′*, *′oadv-core′*, *′ousr-core′*) *core*
where
cpoke (*map-core′ e ia iu oa ou core*) = (*id* ---> *e* ---> *id*) (*cpoke core*)

| *cfunc-adv* (*map-core′ e ia iu oa ou core*) = (*id* −−−> *ia* −−−> *map-spmf* (*map-prod oa id*)) (*cfunc-adv core*)
| *cfunc-usr* (*map-core′ e ia iu oa ou core*) = (*id* −−−> *iu* −−−> *map-spmf* (*map-prod ou id*)) (*cfunc-usr core*)

lemmas *map-core′-simps* [*simp*] = *map-core′.ctr*[**where** *core=Core - - -*, *simplified*]

parametric-constant *map-core′-parametric*[*transfer-rule*]: *map-core′-def*

lemma *core′-rel-Grp*:
rel-core′ (=) (*BNF-Def.Grp UNIV e*)$^{-1-1}$ (*BNF-Def.Grp UNIV ia*)$^{-1-1}$ (*BNF-Def.Grp UNIV iu*)$^{-1-1}$ (*BNF-Def.Grp UNIV oa*) (*BNF-Def.Grp UNIV ou*)
= *BNF-Def.Grp UNIV* (*map-core′ e ia iu oa ou*)
apply(*intro ext*)
subgoal for *x y*
apply(*cases x*; *cases y*; *clarsimp*)
apply(*subst* (*2 4 6*) *eq-alt-conversep*)
apply(*subst* (*2 3 4*) *eq-alt*)
apply(*simp add*: *pmf.rel-Grp option.rel-Grp prod.rel-Grp rel-fun-conversep-grp-grp*)
apply(*auto simp add*: *Grp-def spmf.map-id*[*abs-def*] *id-def*[*symmetric*])
done
done

end

inductive *WT-core* :: (*′iadv*, *′oadv*) $\mathcal{I}$ ⇒ (*′iusr*, *′ousr*) $\mathcal{I}$ ⇒ (*′s* ⇒ *bool*) ⇒ (*′s*, *′event*, *′iadv*, *′iusr*, *′oadv*, *′ousr*) *core* ⇒ *bool*
for $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I core* **where**
WT-core $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I core* **if**
⋀*s e s′*. ⟦ *s′* ∈ *set-spmf* (*cpoke core s e*); *I s* ⟧ ⟹ *I s′*
⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*cfunc-adv core s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-adv*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-adv x* ∧ *I s′*
⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*cfunc-usr core s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-usr*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-usr x* ∧ *I s′*

lemma *WT-coreD*:
assumes *WT-core* $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I core*
shows *WT-coreD-cpoke*: ⋀*s e s′*. ⟦ *s′* ∈ *set-spmf* (*cpoke core s e*); *I s* ⟧ ⟹ *I s′*
and *WT-coreD-cfunc-adv*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*cfunc-adv core s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-adv*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-adv x* ∧ *I s′*
and *WT-coreD-cfund-usr*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*cfunc-usr core s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-usr*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-usr x* ∧ *I s′*
using *assms* **by**(*auto elim*!: *WT-core.cases*)

lemma *WT-coreD-foldl-spmf-cpoke*:
assumes *WT-core* $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I core*
and *s′* ∈ *set-spmf* (*foldl-spmf* (*cpoke core*) *p es*)
and ∀ *s* ∈ *set-spmf p*. *I s*
shows *I s′*

using *assms*(*2*, *3*)
by(*induction es arbitrary*: *p*)(*fastforce dest*: *WT-coreD-cpoke*[*OF assms*(*1*)] *simp add*: *bind-UNION*)+

lemma *WT-core-trivial*:
assumes *adv*: ⋀*s*. *ℐ-adv* ⊢*c cfunc-adv core s* √
and *usr*: ⋀*s*. *ℐ-usr* ⊢*c cfunc-usr core s* √
shows *WT-core ℐ-adv ℐ-usr* (λ-. *True*) *core*
by(*rule WT-core.intros*)(*auto dest*: *assms*[*THEN WT-calleeD*])

codatatype
('*s-rest*, '*event*, '*iadv-rest*, '*iusr-rest*, '*oadv-rest*, '*ousr-rest*, '*more*) *rest-scheme* =
Rest
(*rinit*: '*more*)
(*rfunc-adv*: ('*s-rest*, '*event*, '*iadv-rest*, '*oadv-rest*) *eoracle*)
(*rfunc-usr*: ('*s-rest*, '*event*, '*iusr-rest*, '*ousr-rest*) *eoracle*)

declare *rest-scheme.sel-transfer*[*transfer-rule del*]
declare *rest-scheme.ctr-transfer*[*transfer-rule del*]
declare *rest-scheme.case-transfer*[*transfer-rule del*]

context
includes *lifting-syntax*
begin

inductive
rel-rest'::
('*s-rest* ⇒ '*s-rest*' ⇒ *bool*) ⇒
('*event* ⇒ '*event*' ⇒ *bool*) ⇒
('*iadv-rest* ⇒ '*iadv-rest*' ⇒ *bool*) ⇒
('*iusr-rest* ⇒ '*iusr-rest*' ⇒ *bool*) ⇒
('*oadv-rest* ⇒ '*oadv-rest*' ⇒ *bool*) ⇒
('*ousr-rest* ⇒ '*ousr-rest*' ⇒ *bool*) ⇒
('*more* ⇒ '*more*' ⇒ *bool*) ⇒
('*s-rest*, '*event*, '*iadv-rest*, '*iusr-rest*, '*oadv-rest*, '*ousr-rest*, '*more*) *rest-scheme* ⇒
('*s-rest*', '*event*', '*iadv-rest*', '*iusr-rest*', '*oadv-rest*', '*ousr-rest*', '*more*') *rest-scheme* ⇒ *bool*
for *S E IA IU OA OU M*
where *rel-rest*' *S E IA IU OA OU M* (*Rest rinit rfunc-adv rfunc-usr*) (*Rest rinit*' *rfunc-adv*' *rfunc-usr*')
if
M rinit rinit' **and**
(*S* ===> *IA* ===> *rel-spmf* (*rel-prod* (*rel-prod OA* (*list-all2 E*)) *S*)) *rfunc-adv rfunc-adv*' **and**
(*S* ===> *IU* ===> *rel-spmf* (*rel-prod* (*rel-prod OU* (*list-all2 E*)) *S*)) *rfunc-usr rfunc-usr*'
for *rinit rfunc-adv rfunc-usr*

inductive-simps
rel-rest′-simps [*simp*]:
rel-rest′ S E IA IU OA OU M (*Rest rinit′ rfunc-adv′ rfunc-usr′*) (*Rest rinit″ rfunc-adv″ rfunc-usr″*)

lemma
rel-rest′-eq [*relator-eq*]:
rel-rest′ (=) (=) (=) (=) (=) (=) (=) = (=)
apply(*intro ext*)
subgoal for *x y* **by**(*cases x*; *cases y*)(*auto simp add*: *fun-eq-iff rel-fun-def relator-eq*)
done

lemma
rel-rest′-mono [*relator-mono*]:
rel-rest′ S E IA IU OA OU M ≤ *rel-rest′ S E′ IA′ IU′ OA′ OU′ M′*
if *E* ≤ *E′ IA′* ≤ *IA IU′* ≤ *IU OA* ≤ *OA′ OU* ≤ *OU′ M* ≤ *M′*
apply(*rule predicate2I*)
subgoal for *x y*
apply(*cases x*; *cases y*; *clarsimp*; *intro conjI*)
apply(*erule rel-fun-mono rel-spmf-mono prod.rel-mono*[*THEN predicate2D*, *rotated* −*1*] |
rule impI that order-refl prod.rel-mono list.rel-mono | *erule that*[*THEN predicate2D*] | *erule rel-spmf-mono* | *assumption*)+
done
done

lemma *rel-rest′-sel*: *rel-rest′ S E IA IU OA OU M rest1 rest2*
if *M* (*rinit rest1*) (*rinit rest2*)
and (*S* ===> *IA* ===> *rel-spmf* (*rel-prod* (*rel-prod OA* (*list-all2 E*)) *S*)) (*rfunc-adv rest1*) (*rfunc-adv rest2*)
and (*S* ===> *IU* ===> *rel-spmf* (*rel-prod* (*rel-prod OU* (*list-all2 E*)) *S*)) (*rfunc-usr rest1*) (*rfunc-usr rest2*)
using *that* **by**(*cases rest1*; *cases rest2*) *simp*

lemma *rinit-parametric* [*transfer-rule*]: (*rel-rest′ S E IA IU OA OU M* ===> *M*) *rinit rinit*
by(*rule rel-funI*; *erule rel-rest′.cases*; *simp*)

lemma *rfunc-adv-parametric* [*transfer-rule*]:
(*rel-rest′ S E IA IU OA OU M* ===> *S* ===> *IA* ===> *rel-spmf* (*rel-prod* (*rel-prod OA* (*list-all2 E*)) *S*)) *rfunc-adv rfunc-adv*
by(*rule rel-funI*; *erule rel-rest′.cases*; *simp*)

lemma *rfunc-usr-parametric* [*transfer-rule*]:
(*rel-rest′ S E IA IU OA OU M* ===> *S* ===> *IU* ===> *rel-spmf* (*rel-prod* (*rel-prod OU* (*list-all2 E*)) *S*)) *rfunc-usr rfunc-usr*
by(*rule rel-funI*; *erule rel-rest′.cases*; *simp*)

lemma *Rest-parametric* [*transfer-rule*]:
(*M* ===> (*S* ===> *IA* ===> *rel-spmf* (*rel-prod* (*rel-prod* *OA* (*list-all2* *E*)) *S*))
===> (*S* ===> *IU* ===> *rel-spmf* (*rel-prod* (*rel-prod* *OU* (*list-all2* *E*)) *S*))
===> *rel-rest′* *S* *E* *IA* *IU* *OA* *OU* *M*) *Rest* *Rest*
by(*rule* *rel-funI*)+ *simp*

lemma *case-rest-scheme-parametric* [*transfer-rule*]:
((*M* ===>
(*S* ===> *IA* ===> *rel-spmf* (*rel-prod* (*rel-prod* *OA* (*list-all2* *E*)) *S*)) ===>
(*S* ===> *IU* ===> *rel-spmf* (*rel-prod* (*rel-prod* *OU* (*list-all2* *E*)) *S*)) ===>
X) ===>
rel-rest′ *S* *E* *IA* *IU* *OA* *OU* *M* ===> *X*) *case-rest-scheme* *case-rest-scheme*
by(*rule* *rel-funI*)+(*auto* *4* *4* *split*: *rest-scheme.split* *dest*: *rel-funD*)

lemma *corec-rest-scheme-parametric* [*transfer-rule*]:
((*X* ===> *M*) ===>
(*X* ===> *S* ===> *IA* ===> *rel-spmf* (*rel-prod* (*rel-prod* *OA* (*list-all2* *E*)) *S*)) ===>
(*X* ===> *S* ===> *IU* ===> *rel-spmf* (*rel-prod* (*rel-prod* *OU* (*list-all2* *E*)) *S*)) ===>
X ===> *rel-rest′* *S* *E* *IA* *IU* *OA* *OU* *M*) *corec-rest-scheme* *corec-rest-scheme*
by(*rule* *rel-funI*)+(*auto* *simp* *add*: *rest-scheme.corec* *dest*: *rel-funD*)

primcorec *map-rest′* ::
(*′event* ⇒ *′event′*) ⇒
(*′iadv-rest′* ⇒ *′iadv-rest*) ⇒
(*′iusr-rest′* ⇒ *′iusr-rest*) ⇒
(*′oadv-rest* ⇒ *′oadv-rest′*) ⇒
(*′ousr-rest* ⇒ *′ousr-rest′*) ⇒
(*′more* ⇒ *′more′*) ⇒
(*′s-rest*, *′event*, *′iadv-rest*, *′iusr-rest*, *′oadv-rest*, *′ousr-rest*, *′more*) *rest-scheme* ⇒
(*′s-rest*, *′event′*, *′iadv-rest′*, *′iusr-rest′*, *′oadv-rest′*, *′ousr-rest′*, *′more′*) *rest-scheme*
where
rinit (*map-rest′* *e* *ia* *iu* *oa* *ou* *m* *rest*) = *m* (*rinit* *rest*)
| *rfunc-adv* (*map-rest′* *e* *ia* *iu* *oa* *ou* *m* *rest*) =
(*id* −−−> *ia* −−−> *map-spmf* (*map-prod* (*map-prod* *oa* (*map* *e*)) *id*)) (*rfunc-adv* *rest*)
| *rfunc-usr* (*map-rest′* *e* *ia* *iu* *oa* *ou* *m* *rest*) =
(*id* −−−> *iu* −−−> *map-spmf* (*map-prod* (*map-prod* *ou* (*map* *e*)) *id*)) (*rfunc-usr* *rest*)

lemmas *map-rest′-simps* [*simp*] = *map-rest′.ctr*[**where** *rest*=*Rest* - - -, *simplified*]

parametric-constant *map-rest′-parametric*[*transfer-rule*]: *map-rest′-def*

lemma *rest′-rel-Grp*:
rel-rest′ (=) (*BNF-Def.Grp* *UNIV* *e*) (*BNF-Def.Grp* *UNIV* *ia*)$^{-1-1}$ (*BNF-Def.Grp*

UNIV iu)$^{-1-1}$ (*BNF-Def.Grp UNIV oa*) (*BNF-Def.Grp UNIV ou*) (*BNF-Def.Grp UNIV m*)
  = *BNF-Def.Grp UNIV* (*map-rest′ e ia iu oa ou m*)
  apply(*intro ext*)
  subgoal for *x y*
    apply(*cases x*; *cases y*; *clarsimp*)
    apply(*subst* (*2 4*) *eq-alt-conversep*)
    apply(*subst* (*2 3*) *eq-alt*)
    apply(*simp add*: *pmf.rel-Grp list.rel-Grp option.rel-Grp prod.rel-Grp rel-fun-conversep-grp-grp*)
    apply(*auto simp add*: *Grp-def spmf.map-id*[*abs-def*] *id-def*[*symmetric*])
    done
  done

end

type-synonym
  (*′s-rest*, *′event*, *′iadv-rest*, *′iusr-rest*, *′oadv-rest*, *′ousr-rest*) *rest-wstate* =
    (*′s-rest*, *′event*, *′iadv-rest*, *′iusr-rest*, *′oadv-rest*, *′ousr-rest*, *′s-rest*) *rest-scheme*

inductive *WT-rest* :: (*′iadv*, *′oadv*) $\mathcal{I}$ ⇒ (*′iusr*, *′ousr*) $\mathcal{I}$ ⇒ (*′s* ⇒ *bool*) ⇒ (*′s*, *′event*, *′iadv*, *′iusr*, *′oadv*, *′ousr*) *rest-wstate* ⇒ *bool*
  for $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I rest* **where**
  WT-rest $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I rest* **if**
  ⋀*s x y es s′*. ⟦ ((*y*, *es*), *s′*) ∈ *set-spmf* (*rfunc-adv rest s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-adv*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-adv x* ∧ *I s′*
  ⋀*s x y es s′*. ⟦ ((*y*, *es*), *s′*) ∈ *set-spmf* (*rfunc-usr rest s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-usr*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-usr x* ∧ *I s′*
  I (*rinit rest*)

lemma *WT-restD*:
  assumes *WT-rest* $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I rest*
  shows *WT-restD-rfunc-adv*: ⋀*s x y es s′*. ⟦ ((*y*, *es*), *s′*) ∈ *set-spmf* (*rfunc-adv rest s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-adv*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-adv x* ∧ *I s′*
    and *WT-restD-rfunc-usr*: ⋀*s x y es s′*. ⟦ ((*y*, *es*), *s′*) ∈ *set-spmf* (*rfunc-usr rest s x*); *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$*-usr*; *I s* ⟧ ⟹ *y* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$*-usr x* ∧ *I s′*
    and *WT-restD-rinit*: *I* (*rinit rest*)
  using *assms* **by**(*auto elim*!: *WT-rest.cases*)

abbreviation
  fuse-cfunc ::
    (*′o* ⇒ *′x*) ⇒ (*′s-core*, *′i*, *′o*) *oracle′* ⇒ (*′s-core* × *′s-rest*, *′i* , *′x*) *oracle′*
  where
  fuse-cfunc redirect cfunc state inp ≡ *do* {
    let handle = *map-prod redirect* (*prod.swap o Pair* (*snd state*));
    (*os-cfunc* :: *′o* × *′s-core*) ← *cfunc* (*fst state*) *inp*;
    return-spmf (*handle os-cfunc*)
  }

abbreviation

fuse-rfunc ::
  $('o \Rightarrow 'x) \Rightarrow ('s\text{-}rest, 'e, 'i, 'o)$ *eoracle* $\Rightarrow ('s\text{-}core, 'e)$ *handler* $\Rightarrow$
    $('s\text{-}core \times 's\text{-}rest, 'i , 'x)$ *oracle'*
where
fuse-rfunc redirect rfunc notify state inp $\equiv$
  bind-spmf
    (*rfunc* (*snd state*) *inp*)
    (λ((*o-rfunc*, *e-lst*), *s-rfunc*).
      bind-spmf
        (*foldl-spmf notify* (*return-spmf* (*fst state*)) *e-lst*)
        (λ*s-notify*. *return-spmf* (*redirect o-rfunc*, *s-notify*, *s-rfunc*)))

locale *fused-resource* =
  fixes
    core :: ('*s-core*, '*event*, '*iadv-core*, '*iusr-core*, '*oadv-core*, '*ousr-core*) *core* **and**
    core-init :: '*s-core*
begin

fun
  fuse ::
    ('*s-rest*, '*event*, '*iadv-rest*, '*iusr-rest*, '*oadv-rest*, '*ousr-rest*, '*m*) *rest-scheme* $\Rightarrow$
    ('*s-core* $\times$ '*s-rest*,
      ('*iadv-core* + '*iadv-rest*) + ('*iusr-core* + '*iusr-rest*),
      ('*oadv-core* + '*oadv-rest*) + ('*ousr-core* + '*ousr-rest*)) *oracle'*
  where
    fuse rest state (*Inl* (*Inl iadv-core*)) =
      fuse-cfunc (*Inl o Inl*) (*cfunc-adv core*) *state iadv-core*
  | *fuse rest state* (*Inl* (*Inr iadv-rest*)) =
      fuse-rfunc (*Inl o Inr*) (*rfunc-adv rest*) (*cpoke core*) *state iadv-rest*
  | *fuse rest state* (*Inr* (*Inl iusr-core*)) =
      fuse-cfunc (*Inr o Inl*) (*cfunc-usr core*) *state iusr-core*
  | *fuse rest state* (*Inr* (*Inr iusr-rest*)) =
      fuse-rfunc (*Inr o Inr*) (*rfunc-usr rest*) (*cpoke core*) *state iusr-rest*

case-of-simps *fuse-case*: *fused-resource.fuse.simps*

lemma *callee-invariant-on-fuse*:
  assumes *WT-core* $\mathcal{I}$*-adv-core* $\mathcal{I}$*-usr-core I-core core*
    and *WT-rest* $\mathcal{I}$*-adv-rest* $\mathcal{I}$*-usr-rest I-rest rest*
  shows *callee-invariant-on* (*fuse rest*) (*pred-prod I-core I-rest*) (($\mathcal{I}$*-adv-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-usr-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-rest*))
proof(*unfold-locales*, *goal-cases*)
  case (*1 s x y s'*)
  then show *?case* **using** *assms*
    by(*cases s*; *cases s'*)(*auto 4 4 dest*: *WT-restD WT-coreD WT-coreD-foldl-spmf-cpoke*)
next
  case (*2 s*)

show *?case*
apply(*rule WT-calleeI*)
subgoal for *x y s′* **using** *2 assms*
by (*cases* (*rest*, *s*, *x*) *rule*: *fuse.cases*) (*auto simp add*: *pred-prod-beta dest*: *WT-restD WT-coreD*)
done
qed

definition
resource ::
(*′s-rest*, *′event*, *′iadv-rest*, *′iusr-rest*, *′oadv-rest*, *′ousr-rest*) *rest-wstate* ⇒
((*′iadv-core* + *′iadv-rest*) + (*′iusr-core* + *′iusr-rest*),
(*′oadv-core* + *′oadv-rest*) + (*′ousr-core* + *′ousr-rest*)) *resource*
where
resource rest = *resource-of-oracle* (*fuse rest*) (*core-init*, *rinit rest*)

lemma *WT-resource* [*WT-intro*]:
assumes *WT-core* $\mathcal{I}$*-adv-core* $\mathcal{I}$*-usr-core I-core core*
and *WT-rest* $\mathcal{I}$*-adv-rest* $\mathcal{I}$*-usr-rest I-rest rest*
and *I-core core-init*
shows ($\mathcal{I}$*-adv-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-rest*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-usr-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr-rest*) ⊢*res resource rest* $\surd$
proof −
interpret *callee-invariant-on fuse rest pred-prod I-core I-rest* ($\mathcal{I}$*-adv-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-rest*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-usr-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr-rest*)
using *assms*(*1*,*2*) **by**(*rule callee-invariant-on-fuse*)
show *?thesis* **unfolding** *resource-def*
by(*rule WT-resource-of-oracle*)(*simp add*: *assms*(*3*) *WT-restD-rinit*[*OF assms*(*2*)])
qed

end

parametric-constant
fuse-parametric [*transfer-rule*]: *fused-resource.fuse-case*

4.4 More helpful construction functions

context
fixes
core1 :: (*′s-core1*, *′event1*, *′iadv-core1*, *′iusr-core1*, *′oadv-core1*, *′ousr-core1*) *core*
and
core2 :: (*′s-core2*, *′event2*, *′iadv-core2*, *′iusr-core2*, *′oadv-core2*, *′ousr-core2*) *core*
begin

primcorec *parallel-core* ::
(*′s-core1* × *′s-core2*, *′event1* + *′event2*,
′iadv-core1 + *′iadv-core2*, *′iusr-core1* + *′iusr-core2*,
′oadv-core1 + *′oadv-core2*, *′ousr-core1* + *′ousr-core2*) *core*
where

cpoke parallel-core = *parallel-handler* (*cpoke core1*) (*cpoke core2*)
| *cfunc-adv parallel-core* = *parallel-oracle* (*cfunc-adv core1*) (*cfunc-adv core2*)
| *cfunc-usr parallel-core* = *parallel-oracle* (*cfunc-usr core1*) (*cfunc-usr core2*)

end

context
fixes
cnv-adv :: *'s-adv* ⇒ *'iadv* ⇒ (*'oadv* × *'s-adv*, *'iadv-core*, *'oadv-core*) *gpv* **and**
cnv-usr :: *'s-usr* ⇒ *'iusr* ⇒ (*'ousr* × *'s-usr*, *'iusr-core*, *'ousr-core*) *gpv* **and**
core :: (*'s-core*, *'event*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
begin

primcorec
attach-core :: ((*'s-adv* × *'s-usr*) × *'s-core*, *'event*, *'iadv*, *'iusr*, *'oadv*, *'ousr*) *core*
where
cpoke attach-core = (λ(*s-advusr*, *s-core*) *event*.
map-spmf (λ*s-core'*. (*s-advusr*, *s-core'*)) (*cpoke core s-core event*))
| *cfunc-adv attach-core* = (λ((*s-adv*, *s-usr*), *s-core*) *iadv*.
map-spmf
(λ((*oadv*, *s-adv'*), *s-core'*). (*oadv*, ((*s-adv'*, *s-usr*), *s-core'*)))
(*exec-gpv* (*cfunc-adv core*) (*cnv-adv s-adv iadv*) *s-core*))
| *cfunc-usr attach-core* = (λ((*s-adv*, *s-usr*), *s-core*) *iusr*.
map-spmf
(λ((*ousr*, *s-usr'*), *s-core'*). (*ousr*, ((*s-adv*, *s-usr'*), *s-core'*)))
(*exec-gpv* (*cfunc-usr core*) (*cnv-usr s-usr iusr*) *s-core*))

end

lemma
attach-core-id-oracle-adv: *cfunc-adv* (*attach-core* 1_I *cnv core*) =
(λ(*s-cnv*, *s-core*) *q*. *map-spmf* (λ(*out*, *s-core'*). (*out*, *s-cnv*, *s-core'*)) (*cfunc-adv core s-core q*))
by(*simp add*: *id-oracle-def split-def map-spmf-conv-bind-spmf*)

lemma
attach-core-id-oracle-usr: *cfunc-usr* (*attach-core cnv* 1_I *core*) =
(λ(*s-cnv*, *s-core*) *q*. *map-spmf* (λ(*out*, *s-core'*). (*out*, *s-cnv*, *s-core'*)) (*cfunc-usr core s-core q*))
by(*simp add*: *id-oracle-def split-def map-spmf-conv-bind-spmf*)

context
fixes
rest1 :: (*'s-rest1*, *'event1*, *'iadv-rest1*, *'iusr-rest1*, *'oadv-rest1*, *'ousr-rest1*, *'more1*) *rest-scheme* **and**
rest2 :: (*'s-rest2*, *'event2*, *'iadv-rest2*, *'iusr-rest2*, *'oadv-rest2*, *'ousr-rest2*, *'more2*)

rest-scheme
begin

primcorec *parallel-rest* ::
(*'s-rest1* × *'s-rest2*, *'event1* + *'event2*, *'iadv-rest1* + *'iadv-rest2*, *'iusr-rest1* + *'iusr-rest2*,
'oadv-rest1 + *'oadv-rest2*, *'ousr-rest1* + *'ousr-rest2*, *'more1* × *'more2*) *rest-scheme*

where
rinit parallel-rest = (*rinit rest1*, *rinit rest2*)
| *rfunc-adv parallel-rest* = *parallel-eoracle* (*rfunc-adv rest1*) (*rfunc-adv rest2*)
| *rfunc-usr parallel-rest* = *parallel-eoracle* (*rfunc-usr rest1*) (*rfunc-usr rest2*)

end

lemma *WT-parallel-rest* [*WT-intro*]:
WT-rest (*I-adv1* $\oplus_{\mathcal{I}}$ *I-adv2*) (*I-usr1* $\oplus_{\mathcal{I}}$ *I-usr2*) (*pred-prod I1 I2*) (*parallel-rest rest1 rest2*)
if *WT-rest I-adv1 I-usr1 I1 rest1*
and *WT-rest I-adv2 I-usr2 I2 rest2*
by(*rule WT-rest.intros*)
(*auto 4 3 simp add*: *parallel-eoracle-def simp add*: *that*[*THEN WT-restD-rinit*] *dest*: *that*[*THEN WT-restD-rfunc-adv*] *that*[*THEN WT-restD-rfunc-usr*])

context
fixes
cnv-adv :: *'s-adv* ⇒ *'iadv* ⇒ (*'oadv* × *'s-adv*, *'iadv-rest*, *'oadv-rest*) *gpv* **and**
cnv-usr :: *'s-usr* ⇒ *'iusr* ⇒ (*'ousr* × *'s-usr*, *'iusr-rest*, *'ousr-rest*) *gpv* **and**
f-init :: *'more* ⇒ *'more'* **and**
rest :: (*'s-rest*, *'event*, *'iadv-rest*, *'iusr-rest*, *'oadv-rest*, *'ousr-rest*, *'more*) *rest-scheme*
begin

primcorec
attach-rest ::
((*'s-adv* × *'s-usr*) × *'s-rest*, *'event*, *'iadv*, *'iusr*, *'oadv*, *'ousr*, *'more'*) *rest-scheme*
where
rinit attach-rest = *f-init* (*rinit rest*)
| *rfunc-adv attach-rest* = (λ((*s-adv*, *s-usr*), *s-rest*) *iadv*.
let orc-of = λ*orc* (*s*, *es*) *q*. *map-spmf* (λ ((*out*, *e*), *s'*). (*out*, *s'*, *es* @ *e*)) (*orc s q*) *in*
let eorc-of = λ((*oadv*, *s-adv'*), (*s-rest'*, *es*)). ((*oadv*, *es*), ((*s-adv'*, *s-usr*), *s-rest'*)) *in*
map-spmf eorc-of (*exec-gpv* (*orc-of* (*rfunc-adv rest*)) (*cnv-adv s-adv iadv*) (*s-rest*, [])))
| *rfunc-usr attach-rest* = (λ((*s-adv*, *s-usr*), *s-rest*) *iusr*.
let orc-of = λ*orc* (*s*, *es*) *q*. *map-spmf* (λ ((*out*, *e*), *s'*). (*out*, *s'*, *es* @ *e*)) (*orc s q*) *in*
let eorc-of = λ((*ousr*, *s-usr'*), (*s-rest'*, *es*)). ((*ousr*, *es*), ((*s-adv*, *s-usr'*), *s-rest'*)) *in*

map-spmf eorc-of (exec-gpv (orc-of (rfunc-usr rest)) (cnv-usr s-usr iusr) (s-rest, [])))

end

lemma
attach-rest-id-oracle-adv: *rfunc-adv (attach-rest* 1_I *cnv f-init rest)* =
(λ(*s-cnv*, *s-core*) *q*. *map-spmf* (λ(*out*, *s-core'*). (*out*, *s-cnv*, *s-core'*)) (*rfunc-adv rest s-core q*))
by(*simp add*: *id-oracle-def split-def map-spmf-conv-bind-spmf fun-eq-iff*)

lemma
attach-rest-id-oracle-usr: *rfunc-usr (attach-rest cnv* 1_I *f-init rest)* =
(λ(*s-cnv*, *s-core*) *q*. *map-spmf* (λ(*out*, *s-core'*). (*out*, *s-cnv*, *s-core'*)) (*rfunc-usr rest s-core q*))
by(*simp add*: *id-oracle-def split-def map-spmf-conv-bind-spmf*)

5 Traces

type-synonym (*'event*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *trace-core* =
(*'event* + *'iadv-core* × *'oadv-core* + *'iusr-core* × *'ousr-core*) *list*
⇒ (*'event* ⇒ *real*)
× (*'iadv-core* ⇒ *'oadv-core spmf*)
× (*'iusr-core* ⇒ *'ousr-core spmf*)

context
fixes *core* :: (*'s-core*, *'event*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
begin

primrec *trace-core'* :: *'s-core spmf* ⇒ (*'event*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *trace-core* **where**
trace-core' S [] =
(λ*e*. *weight-spmf'* (*bind-spmf S* (λ*s*. *cpoke core s e*)),
λ*ia*. *bind-spmf S* (λ*s*. *map-spmf fst* (*cfunc-adv core s ia*)),
λ*iu*. *bind-spmf S* (λ*s*. *map-spmf fst* (*cfunc-usr core s iu*)))
| *trace-core' S* (*obs* # *tr*) = (*case obs of*
Inl e ⇒ *trace-core'* (*mk-lossless* (*bind-spmf S* (λ*s*. *cpoke core s e*))) *tr*
| *Inr* (*Inl* (*ia*, *oa*)) ⇒ *trace-core'* (*cond-spmf-fst* (*bind-spmf S* (λ*s*. *cfunc-adv core s ia*)) *oa*) *tr*
| *Inr* (*Inr* (*iu*, *ou*)) ⇒ *trace-core'* (*cond-spmf-fst* (*bind-spmf S* (λ*s*. *cfunc-usr core s iu*)) *ou*) *tr*
)

end

declare *trace-core'.simps* [*simp del*]
case-of-simps *trace-core'-unfold*: *trace-core'.simps*[*unfolded weight-spmf'-def*]
simps-of-case *trace-core'-simps* [*simp*]: *trace-core'-unfold*

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

lemma *trace-core′-parametric* [*transfer-rule*]:
(*rel-core′ S E IA IU* (=) (=) ===>
rel-spmf S ===>
list-all2 (*rel-sum E* (*rel-sum* (*rel-prod IA* (=)) (*rel-prod IU* (=)))) ===>
rel-prod (*E* ===> (=)) (*rel-prod* (*IA* ===> (=)) (*IU* ===> (=))))
trace-core′ trace-core′
unfolding *trace-core′-def* **by** *transfer-prover*

definition *trace-core-eq*
:: (*′s-core*, *′event*, *′iadv-core*, *′iusr-core*, *′oadv-core*, *′ousr-core*) *core*
⇒ (*′s-core′*, *′event*, *′iadv-core*, *′iusr-core*, *′oadv-core*, *′ousr-core*) *core*
⇒ *′event set* ⇒ *′iadv-core set* ⇒ *′iusr-core set*
⇒ *′s-core spmf* ⇒ *′s-core′ spmf* ⇒ *bool* **where**
trace-core-eq core1 core2 E IA IU p q ⟷
(∀ *tr*. *set tr* ⊆ *E* <+> (*IA* × *UNIV*) <+> (*IU* × *UNIV*) ⟶
rel-prod (*eq-onp* (λ*e*. *e* ∈ *E*) ===> (=)) (*rel-prod* (*eq-onp* (λ*ia*. *ia* ∈ *IA*) ===> (=)) (*eq-onp* (λ*iu*. *iu* ∈ *IU*) ===> (=)))
(*trace-core′ core1 p tr*) (*trace-core′ core2 q tr*))

end

lemma *trace-core-eqD*:
assumes *trace-core-eq core1 core2 E IA IU p q*
and *set tr* ⊆ *E* <+> (*IA* × *UNIV*) <+> (*IU* × *UNIV*)
shows *trace-core-eqD-cpoke*:
e ∈ *E* ⟹ *fst* (*trace-core′ core1 p tr*) *e* = *fst* (*trace-core′ core2 q tr*) *e*
and *trace-core-eqD-cfunc-adv*:
ia ∈ *IA* ⟹ *fst* (*snd* (*trace-core′ core1 p tr*)) *ia* = *fst* (*snd* (*trace-core′ core2 q tr*)) *ia*
and *trace-core-eqD-cfunc-usr*:
iu ∈ *IU* ⟹ *snd* (*snd* (*trace-core′ core1 p tr*)) *iu* = *snd* (*snd* (*trace-core′ core2 q tr*)) *iu*
using *assms* **by**(*auto simp add*: *trace-core-eq-def rel-fun-def eq-onp-def rel-prod-sel*)

lemma *trace-core-eqI*:
assumes ⋀*tr e*. ⟦ *set tr* ⊆ *E* <+> (*IA* × *UNIV*) <+> (*IU* × *UNIV*); *e* ∈ *E* ⟧
⟹ *fst* (*trace-core′ core1 p tr*) *e* = *fst* (*trace-core′ core2 q tr*) *e*
and ⋀*tr ia*. ⟦ *set tr* ⊆ *E* <+> (*IA* × *UNIV*) <+> (*IU* × *UNIV*); *ia* ∈ *IA* ⟧
⟹ *fst* (*snd* (*trace-core′ core1 p tr*)) *ia* = *fst* (*snd* (*trace-core′ core2 q tr*)) *ia*
and ⋀*tr iu*. ⟦ *set tr* ⊆ *E* <+> (*IA* × *UNIV*) <+> (*IU* × *UNIV*); *iu* ∈ *IU* ⟧
⟹ *snd* (*snd* (*trace-core′ core1 p tr*)) *iu* = *snd* (*snd* (*trace-core′ core2 q tr*)) *iu*
shows *trace-core-eq core1 core2 E IA IU p q*
using *assms* **by**(*auto simp add*: *trace-core-eq-def rel-fun-def eq-onp-def rel-prod-sel*)

lemma *trace-core-return-pmf-None* [*simp*]:
trace-core′ core (*return-pmf None*) *tr* = (λ-. *0*, λ-. *return-pmf None*, λ-. *return-pmf*

```
None)
  by(induction tr)(simp-all add: trace-core'.simps split: sum.split)

lemma rel-core'-into-trace-core-eq: trace-core-eq core core' E IA IU p q
  if rel-core' S (eq-onp (λe. e ∈ E)) (eq-onp (λia. ia ∈ IA)) (eq-onp (λiu. iu ∈ IU))
(=) (=) core core'
     rel-spmf S p q
  using trace-core'-parametric[THEN rel-funD, THEN rel-funD, OF that]
  unfolding trace-core-eq-def
  apply(intro strip)
  subgoal for tr
   apply(simp add: eq-onp-True[symmetric] prod.rel-eq-onp sum.rel-eq-onp list.rel-eq-onp)
    apply(auto 4 3 simp add: eq-onp-def list-all-iff dest: rel-funD[where x=tr and
y=tr])
    done
  done

lemma trace-core-eq-simI:
  fixes core1 :: ('s-core, 'event, 'iadv-core, 'iusr-core, 'oadv-core, 'ousr-core) core
    and core2 :: ('s-core', 'event, 'iadv-core, 'iusr-core, 'oadv-core, 'ousr-core) core
    and S :: 's-core spmf ⇒ 's-core' spmf ⇒ bool
  assumes start: S p q
    and step-cpoke: ⋀p q e. ⟦ S p q; e ∈ E ⟧ ⟹
      weight-spmf (bind-spmf p (λs. cpoke core1 s e)) = weight-spmf (bind-spmf q
(λs. cpoke core2 s e))
    and sim-cpoke: ⋀p q e. ⟦ S p q; e ∈ E ⟧ ⟹
      S (mk-lossless (bind-spmf p (λs. cpoke core1 s e))) (mk-lossless (bind-spmf q
(λs. cpoke core2 s e)))
    and step-cfunc-adv: ⋀p q ia. ⟦ S p q; ia ∈ IA ⟧ ⟹
      bind-spmf p (λs1. map-spmf fst (cfunc-adv core1 s1 ia)) = bind-spmf q (λs2.
map-spmf fst (cfunc-adv core2 s2 ia))
    and sim-cfunc-adv: ⋀p q ia s1 s2 s1' s2' oa. ⟦ S p q; ia ∈ IA;
      s1 ∈ set-spmf p; s2 ∈ set-spmf q; (oa, s1') ∈ set-spmf (cfunc-adv core1 s1 ia);
(oa, s2') ∈ set-spmf (cfunc-adv core2 s2 ia) ⟧
    ⟹ S (cond-spmf-fst (bind-spmf p (λs1. cfunc-adv core1 s1 ia)) oa) (cond-spmf-fst
(bind-spmf q (λs2. cfunc-adv core2 s2 ia)) oa)
    and step-cfunc-usr: ⋀p q iu. ⟦ S p q; iu ∈ IU ⟧ ⟹
      bind-spmf p (λs1. map-spmf fst (cfunc-usr core1 s1 iu)) = bind-spmf q (λs2.
map-spmf fst (cfunc-usr core2 s2 iu))
    and sim-cfunc-usr: ⋀p q iu s1 s2 s1' s2' ou. ⟦ S p q; iu ∈ IU;
      s1 ∈ set-spmf p; s2 ∈ set-spmf q; (ou, s1') ∈ set-spmf (cfunc-usr core1 s1 iu);
(ou, s2') ∈ set-spmf (cfunc-usr core2 s2 iu) ⟧
    ⟹ S (cond-spmf-fst (bind-spmf p (λs1. cfunc-usr core1 s1 iu)) ou) (cond-spmf-fst
(bind-spmf q (λs2. cfunc-usr core2 s2 iu)) ou)
  shows trace-core-eq core1 core2 E IA IU p q
proof(rule trace-core-eqI)
  fix tr :: ('event + 'iadv-core × 'oadv-core + 'iusr-core × 'ousr-core) list
  assume set tr ⊆ E <+> IA × UNIV <+> IU × UNIV
  then have (∀ e∈E. fst (trace-core' core1 p tr) e = fst (trace-core' core2 q tr) e)
```

∧
(∀ *ia*∈*IA*. *fst* (*snd* (*trace-core′ core1 p tr*)) *ia* = *fst* (*snd* (*trace-core′ core2 q tr*)) *ia*) ∧
(∀ *iu*∈*IU*. *snd* (*snd* (*trace-core′ core1 p tr*)) *iu* = *snd* (*snd* (*trace-core′ core2 q tr*)) *iu*)
using *start*
proof(*induction tr arbitrary*: *p q*)
case *Nil*
then show *?case* **by**(*simp add*: *step-cpoke step-cfunc-adv step-cfunc-usr*)
next
case (*Cons a tr*)
from *Cons.prems*(*1*) **have** *tr*: *set tr* ⊆ *E* <+> *IA* × *UNIV* <+> *IU* × *UNIV* **by** *simp*
from *Cons.prems*(*1*)
consider (*cpoke*) *e* **where** *a* = *Inl e e* ∈ *E*
| (*cfunc-adv*) *ia oa* **where** *a* = *Inr* (*Inl* (*ia*, *oa*)) *ia* ∈ *IA*
| (*cfunc-usr*) *iu ou* **where** *a* = *Inr* (*Inr* (*iu*, *ou*)) *iu* ∈ *IU* **by** *auto*
then show *?case*
proof *cases*
case *cpoke*
then show *?thesis* **using** *tr Cons.prems*(*2*) **by**(*auto simp add*: *sim-cpoke intro*!: *Cons.IH*)
next
case *cfunc-adv*
let *?p* = *bind-spmf p* (λ*s1*. *cfunc-adv core1 s1 ia*)
let *?q* = *bind-spmf q* (λ*s2*. *cfunc-adv core2 s2 ia*)
show *?thesis*
proof(*cases oa* ∈ *fst* ‘ *set-spmf ?p*)
case *True*
with *step-cfunc-adv*[*OF Cons.prems*(*2*) *cfunc-adv*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]]
have *oa* ∈ *fst* ‘ *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*)
then show *?thesis* **using** *True Cons.prems cfunc-adv*
by(*clarsimp*)(*rule Cons.IH*; *blast intro*: *sim-cfunc-adv*)
next
case *False*
hence *cond-spmf-fst ?p oa* = *return-pmf None* **by** *simp*
moreover
from *step-cfunc-adv*[*OF Cons.prems*(*2*) *cfunc-adv*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]] *False*
have *oa′*: *oa* ∉ *fst* ‘ *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*) *simp*
hence *cond-spmf-fst ?q oa* = *return-pmf None* **by** *simp*
ultimately show *?thesis* **using** *cfunc-adv* **by**(*simp del*: *cond-spmf-fst-eq-return-None*)
qed
next
case *cfunc-usr*

let *?p* = *bind-spmf p* (*λs1. cfunc-usr core1 s1 iu*)
let *?q* = *bind-spmf q* (*λs2. cfunc-usr core2 s2 iu*)
show *?thesis*
proof(*cases ou* ∈ *fst* ‘ *set-spmf ?p*)
case *True*
with *step-cfunc-usr*[*OF Cons.prems*(*2*) *cfunc-usr*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]]
have *ou* ∈ *fst* ‘ *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*)
then show *?thesis* **using** *True Cons.prems cfunc-usr*
by(*clarsimp*)(*rule Cons.IH*; *blast intro*: *sim-cfunc-usr*)
next
case *False*
hence *cond-spmf-fst ?p ou* = *return-pmf None* **by** *simp*
moreover
from *step-cfunc-usr*[*OF Cons.prems*(*2*) *cfunc-usr*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]] *False*
have *oa*′: *ou* ∉ *fst* ‘ *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*) *simp*
hence *cond-spmf-fst ?q ou* = *return-pmf None* **by** *simp*
ultimately show *?thesis* **using** *cfunc-usr* **by**(*simp del*: *cond-spmf-fst-eq-return-None*)
qed
qed
qed
then show *e* ∈ *E* ⟹ *fst* (*trace-core*′ *core1 p tr*) *e* = *fst* (*trace-core*′ *core2 q tr*) *e*
and *ia* ∈ *IA* ⟹ *fst* (*snd* (*trace-core*′ *core1 p tr*)) *ia* = *fst* (*snd* (*trace-core*′ *core2 q tr*)) *ia*
and *iu* ∈ *IU* ⟹ *snd* (*snd* (*trace-core*′ *core1 p tr*)) *iu* = *snd* (*snd* (*trace-core*′ *core2 q tr*)) *iu*
for *e ia iu* **by** *blast+*
qed

context
fixes *core* :: (′*s-core*, ′*event*, ′*iadv-core*, ′*iusr-core*, ′*oadv-core*, ′*ousr-core*) *core*
begin

fun *trace-core-aux*
:: ′*s-core spmf* ⇒ (′*event* + ′*iadv-core* × ′*oadv-core* + ′*iusr-core* × ′*ousr-core*) *list* ⇒ ′*s-core spmf* **where**
trace-core-aux p [] = *p*
| *trace-core-aux p* (*Inl e* # *tr*) = *trace-core-aux* (*mk-lossless* (*bind-spmf p* (*λs. cpoke core s e*))) *tr*
| *trace-core-aux p* (*Inr* (*Inl* (*ia*, *oa*)) # *tr*) = *trace-core-aux* (*cond-spmf-fst* (*bind-spmf p* (*λs. cfunc-adv core s ia*)) *oa*) *tr*
| *trace-core-aux p* (*Inr* (*Inr* (*iu*, *ou*)) # *tr*) = *trace-core-aux* (*cond-spmf-fst* (*bind-spmf p* (*λs. cfunc-usr core s iu*)) *ou*) *tr*

end

lemma *trace-core-conv-trace-core-aux*:
trace-core′ core p tr =
(*λe. weight-spmf* (*bind-spmf* (*trace-core-aux core p tr*) (*λs. cpoke core s e*)),
λia. bind-spmf (*trace-core-aux core p tr*) (*λs. map-spmf fst* (*cfunc-adv core s ia*)),
λiu. bind-spmf (*trace-core-aux core p tr*) (*λs. map-spmf fst* (*cfunc-usr core s iu*)))
by(*induction p tr rule*: *trace-core-aux.induct*) *simp-all*

lemma *trace-core-aux-append*:
trace-core-aux core p (*tr* @ *tr′*) = *trace-core-aux core* (*trace-core-aux core p tr*) *tr′*
by(*induction p tr rule*: *trace-core-aux.induct*) *auto*

inductive *trace-core-closure*
:: (*′s-core*, *′event*, *′iadv-core*, *′iusr-core*, *′oadv-core*, *′ousr-core*) *core*
⇒ (*′s-core′*, *′event*, *′iadv-core*, *′iusr-core*, *′oadv-core*, *′ousr-core*) *core*
⇒ *′event set* ⇒ *′iadv-core set* ⇒ *′iusr-core set*
⇒ *′s-core spmf* ⇒ *′s-core′ spmf* ⇒ *′s-core spmf* ⇒ *′s-core′ spmf* ⇒ *bool*
for *core1 core2 E IA IU p q* **where**
trace-core-closure core1 core2 E IA IU p q (*trace-core-aux core1 p tr*) (*trace-core-aux core2 q tr*)
if *set tr* ⊆ *E* <+> *IA* × *UNIV* <+> *IU* × *UNIV*

lemma *trace-core-closure-start*: *trace-core-closure core1 core2 E IA IU p q p q*
by(*simp add*: *trace-core-closure.simps exI*[**where** *x*=[]])

lemma *trace-core-closure-step*:
assumes *trace-core-eq core1 core2 E IA IU p q*
and *trace-core-closure core1 core2 E IA IU p q p′ q′*
shows *trace-core-closure-step-cpoke*:
e ∈ *E* ⟹ *weight-spmf* (*bind-spmf p′* (*λs. cpoke core1 s e*)) = *weight-spmf* (*bind-spmf q′* (*λs. cpoke core2 s e*))
(**is** *PROP ?thesis1*)
and *trace-core-closure-step-cfunc-adv*:
ia ∈ *IA* ⟹ *bind-spmf p′* (*λs1. map-spmf fst* (*cfunc-adv core1 s1 ia*)) = *bind-spmf q′* (*λs2. map-spmf fst* (*cfunc-adv core2 s2 ia*))
(**is** *PROP ?thesis2*)
and *trace-core-closure-step-cfunc-usr*:
iu ∈ *IU* ⟹ *bind-spmf p′* (*λs1. map-spmf fst* (*cfunc-usr core1 s1 iu*)) = *bind-spmf q′* (*λs2. map-spmf fst* (*cfunc-usr core2 s2 iu*))
(**is** *PROP ?thesis3*)
proof −
from *assms*(*2*) **obtain** *tr* **where** *p*: *p′* = *trace-core-aux core1 p tr*
and *q*: *q′* = *trace-core-aux core2 q tr*
and *tr*: *set tr* ⊆ *E* <+> *IA* × *UNIV* <+> *IU* × *UNIV* **by** *cases*
from *trace-core-eqD*[*OF assms*(*1*) *tr*] *p q*

show *PROP ?thesis1* **and** *PROP ?thesis2 PROP ?thesis3*
by(*simp-all add*: *trace-core-conv-trace-core-aux*)
qed

lemma *trace-core-closure-sim*:
fixes *core1 core2 E IA IU p q*
defines *S* ≡ *trace-core-closure core1 core2 E IA IU p q*
assumes *S p′ q′*
shows *trace-core-closure-sim-cpoke*:
e ∈ *E* ⟹ *S* (*mk-lossless* (*bind-spmf p′* (*λs. cpoke core1 s e*))) (*mk-lossless* (*bind-spmf q′* (*λs. cpoke core2 s e*)))
(**is** *PROP ?thesis1*)
and *trace-core-closure-sim-cfunc-adv*: *ia* ∈ *IA*
⟹ *S* (*cond-spmf-fst* (*bind-spmf p′* (*λs1. cfunc-adv core1 s1 ia*)) *oa*)
(*cond-spmf-fst* (*bind-spmf q′* (*λs2. cfunc-adv core2 s2 ia*)) *oa*)
(**is** *PROP ?thesis2*)
and *trace-core-closure-sim-cfunc-usr*: *iu* ∈ *IU*
⟹ *S* (*cond-spmf-fst* (*bind-spmf p′* (*λs1. cfunc-usr core1 s1 iu*)) *ou*)
(*cond-spmf-fst* (*bind-spmf q′* (*λs2. cfunc-usr core2 s2 iu*)) *ou*)
(**is** *PROP ?thesis3*)
proof −
from *assms*(*2*) **obtain** *tr* **where** *p*: *p′* = *trace-core-aux core1 p tr*
and *q*: *q′* = *trace-core-aux core2 q tr*
and *tr*: *set tr* ⊆ *E* <+> *IA* × *UNIV* <+> *IU* × *UNIV* **unfolding** *S-def* **by** *cases*
show *PROP ?thesis1* **using** *p q tr*
by(*auto simp add*: *S-def trace-core-closure.simps trace-core-aux-append intro*!: *exI*[**where** *x*=*tr* @ [*Inl* -]])
show *PROP ?thesis2* **using** *p q tr*
by(*auto simp add*: *S-def trace-core-closure.simps trace-core-aux-append intro*!: *exI*[**where** *x*=*tr* @ [*Inr* (*Inl* (-, -))]])
show *PROP ?thesis3* **using** *p q tr*
by(*auto simp add*: *S-def trace-core-closure.simps trace-core-aux-append intro*!: *exI*[**where** *x*=*tr* @ [*Inr* (*Inr* (-, -))]])
qed

proposition *trace-core-eq-complete*:
assumes *trace-core-eq core1 core2 E IA IU p q*
obtains *S*
where *S p q*
and ⋀*p q e*. ⟦ *S p q*; *e* ∈ *E* ⟧ ⟹
weight-spmf (*bind-spmf p* (*λs. cpoke core1 s e*)) = *weight-spmf* (*bind-spmf q* (*λs. cpoke core2 s e*))
and ⋀*p q e*. ⟦ *S p q*; *e* ∈ *E* ⟧ ⟹
S (*mk-lossless* (*bind-spmf p* (*λs. cpoke core1 s e*))) (*mk-lossless* (*bind-spmf q* (*λs. cpoke core2 s e*)))
and ⋀*p q ia*. ⟦ *S p q*; *ia* ∈ *IA* ⟧ ⟹
bind-spmf p (*λs1. map-spmf fst* (*cfunc-adv core1 s1 ia*)) = *bind-spmf q* (*λs2. map-spmf fst* (*cfunc-adv core2 s2 ia*))

and ⋀*p q ia oa*. ⟦ *S p q*; *ia* ∈ *IA* ⟧
⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s1*. *cfunc-adv core1 s1 ia*)) *oa*) (*cond-spmf-fst* (*bind-spmf q* (λ*s2*. *cfunc-adv core2 s2 ia*)) *oa*)
and ⋀*p q iu*. ⟦ *S p q*; *iu* ∈ *IU* ⟧ ⟹
bind-spmf p (λ*s1*. *map-spmf fst* (*cfunc-usr core1 s1 iu*)) = *bind-spmf q* (λ*s2*. *map-spmf fst* (*cfunc-usr core2 s2 iu*))
and ⋀*p q iu ou*. ⟦ *S p q*; *iu* ∈ *IU* ⟧
⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s1*. *cfunc-usr core1 s1 iu*)) *ou*) (*cond-spmf-fst* (*bind-spmf q* (λ*s2*. *cfunc-usr core2 s2 iu*)) *ou*)
proof −
show *thesis*
by(*rule that*[**where** *S*=*trace-core-closure core1 core2 E IA IU p q*])
(*auto intro*: *trace-core-closure-start trace-core-closure-step*[*OF assms*] *trace-core-closure-sim*)
qed

type-synonym (′*event*, ′*iadv-rest*, ′*iusr-rest*, ′*oadv-rest*, ′*ousr-rest*) *trace-rest* =
(′*iadv-rest* × ′*oadv-rest* × ′*event list* + ′*iusr-rest* × ′*ousr-rest* × ′*event list*) *list*
⇒ (′*iadv-rest* ⇒ (′*oadv-rest* × ′*event list*) *spmf*)
× (′*iusr-rest* ⇒ (′*ousr-rest* × ′*event list*) *spmf*)

context
fixes *rest* :: (′*s-rest*, ′*event*, ′*iadv-rest*, ′*iusr-rest*, ′*oadv-rest*, ′*ousr-rest*, ′*more*) *rest-scheme*
begin

primrec *trace-rest′* :: ′*s-rest spmf* ⇒ (′*event*, ′*iadv-rest*, ′*iusr-rest*, ′*oadv-rest*, ′*ousr-rest*) *trace-rest* **where**
trace-rest′ S [] =
(λ*ia*. *bind-spmf S* (λ*s*. *map-spmf fst* (*rfunc-adv rest s ia*)),
λ*iu*. *bind-spmf S* (λ*s*. *map-spmf fst* (*rfunc-usr rest s iu*)))
| *trace-rest′ S* (*obs* # *tr*) = (*case obs of*
Inl (*ia*, *oa*) ⇒ *trace-rest′* (*cond-spmf-fst* (*bind-spmf S* (λ*s*. *rfunc-adv rest s ia*)) *oa*) *tr*
| *Inr* (*iu*, *ou*) ⇒ *trace-rest′* (*cond-spmf-fst* (*bind-spmf S* (λ*s*. *rfunc-usr rest s iu*)) *ou*) *tr*)

end

declare *trace-rest′.simps* [*simp del*]
case-of-simps *trace-rest′-unfold*: *trace-rest′.simps*
simps-of-case *trace-rest′-simps* [*simp*]: *trace-rest′-unfold*

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

lemma *trace-rest′-parametric* [*transfer-rule*]:
(*rel-rest′ S* (=) *IA IU* (=) (=) *M* ===> *rel-spmf S* ===>

```
    list-all2 (rel-sum (rel-prod IA (=)) (rel-prod IU (=))) ===>
    rel-prod (IA ===> (=)) (IU ===> (=)))
    trace-rest′ trace-rest′
  unfolding trace-rest′-def by transfer-prover

definition trace-rest-eq
  :: ('s-rest, 'event, 'iadv-rest, 'iusr-rest, 'oadv-rest, 'ousr-rest, 'more1) rest-scheme
  ⇒ ('s-rest′, 'event, 'iadv-rest, 'iusr-rest, 'oadv-rest, 'ousr-rest, 'more2) rest-scheme
  ⇒ 'iadv-rest set ⇒ 'iusr-rest set
  ⇒ 's-rest spmf ⇒ 's-rest′ spmf ⇒ bool where
  trace-rest-eq rest1 rest2 IA IU p q ⟷
  (∀ tr. set tr ⊆ (IA × UNIV) <+> (IU × UNIV) ⟶
   rel-prod (eq-onp (λia. ia ∈ IA) ===> (=)) (eq-onp (λiu. iu ∈ IU) ===> (=))
     (trace-rest′ rest1 p tr) (trace-rest′ rest2 q tr))

end

lemma trace-rest-eqD:
  assumes trace-rest-eq rest1 rest2 IA IU p q
    and set tr ⊆ (IA × UNIV) <+> (IU × UNIV)
  shows trace-rest-eqD-rfunc-adv:
    ia ∈ IA ⟹ fst (trace-rest′ rest1 p tr) ia = fst (trace-rest′ rest2 q tr) ia
    and trace-rest-eqD-rfunc-usr:
    iu ∈ IU ⟹ snd (trace-rest′ rest1 p tr) iu = snd (trace-rest′ rest2 q tr) iu
 using assms by(auto simp add: trace-rest-eq-def rel-fun-def rel-prod-sel eq-onp-def)

lemma trace-rest-eqI:
  assumes ⋀tr ia. ⟦ set tr ⊆ (IA × UNIV) <+> (IU × UNIV); ia ∈ IA ⟧
     ⟹ fst (trace-rest′ rest1 p tr) ia = fst (trace-rest′ rest2 q tr) ia
    and ⋀tr iu. ⟦ set tr ⊆ (IA × UNIV) <+> (IU × UNIV); iu ∈ IU ⟧
      ⟹ snd (trace-rest′ rest1 p tr) iu = snd (trace-rest′ rest2 q tr) iu
  shows trace-rest-eq rest1 rest2 IA IU p q
 using assms by(auto simp add: trace-rest-eq-def rel-fun-def eq-onp-def rel-prod-sel)

lemma trace-rest-return-pmf-None [simp]:
   trace-rest′ rest (return-pmf None) tr = (λ-. return-pmf None, λ-. return-pmf
None)
  by(induction tr)(simp-all add: trace-rest′.simps split: sum.split)

lemma rel-rest′-into-trace-rest-eq: trace-rest-eq rest rest′ IA IU p q
  if rel-rest′ S (=) (eq-onp (λia. ia ∈ IA)) (eq-onp (λiu. iu ∈ IU)) (=) (=) M rest
rest′
     rel-spmf S p q
  using trace-rest′-parametric[THEN rel-funD, THEN rel-funD, OF that]
  unfolding trace-rest-eq-def
  apply(intro strip)
  subgoal for tr
   apply(simp add: eq-onp-True[symmetric] prod.rel-eq-onp sum.rel-eq-onp list.rel-eq-onp)
    apply(auto 4 3 simp add: eq-onp-def list-all-iff dest: rel-funD[where x=tr and
```

98

y=*tr*])
  done
 done

lemma *trace-rest-eq-simI*:
 fixes *rest1* :: ($'$*s-rest*, $'$*event*, $'$*iadv-rest*, $'$*iusr-rest*, $'$*oadv-rest*, $'$*ousr-rest*, $'$*more*) *rest-scheme*
  and *rest2* :: ($'$*s-rest*$'$, $'$*event*, $'$*iadv-rest*, $'$*iusr-rest*, $'$*oadv-rest*, $'$*ousr-rest*, $'$*more*) *rest-scheme*
  and *S* :: $'$*s-rest spmf* ⇒ $'$*s-rest*$'$ *spmf* ⇒ *bool*
 assumes *start*: *S p q*
  and *step-rfunc-adv*: ⋀*p q ia*. ⟦ *S p q*; *ia* ∈ *IA* ⟧ ⟹
   bind-spmf p (λ*s1*. *map-spmf fst* (*rfunc-adv rest1 s1 ia*)) = *bind-spmf q* (λ*s2*. *map-spmf fst* (*rfunc-adv rest2 s2 ia*))
  and *sim-rfunc-adv*: ⋀*p q ia s1 s2 s1*$'$ *s2*$'$ *oa*. ⟦ *S p q*; *ia* ∈ *IA*;
   s1 ∈ *set-spmf p*; *s2* ∈ *set-spmf q*; (*oa*, *s1*$'$) ∈ *set-spmf* (*rfunc-adv rest1 s1 ia*); (*oa*, *s2*$'$) ∈ *set-spmf* (*rfunc-adv rest2 s2 ia*) ⟧
  ⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s1*. *rfunc-adv rest1 s1 ia*)) *oa*) (*cond-spmf-fst* (*bind-spmf q* (λ*s2*. *rfunc-adv rest2 s2 ia*)) *oa*)
  and *step-rfunc-usr*: ⋀*p q iu*. ⟦ *S p q*; *iu* ∈ *IU* ⟧ ⟹
   bind-spmf p (λ*s1*. *map-spmf fst* (*rfunc-usr rest1 s1 iu*)) = *bind-spmf q* (λ*s2*. *map-spmf fst* (*rfunc-usr rest2 s2 iu*))
  and *sim-rfunc-usr*: ⋀*p q iu s1 s2 s1*$'$ *s2*$'$ *ou*. ⟦ *S p q*; *iu* ∈ *IU*;
   s1 ∈ *set-spmf p*; *s2* ∈ *set-spmf q*; (*ou*, *s1*$'$) ∈ *set-spmf* (*rfunc-usr rest1 s1 iu*); (*ou*, *s2*$'$) ∈ *set-spmf* (*rfunc-usr rest2 s2 iu*) ⟧
  ⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (λ*s1*. *rfunc-usr rest1 s1 iu*)) *ou*) (*cond-spmf-fst* (*bind-spmf q* (λ*s2*. *rfunc-usr rest2 s2 iu*)) *ou*)
 shows *trace-rest-eq rest1 rest2 IA IU p q*
proof(*rule trace-rest-eqI*)
 fix *tr* :: ($'$*iadv-rest* × $'$*oadv-rest* × $'$*event list* + $'$*iusr-rest* × $'$*ousr-rest* × $'$*event list*) *list*
 assume *set tr* ⊆ *IA* × *UNIV* <+> *IU* × *UNIV*
 then have (∀ *ia*∈*IA*. *fst* (*trace-rest*$'$ *rest1 p tr*) *ia* = *fst* (*trace-rest*$'$ *rest2 q tr*) *ia*) ∧
    (∀ *iu*∈*IU*. *snd* (*trace-rest*$'$ *rest1 p tr*) *iu* = *snd* (*trace-rest*$'$ *rest2 q tr*) *iu*)
  using *start*
 proof(*induction tr arbitrary*: *p q*)
  case *Nil*
  then show *?case* **by**(*simp add*: *step-rfunc-adv step-rfunc-usr*)
 next
  case (*Cons a tr*)
  from *Cons.prems*(*1*) **have** *tr*: *set tr* ⊆ *IA* × *UNIV* <+> *IU* × *UNIV* **by** *simp*
  from *Cons.prems*(*1*)
  consider (*rfunc-adv*) *ia oa* **where** *a* = *Inl* (*ia*, *oa*) *ia* ∈ *IA*
   | (*rfunc-usr*) *iu ou* **where** *a* = *Inr* (*iu*, *ou*) *iu* ∈ *IU* **by** *auto*
  then show *?case*
  proof *cases*
   case *rfunc-adv*
   let *?p* = *bind-spmf p* (λ*s1*. *rfunc-adv rest1 s1 ia*)

let *?q* = *bind-spmf q* (λ*s2*. *rfunc-adv rest2 s2 ia*)
show *?thesis*
proof(*cases oa* ∈ *fst* ' *set-spmf ?p*)
case *True*
with *step-rfunc-adv*[*OF Cons.prems*(*2*) *rfunc-adv*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]]
have *oa* ∈ *fst* ' *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*)
then show *?thesis* **using** *True Cons.prems rfunc-adv*
by(*clarsimp*)(*rule Cons.IH*; *blast intro*: *sim-rfunc-adv*)
next
case *False*
hence *cond-spmf-fst ?p oa* = *return-pmf None* **by** *simp*
moreover
from *step-rfunc-adv*[*OF Cons.prems*(*2*) *rfunc-adv*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]] *False*
have *oa'*: *oa* ∉ *fst* ' *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*) *simp*
hence *cond-spmf-fst ?q oa* = *return-pmf None* **by** *simp*
ultimately show *?thesis* **using** *rfunc-adv* **by**(*simp del*: *cond-spmf-fst-eq-return-None*)
qed
next
case *rfunc-usr*
let *?p* = *bind-spmf p* (λ*s1*. *rfunc-usr rest1 s1 iu*)
let *?q* = *bind-spmf q* (λ*s2*. *rfunc-usr rest2 s2 iu*)
show *?thesis*
proof(*cases ou* ∈ *fst* ' *set-spmf ?p*)
case *True*
with *step-rfunc-usr*[*OF Cons.prems*(*2*) *rfunc-usr*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]]
have *ou* ∈ *fst* ' *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*)
then show *?thesis* **using** *True Cons.prems rfunc-usr*
by(*clarsimp*)(*rule Cons.IH*; *blast intro*: *sim-rfunc-usr*)
next
case *False*
hence *cond-spmf-fst ?p ou* = *return-pmf None* **by** *simp*
moreover
from *step-rfunc-usr*[*OF Cons.prems*(*2*) *rfunc-usr*(*2*), *THEN arg-cong*[**where** *f*=*set-spmf*]] *False*
have *oa'*: *ou* ∉ *fst* ' *set-spmf ?q*
unfolding *set-map-spmf*[*symmetric*] **by**(*simp only*: *map-bind-spmf o-def*) *simp*
hence *cond-spmf-fst ?q ou* = *return-pmf None* **by** *simp*
ultimately show *?thesis* **using** *rfunc-usr* **by**(*simp del*: *cond-spmf-fst-eq-return-None*)
qed
qed
qed

then show *ia* ∈ *IA* ⟹ *fst* (*trace-rest′ rest1 p tr*) *ia* = *fst* (*trace-rest′ rest2 q tr*) *ia*
and *iu* ∈ *IU* ⟹ *snd* (*trace-rest′ rest1 p tr*) *iu* = *snd* (*trace-rest′ rest2 q tr*) *iu*
for *ia iu* **by** *blast+*
qed

context
fixes *rest* :: (*′s-rest*, *′event*, *′iadv-rest*, *′iusr-rest*, *′oadv-rest*, *′ousr-rest*, *′more*) *rest-scheme*
begin

fun *trace-rest-aux*
:: *′s-rest spmf* ⇒ (*′iadv-rest* × *′oadv-rest* × *′event list* + *′iusr-rest* × *′ousr-rest* × *′event list*) *list* ⇒ *′s-rest spmf* **where**
trace-rest-aux p [] = *p*
| *trace-rest-aux p* (*Inl* (*ia*, *oaes*) # *tr*) = *trace-rest-aux* (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *rfunc-adv rest s ia*)) *oaes*) *tr*
| *trace-rest-aux p* (*Inr* (*iu*, *oues*) # *tr*) = *trace-rest-aux* (*cond-spmf-fst* (*bind-spmf p* (λ*s*. *rfunc-usr rest s iu*)) *oues*) *tr*

end

lemma *trace-rest-conv-trace-rest-aux*:
trace-rest′ rest p tr =
(λ*ia*. *bind-spmf* (*trace-rest-aux rest p tr*) (λ*s*. *map-spmf fst* (*rfunc-adv rest s ia*)),
λ*iu*. *bind-spmf* (*trace-rest-aux rest p tr*) (λ*s*. *map-spmf fst* (*rfunc-usr rest s iu*)))
by(*induction p tr rule*: *trace-rest-aux.induct*) *simp-all*

lemma *trace-rest-aux-append*:
trace-rest-aux rest p (*tr* @ *tr′*) = *trace-rest-aux rest* (*trace-rest-aux rest p tr*) *tr′*
by(*induction p tr rule*: *trace-rest-aux.induct*) *auto*

inductive *trace-rest-closure*
:: (*′s-rest*, *′event*, *′iadv-rest*, *′iusr-rest*, *′oadv-rest*, *′ousr-rest*, *′more*) *rest-scheme*
⇒ (*′s-rest′*, *′event*, *′iadv-rest*, *′iusr-rest*, *′oadv-rest*, *′ousr-rest*, *′more′*) *rest-scheme*
⇒ *′iadv-rest set* ⇒ *′iusr-rest set*
⇒ *′s-rest spmf* ⇒ *′s-rest′ spmf* ⇒ *′s-rest spmf* ⇒ *′s-rest′ spmf* ⇒ *bool*
for *rest1 rest2 IA IU p q* **where**
trace-rest-closure rest1 rest2 IA IU p q (*trace-rest-aux rest1 p tr*) (*trace-rest-aux rest2 q tr*)
if *set tr* ⊆ *IA* × *UNIV* <+> *IU* × *UNIV*

lemma *trace-rest-closure-start*: *trace-rest-closure rest1 rest2 IA IU p q p q*
by(*simp add*: *trace-rest-closure.simps exI*[**where** *x*=[]])

lemma *trace-rest-closure-step*:
assumes *trace-rest-eq rest1 rest2 IA IU p q*
and *trace-rest-closure rest1 rest2 IA IU p q p′ q′*
shows *trace-rest-closure-step-rfunc-adv*:

ia ∈ *IA* ⟹ *bind-spmf p′* (*λs1. map-spmf fst* (*rfunc-adv rest1 s1 ia*)) = *bind-spmf q′* (*λs2. map-spmf fst* (*rfunc-adv rest2 s2 ia*))
(**is** *PROP ?thesis1*)
and *trace-rest-closure-step-rfunc-usr*:
iu ∈ *IU* ⟹ *bind-spmf p′* (*λs1. map-spmf fst* (*rfunc-usr rest1 s1 iu*)) = *bind-spmf q′* (*λs2. map-spmf fst* (*rfunc-usr rest2 s2 iu*))
(**is** *PROP ?thesis2*)
proof −
from *assms*(*2*) **obtain** *tr* **where** *p*: *p′* = *trace-rest-aux rest1 p tr*
and *q*: *q′* = *trace-rest-aux rest2 q tr*
and *tr*: *set tr* ⊆ *IA* × *UNIV* <+> *IU* × *UNIV* **by** *cases*
from *trace-rest-eqD*[*OF assms*(*1*) *tr*] *p q*
show *PROP ?thesis1* **and** *PROP ?thesis2*
by(*simp-all add*: *trace-rest-conv-trace-rest-aux*)
qed

lemma *trace-rest-closure-sim*:
fixes *rest1 rest2 IA IU p q*
defines *S* ≡ *trace-rest-closure rest1 rest2 IA IU p q*
assumes *S p′ q′*
shows *trace-rest-closure-sim-rfunc-adv*: *ia* ∈ *IA*
⟹ *S* (*cond-spmf-fst* (*bind-spmf p′* (*λs1. rfunc-adv rest1 s1 ia*)) *oa*)
(*cond-spmf-fst* (*bind-spmf q′* (*λs2. rfunc-adv rest2 s2 ia*)) *oa*)
(**is** *PROP ?thesis1*)
and *trace-rest-closure-sim-rfunc-usr*: *iu* ∈ *IU*
⟹ *S* (*cond-spmf-fst* (*bind-spmf p′* (*λs1. rfunc-usr rest1 s1 iu*)) *ou*)
(*cond-spmf-fst* (*bind-spmf q′* (*λs2. rfunc-usr rest2 s2 iu*)) *ou*)
(**is** *PROP ?thesis2*)
proof −
from *assms*(*2*) **obtain** *tr* **where** *p*: *p′* = *trace-rest-aux rest1 p tr*
and *q*: *q′* = *trace-rest-aux rest2 q tr*
and *tr*: *set tr* ⊆ *IA* × *UNIV* <+> *IU* × *UNIV* **unfolding** *S-def* **by** *cases*
show *PROP ?thesis1* **using** *p q tr*
by(*auto simp add*: *S-def trace-rest-closure.simps trace-rest-aux-append intro*!: *exI*[**where** *x*=*tr* @ [*Inl* (-, -)]])
show *PROP ?thesis2* **using** *p q tr*
by(*auto simp add*: *S-def trace-rest-closure.simps trace-rest-aux-append intro*!: *exI*[**where** *x*=*tr* @ [*Inr* (-, -)]])
qed

proposition *trace-rest-eq-complete*:
assumes *trace-rest-eq rest1 rest2 IA IU p q*
obtains *S*
where *S p q*
and ⋀*p q ia.* ⟦ *S p q*; *ia* ∈ *IA* ⟧ ⟹
bind-spmf p (*λs1. map-spmf fst* (*rfunc-adv rest1 s1 ia*)) = *bind-spmf q* (*λs2. map-spmf fst* (*rfunc-adv rest2 s2 ia*))
and ⋀*p q ia oa.* ⟦ *S p q*; *ia* ∈ *IA* ⟧
⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (*λs1. rfunc-adv rest1 s1 ia*)) *oa*) (*cond-spmf-fst*

(*bind-spmf q* (*λs2. rfunc-adv rest2 s2 ia*)) *oa*)
  and ⋀*p q iu*. ⟦ *S p q*; *iu* ∈ *IU* ⟧ ⟹
    bind-spmf p (*λs1. map-spmf fst* (*rfunc-usr rest1 s1 iu*)) = *bind-spmf q* (*λs2. map-spmf fst* (*rfunc-usr rest2 s2 iu*))
  and ⋀*p q iu ou*. ⟦ *S p q*; *iu* ∈ *IU* ⟧
  ⟹ *S* (*cond-spmf-fst* (*bind-spmf p* (*λs1. rfunc-usr rest1 s1 iu*)) *ou*) (*cond-spmf-fst* (*bind-spmf q* (*λs2. rfunc-usr rest2 s2 iu*)) *ou*)
proof –
 show *thesis*
  by(*rule that*[**where** *S*=*trace-rest-closure rest1 rest2 IA IU p q*])
  (*auto intro*: *trace-rest-closure-start trace-rest-closure-step*[*OF assms*] *trace-rest-closure-sim*)
qed

definition *callee-of-core*
 :: (*'s-core*, *'event*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
  ⇒ (*'s-core*, *'event* + *'iadv-core* + *'iusr-core*, *unit* + *'oadv-core* + *'ousr-core*) *oracle'* **where**
 callee-of-core core =
  map-fun id (*map-fun id* (*map-spmf* (*Pair* ()))) (*cpoke core*) $\oplus_O$ *cfunc-adv core* $\oplus_O$ *cfunc-usr core*

lemma *callee-of-core-simps* [*simp*]:
 callee-of-core core s (*Inl e*) = *map-spmf* (*Pair* (*Inl* ())) (*cpoke core s e*)
 callee-of-core core s (*Inr* (*Inl iadv-core*)) = *map-spmf* (*apfst* (*Inr* ∘ *Inl*)) (*cfunc-adv core s iadv-core*)
 callee-of-core core s (*Inr* (*Inr iusr-core*)) = *map-spmf* (*apfst* (*Inr* ∘ *Inr*)) (*cfunc-usr core s iusr-core*)
 by(*simp-all add*: *callee-of-core-def spmf.map-comp o-def apfst-def prod.map-comp id-def*)

lemma *WT-callee-of-core* [*WT-intro*]:
 assumes *WT*: *WT-core* $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I core*
  and *I*: *I s*
 shows $\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-adv* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr*) ⊢*c callee-of-core core s* √
 apply(*rule WT-calleeI*)
 subgoal for *x y s'* **using** *I WT-coreD*[*OF WT*]
  by(*auto simp add*: *callee-of-core-def plus-oracle-def split*!: *sum.splits*)
 done

lemma *WT-core-callee-invariant-on* [*WT-intro*]:
 assumes *WT*: *WT-core* $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I core*
 shows *callee-invariant-on* (*callee-of-core core*) *I* ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-adv* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr*))
 apply *unfold-locales*
 subgoal for *s x y s'* **by**(*auto simp add*: *callee-of-core-def plus-oracle-def split*!: *sum.splits dest*: *WT-coreD*[*OF assms*])
 subgoal by(*rule WT-callee-of-core*[*OF WT*])
 done

definition *callee-of-rest*
:: (*'s-rest*, *'event*, *'iadv-rest*, *'iusr-rest*, *'oadv-rest*, *'ousr-rest*, *'more*) *rest-scheme*
⇒ (*'s-rest*, *'iadv-rest* + *'iusr-rest*, *'oadv-rest* × *'event list* + *'ousr-rest* × *'event list*) *oracle'* **where**
callee-of-rest rest = *rfunc-adv rest* $\oplus_O$ *rfunc-usr rest*

lemma *callee-of-rest-simps* [*simp*]:
callee-of-rest rest s (*Inl iadv-rest*) = *map-spmf* (*apfst Inl*) (*rfunc-adv rest s iadv-rest*)
callee-of-rest rest s (*Inr iusr-rest*) = *map-spmf* (*apfst Inr*) (*rfunc-usr rest s iusr-rest*)
by(*simp-all add*: *callee-of-rest-def*)

lemma *WT-callee-of-rest* [*WT-intro*]:
assumes *WT*: *WT-rest* $\mathcal{I}$*-adv* $\mathcal{I}$*-usr I rest*
and *I*: *I s*
shows *e*$\mathcal{I}$ $\mathcal{I}$*-adv* $\oplus_{\mathcal{I}}$ *e*$\mathcal{I}$ $\mathcal{I}$*-usr* ⊢*c callee-of-rest rest s* √
apply(*rule WT-calleeI*)
subgoal for *x y s'* **using** *I WT-restD*[*OF WT*]
by(*auto simp add*: *callee-of-core-def plus-oracle-def split*!: *sum.splits*)
done

fun *fuse-callee*
:: (*'iadv-core* + *'iadv-rest*) + (*'iusr-core* + *'iusr-rest*) ⇒
((*'oadv-core* + *'oadv-rest*) + (*'ousr-core* + *'ousr-rest*),
(*'event* + *'iadv-core* + *'iusr-core*) + (*'iadv-rest* + *'iusr-rest*),
(*unit* + *'oadv-core* + *'ousr-core*) + (*'oadv-rest* × *'event list* + *'ousr-rest* × *'event list*)) *gpv*
where
fuse-callee (*Inl* (*Inl iadv-core*)) = *Pause* (*Inl* (*Inr* (*Inl iadv-core*))) (*λx. case x of*
Inl (*Inr* (*Inl oadv-core*)) ⇒ *Done* (*Inl* (*Inl oadv-core*))
| - ⇒ *Fail*)
| *fuse-callee* (*Inl* (*Inr iadv-rest*)) = *Pause* (*Inr* (*Inl iadv-rest*)) (*λx. case x of*
Inr (*Inl* (*oadv-rest*, *es*)) ⇒ *bind-gpv* (*pauses* (*map* (*Inl* ∘ *Inl*) *es*)) (*λ-. Done* (*Inl* (*Inr oadv-rest*)))
| - ⇒ *Fail*)
| *fuse-callee* (*Inr* (*Inl iusr-core*)) = *Pause* (*Inl* (*Inr* (*Inr iusr-core*))) (*λx. case x of*
Inl (*Inr* (*Inr oadv-core*)) ⇒ *Done* (*Inr* (*Inl oadv-core*)))
| *fuse-callee* (*Inr* (*Inr iusr-rest*)) = *Pause* (*Inr* (*Inr iusr-rest*)) (*λx. case x of*
Inr (*Inr* (*ousr-rest*, *es*)) ⇒ *bind-gpv* (*pauses* (*map* (*Inl* ∘ *Inl*) *es*)) (*λ-. Done* (*Inr* (*Inr ousr-rest*))))

case-of-simps *fuse-callee-case*: *fuse-callee.simps*

definition *fuse-converter*
:: ((*'iadv-core* + *'iadv-rest*) + (*'iusr-core* + *'iusr-rest*),
(*'oadv-core* + *'oadv-rest*) + (*'ousr-core* + *'ousr-rest*),

('event + 'iadv-core + 'iusr-core) + ('iadv-rest + 'iusr-rest),
(unit + 'oadv-core + 'ousr-core) + ('oadv-rest × 'event list + 'ousr-rest × 'event list)) *converter*
where
fuse-converter = *converter-of-callee* (*stateless-callee fuse-callee*) ()

lemma *fuse-converter*:
resource-of-oracle (*fused-resource.fuse core rest*) (*s-core*, *s-rest*) =
fuse-converter ⊳ (*resource-of-oracle* (*callee-of-core core*) *s-core* ∥ *resource-of-oracle* (*callee-of-rest rest*) *s-rest*)
unfolding *fuse-converter-def resource-of-parallel-oracle*[*symmetric*] *attach-CNV-RES attach-stateless-callee resource-of-oracle-extend-state-oracle*
proof(*rule arg-cong2*[**where** *f*=*resource-of-oracle*]; *clarsimp simp add*: *fun-eq-iff*)
interpret *fused-resource core core-init* **for** *core-init* **.**
have *foldl-spmf* (*map-fun id* (*map-fun* (*Inl* ∘ *Inl*) *id*) (*map-fun id* (*map-fun id* (*map-spmf snd*)) (*callee-of-core core* ‡$_O$ *callee-of-rest rest*))) (*return-spmf* (*s-core*, *s-rest*)) *xs*
= *map-spmf* (λ*s-core*. (*s-core*, *s-rest*)) (*foldl-spmf* (*cpoke core*) (*return-spmf s-core*) *xs*) **for** *s-core s-rest xs*
by(*induction xs arbitrary*: *s-core*)
(*simp-all add*: *spmf.map-comp foldl-spmf-Cons' map-bind-spmf bind-map-spmf o-def del*: *foldl-spmf-Cons*)
then show *fuse rest* (*s-core*, *s-rest*) *q* = *exec-gpv* (*callee-of-core core* ‡$_O$ *callee-of-rest rest*) (*fuse-callee q*) (*s-core*, *s-rest*)
for *s-core s-rest q*
by(*cases q rule*: *fuse-callee.cases*; *clarsimp simp add*: *map-bind-spmf bind-map-spmf exec-gpv-bind exec-gpv-pauses intro*!: *bind-spmf-cong*[*OF refl*]; *simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
qed

lemma *trace-eq-callee-of-coreI*:
trace-callee-eq (*callee-of-core core1*) (*callee-of-core core2*) (*E* <+> *IA* <+> *IU*) *p q*
if *trace-core-eq core1 core2 E IA IU p q*
proof −
from *that* **obtain** *S-core*
where *core-start*: *S-core p q*
and *step-cpoke*: ⋀*p q e*. *S-core p q* ⟹ *e* ∈ *E*
⟹ *weight-spmf* (*bind-spmf p* (λ*s*. *cpoke core1 s e*)) = *weight-spmf* (*bind-spmf q* (λ*s*. *cpoke core2 s e*))
and *sim-cpoke*: ⋀*p q e*. *S-core p q* ⟹ *e* ∈ *E*
⟹ *S-core* (*mk-lossless* (*bind-spmf p* (λ*s*. *cpoke core1 s e*))) (*mk-lossless* (*bind-spmf q* (λ*s*. *cpoke core2 s e*)))
and *step-cfunc-adv*: ⋀*p q ia*. ⟦ *S-core p q*; *ia* ∈ *IA* ⟧
⟹ *bind-spmf p* (λ*s1*. *map-spmf fst* (*cfunc-adv core1 s1 ia*)) = *bind-spmf q* (λ*s2*. *map-spmf fst* (*cfunc-adv core2 s2 ia*))
and *sim-cfunc-adv*: ⋀*p q ia oa*. ⟦ *S-core p q*; *ia* ∈ *IA* ⟧ ⟹
S-core (*cond-spmf-fst* (*bind-spmf p* (λ*s1*. *cfunc-adv core1 s1 ia*)) *oa*)
(*cond-spmf-fst* (*bind-spmf q* (λ*s2*. *cfunc-adv core2 s2 ia*)) *oa*)
and *step-cfunc-usr*: ⋀*p q iu*. ⟦ *S-core p q*; *iu* ∈ *IU* ⟧

$\Longrightarrow$ *bind-spmf p* (λ*s1. map-spmf fst* (*cfunc-usr core1 s1 iu*)) = *bind-spmf q* (λ*s2. map-spmf fst* (*cfunc-usr core2 s2 iu*))
and *sim-cfunc-usr*: $\bigwedge$*p q iu ou.* ⟦ *S-core p q*; *iu* ∈ *IU* ⟧ $\Longrightarrow$
S-core (*cond-spmf-fst* (*bind-spmf p* (λ*s1. cfunc-usr core1 s1 iu*)) *ou*)
(*cond-spmf-fst* (*bind-spmf q* (λ*s2. cfunc-usr core2 s2 iu*)) *ou*)
by(*rule trace-core-eq-complete*) *blast*

show *?thesis* **using** *core-start*
proof(*coinduct rule*: *trace′-eqI-sim*[*consumes 1*, *case-names step sim*])
case (*step p q a*)
then consider (*cpoke*) *e* **where** *a* = *Inl e e* ∈ *E*
| (*cfunc-adv*) *ia* **where** *a* = *Inr* (*Inl ia*) *ia* ∈ *IA*
| (*cfunc-usr*) *iu* **where** *a* = *Inr* (*Inr iu*) *iu* ∈ *IU* **by** *auto*
then show *?case*
proof *cases*
case *cpoke*
with *step-cpoke*[*OF step*(*1*), *of e*] **show** *?thesis*
by(*simp add*: *spmf.map-comp o-def map-spmf-const weight-bind-spmf*)
(*auto intro*!: *spmf-eqI simp add*: *spmf-bind spmf-scale-spmf max-def min-absorb2 weight-spmf-le-1*)
next
case *cfunc-adv*
with *step-cfunc-adv*[*OF step*(*1*) *cfunc-adv*(*2*)] **show** *?thesis*
by(*simp add*: *spmf.map-comp*)(*simp add*: *spmf.map-comp*[*symmetric*] *map-bind-spmf*[*unfolded o-def*, *symmetric*])
next
case *cfunc-usr*
with *step-cfunc-usr*[*OF step*(*1*) *cfunc-usr*(*2*)] **show** *?thesis*
by(*simp add*: *spmf.map-comp*)(*simp add*: *spmf.map-comp*[*symmetric*] *map-bind-spmf*[*unfolded o-def*, *symmetric*])
qed
next
case (*sim p q a res b s′*)
then consider (*cpoke*) *e* **where** *a* = *Inl e e* ∈ *E*
| (*cfunc-adv*) *ia* **where** *a* = *Inr* (*Inl ia*) *ia* ∈ *IA*
| (*cfunc-usr*) *iu* **where** *a* = *Inr* (*Inr iu*) *iu* ∈ *IU* **by** *auto*
then show *?case*
proof *cases*
case *cpoke*
with *sim-cpoke*[*OF sim*(*1*) , *of e*] *sim* **show** *?thesis*
by(*clarsimp simp add*: *map-bind-spmf*[*unfolded o-def*, *symmetric*])
next
case *cfunc-adv*
with *sim-cfunc-adv*[*OF sim*(*1*) *cfunc-adv*(*2*)] *sim* **show** *?thesis*
apply(*clarsimp simp add*: *map-bind-spmf*[*unfolded o-def*, *symmetric*] *apfst-def map-prod-def*)
apply(*subst* (*1 2*) *cond-spmf-fst-map-prod-inj*)
apply(*simp-all add*: *o-def*[*symmetric*] *inj-compose del*: *o-apply*)
done

```
    next
      case cfunc-usr
      with sim-cfunc-usr[OF sim(1) cfunc-usr(2)] sim show ?thesis
      apply(clarsimp simp add: map-bind-spmf[unfolded o-def, symmetric] apfst-def
map-prod-def)
        apply(subst (1 2) cond-spmf-fst-map-prod-inj)
         apply(simp-all add: o-def[symmetric] inj-compose del: o-apply)
        done
    qed
  qed
qed

lemma trace-eq-callee-of-restI:
  trace-callee-eq (callee-of-rest rest1) (callee-of-rest rest2) (IA <+> IU) p q
  if trace-rest-eq rest1 rest2 IA IU p q
proof −
  from that obtain S-rest
    where rest-start: S-rest p q
      and step-rfunc-adv: ⋀p q ia. ⟦ S-rest p q; ia ∈ IA ⟧
        ⟹ bind-spmf p (λs1. map-spmf fst (rfunc-adv rest1 s1 ia)) = bind-spmf q
(λs2. map-spmf fst (rfunc-adv rest2 s2 ia))
      and sim-rfunc-adv: ⋀p q ia oa. ⟦ S-rest p q; ia ∈ IA ⟧ ⟹
        S-rest (cond-spmf-fst (bind-spmf p (λs1. rfunc-adv rest1 s1 ia)) oa)
               (cond-spmf-fst (bind-spmf q (λs2. rfunc-adv rest2 s2 ia)) oa)
      and step-rfunc-usr: ⋀p q iu. ⟦ S-rest p q; iu ∈ IU ⟧
        ⟹ bind-spmf p (λs1. map-spmf fst (rfunc-usr rest1 s1 iu)) = bind-spmf q
(λs2. map-spmf fst (rfunc-usr rest2 s2 iu))
      and sim-rfunc-usr: ⋀p q iu ou. ⟦ S-rest p q; iu ∈ IU ⟧ ⟹
        S-rest (cond-spmf-fst (bind-spmf p (λs1. rfunc-usr rest1 s1 iu)) ou)
               (cond-spmf-fst (bind-spmf q (λs2. rfunc-usr rest2 s2 iu)) ou)
    by(rule trace-rest-eq-complete) blast

  show ?thesis using rest-start
  proof(coinduct rule: trace′-eqI-sim[consumes 1, case-names step sim])
    case (step p q a)
    then consider (rfunc-adv) ia where a = Inl ia ia ∈ IA
      | (rfunc-usr) iu where a = Inr iu iu ∈ IU by auto
    then show ?case
    proof cases
      case rfunc-adv
      with step-rfunc-adv[OF step(1) rfunc-adv(2)] show ?thesis
          by(simp add: spmf.map-comp)(simp add: spmf.map-comp[symmetric]
map-bind-spmf[unfolded o-def, symmetric])
    next
      case rfunc-usr
      with step-rfunc-usr[OF step(1) rfunc-usr(2)] show ?thesis
          by(simp add: spmf.map-comp)(simp add: spmf.map-comp[symmetric]
map-bind-spmf[unfolded o-def, symmetric])
    qed
```

next
case (*sim p q a res b s*′)
then consider (*rfunc-adv*) *ia* **where** *a* = *Inl ia ia* ∈ *IA*
| (*rfunc-usr*) *iu* **where** *a* = *Inr iu iu* ∈ *IU* **by** *auto*
then show *?case*
proof *cases*
case *rfunc-adv*
with *sim-rfunc-adv*[*OF sim*(*1*) *rfunc-adv*(*2*)] *sim* **show** *?thesis*
by(*clarsimp simp add*: *map-bind-spmf*[*unfolded o-def*, *symmetric*] *apfst-def map-prod-def*)
(*subst* (*1 2*) *cond-spmf-fst-map-prod-inj*; *simp*)
next
case *rfunc-usr*
with *sim-rfunc-usr*[*OF sim*(*1*) *rfunc-usr*(*2*)] *sim* **show** *?thesis*
by(*clarsimp simp add*: *map-bind-spmf*[*unfolded o-def*, *symmetric*] *apfst-def map-prod-def*)
(*subst* (*1 2*) *cond-spmf-fst-map-prod-inj*; *simp*)
qed
qed
qed

lemma *trace-callee-resource-of-oracle*:
trace-callee run-resource (*map-spmf* (*resource-of-oracle callee*) *p*) = *trace-callee callee p*
(**is** *?lhs* = *?rhs*)
proof(*intro ext*)
show *?lhs tr x* = *?rhs tr x* **for** *tr x*
proof(*induction tr arbitrary*: *p*)
case *Nil* **show** *?case* **by**(*simp add*: *bind-map-spmf o-def spmf.map-comp*)
next
case (*Cons a tr*)
obtain *y z* **where** *a* [*simp*]: *a* = (*y*, *z*) **by**(*cases a*)
have *trace-callee run-resource* (*map-spmf* (*RES callee*) *p*) (*a* # *tr*) *x* =
trace-callee run-resource (*cond-spmf-fst* (*map-spmf* (λ(*x*, *y*). (*x*, *RES callee y*)) (*p* ⤜ (λ*x*. (*callee x y*)))) *z*) *tr x*
by(*clarsimp simp add*: *bind-map-spmf o-def map-prod-def map-bind-spmf*)
also have . . . = *trace-callee run-resource* (*map-spmf* (*RES callee*) (*cond-spmf-fst* (*p* ⤜ (λ*x*. (*callee x y*))) *z*)) *tr x*
by(*subst cond-spmf-fst-map-prod-inj*) *simp-all*
finally show *?case* **using** *Cons.IH* **by** *simp*
qed
qed

lemma *trace-callee-resource-of-oracle*′:
trace-callee run-resource (*return-spmf* (*resource-of-oracle callee s*)) = *trace-callee callee* (*return-spmf s*)
using *trace-callee-resource-of-oracle*[**where** *p*=*return-spmf s*]
by *simp*

lemma *trace-eq-resource-of-oracle*:
trace-eq A (*map-spmf* (*resource-of-oracle callee1*) *p*) (*map-spmf* (*resource-of-oracle callee2*) *q*) =
trace-callee-eq callee1 callee2 A p q
unfolding *trace-callee-eq-def trace-callee-resource-of-oracle* **by**(*rule refl*)

lemma *WT-fuse-converter* [*WT-intro*]:
$(\mathcal{I}AC \oplus_{\mathcal{I}} map\text{-}\mathcal{I}\ id\ fst\ \mathcal{I}AR) \oplus_{\mathcal{I}} (\mathcal{I}UC \oplus_{\mathcal{I}} map\text{-}\mathcal{I}\ id\ fst\ \mathcal{I}UR), (\mathcal{I}E \oplus_{\mathcal{I}} (\mathcal{I}AC \oplus_{\mathcal{I}} \mathcal{I}UC)) \oplus_{\mathcal{I}} (\mathcal{I}AR \oplus_{\mathcal{I}} \mathcal{I}UR) \vdash_C fuse\text{-}converter\ \surd$
if $\forall x.\ \forall (y,\ es) \in responses\text{-}\mathcal{I}\ \mathcal{I}AR\ x.\ set\ es \subseteq outs\text{-}\mathcal{I}\ \mathcal{I}E\ \forall x.\ \forall (y,\ es) \in responses\text{-}\mathcal{I}\ \mathcal{I}UR\ x.\ set\ es \subseteq outs\text{-}\mathcal{I}\ \mathcal{I}E$
unfolding *fuse-converter-def* **using** *that*
by(*intro WT-converter-of-callee*)
(*fastforce simp add*: *stateless-callee-def image-image intro*: *rev-image-eqI intro*!: *WT-gpv-pauses split*: *if-split-asm*)+

theorem *fuse-trace-eq*:
fixes *core1* :: (*'s-core*, *'event*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
and *core2* :: (*'s-core'*, *'event*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
and *rest1* :: (*'s-rest*, *'event*, *'iadv-rest*, *'iusr-rest*, *'oadv-rest*, *'ousr-rest*, *'more1*) *rest-scheme*
and *rest2* :: (*'s-rest'*, *'event*, *'iadv-rest*, *'iusr-rest*, *'oadv-rest*, *'ousr-rest*, *'more2*) *rest-scheme*
assumes *core*: *trace-core-eq core1 core2* $(outs\text{-}\mathcal{I}\ \mathcal{I}E)$ $(outs\text{-}\mathcal{I}\ \mathcal{I}CA)$ $(outs\text{-}\mathcal{I}\ \mathcal{I}CU)$ (*return-spmf s-core*) (*return-spmf s-core'*)
and *rest*: *trace-rest-eq rest1 rest2* $(outs\text{-}\mathcal{I}\ \mathcal{I}RA)$ $(outs\text{-}\mathcal{I}\ \mathcal{I}RU)$ (*return-spmf s-rest*) (*return-spmf s-rest'*)
and *IC1*: *callee-invariant-on* (*callee-of-core core1*) *IC1* $(\mathcal{I}E \oplus_{\mathcal{I}} (\mathcal{I}CA \oplus_{\mathcal{I}} \mathcal{I}CU))$ *IC1 s-core*
and *IC2*: *callee-invariant-on* (*callee-of-core core2*) *IC2* $(\mathcal{I}E \oplus_{\mathcal{I}} (\mathcal{I}CA \oplus_{\mathcal{I}} \mathcal{I}CU))$ *IC2 s-core'*
and *IR1*: *callee-invariant-on* (*callee-of-rest rest1*) *IR1* $(\mathcal{I}RA \oplus_{\mathcal{I}} \mathcal{I}RU)$ *IR1 s-rest*
and *IR2*: *callee-invariant-on* (*callee-of-rest rest2*) *IR2* $(\mathcal{I}RA \oplus_{\mathcal{I}} \mathcal{I}RU)$ *IR2 s-rest'*
and *E1* [*WT-intro*]: $\forall x.\ \forall (y,\ es) \in responses\text{-}\mathcal{I}\ \mathcal{I}RA\ x.\ set\ es \subseteq outs\text{-}\mathcal{I}\ \mathcal{I}E$
and *E2* [*WT-intro*]: $\forall x.\ \forall (y,\ es) \in responses\text{-}\mathcal{I}\ \mathcal{I}RU\ x.\ set\ es \subseteq outs\text{-}\mathcal{I}\ \mathcal{I}E$
shows *trace-callee-eq* (*fused-resource.fuse core1 rest1*) (*fused-resource.fuse core2 rest2*)
$((outs\text{-}\mathcal{I}\ \mathcal{I}CA <+> outs\text{-}\mathcal{I}\ \mathcal{I}RA) <+> (outs\text{-}\mathcal{I}\ \mathcal{I}CU <+> outs\text{-}\mathcal{I}\ \mathcal{I}RU))$ (*return-spmf* (*s-core*, *s-rest*)) (*return-spmf* (*s-core'*, *s-rest'*))
proof –
let $?\mathcal{I}C = \mathcal{I}E \oplus_{\mathcal{I}} (\mathcal{I}CA \oplus_{\mathcal{I}} \mathcal{I}CU)$
let $?\mathcal{I}R = \mathcal{I}RA \oplus_{\mathcal{I}} \mathcal{I}RU$
let $?\mathcal{I}' = ?\mathcal{I}C \oplus_{\mathcal{I}} ?\mathcal{I}R$
let $?\mathcal{I} = (\mathcal{I}CA \oplus_{\mathcal{I}} map\text{-}\mathcal{I}\ id\ fst\ \mathcal{I}RA) \oplus_{\mathcal{I}} (\mathcal{I}CU \oplus_{\mathcal{I}} map\text{-}\mathcal{I}\ id\ fst\ \mathcal{I}RU)$

interpret *fuse1*: *fused-resource core1 s1* **for** *s1* **.**
interpret *fuse2*: *fused-resource core2 s2* **for** *s2* **.**

interpret *IC1*: *callee-invariant-on callee-of-core core1 IC1 ?$\mathcal{I}$C* **by** *fact*
interpret *IC2*: *callee-invariant-on callee-of-core core2 IC2 ?$\mathcal{I}$C* **by** *fact*
interpret *IR1*: *callee-invariant-on callee-of-rest rest1 IR1 ?$\mathcal{I}$R* **by** *fact*
interpret *IR2*: *callee-invariant-on callee-of-rest rest2 IR2 ?$\mathcal{I}$R* **by** *fact*

from *core* **have** *outs-$\mathcal{I}$?$\mathcal{I}$C* $\vdash_C$ *callee-of-core core1*(*s-core*) $\approx$ *callee-of-core core2*(*s-core'*)
by(*simp add*: *trace-eq-callee-of-coreI*)
hence *outs-$\mathcal{I}$?$\mathcal{I}$C* $\vdash_R$ *RES* (*callee-of-core core1*) *s-core* $\approx$ *RES* (*callee-of-core core2*) *s-core'* **by** *simp*
moreover have *outs-$\mathcal{I}$?$\mathcal{I}$R* $\vdash_C$ *callee-of-rest rest1*(*s-rest*) $\approx$ *callee-of-rest rest2*(*s-rest'*) **using** *rest*
by(*simp add*: *trace-eq-callee-of-restI*)
hence *outs-$\mathcal{I}$?$\mathcal{I}$R* $\vdash_R$ *RES* (*callee-of-rest rest1*) *s-rest* $\approx$ *RES* (*callee-of-rest rest2*) *s-rest'* **by** *simp*
ultimately have *outs-$\mathcal{I}$?$\mathcal{I}$'* $\vdash_R$
RES (*callee-of-core core1*) *s-core* ∥ *RES* (*callee-of-rest rest1*) *s-rest* $\approx$
RES (*callee-of-core core2*) *s-core'* ∥ *RES* (*callee-of-rest rest2*) *s-rest'*
by(*simp add*: *trace-eq'-parallel-resource*)
hence *outs-$\mathcal{I}$?$\mathcal{I}$* $\vdash_R$ *fuse-converter* ⊳ (*RES* (*callee-of-core core1*) *s-core* ∥ *RES* (*callee-of-rest rest1*) *s-rest*) $\approx$
fuse-converter ⊳ (*RES* (*callee-of-core core2*) *s-core'* ∥ *RES* (*callee-of-rest rest2*) *s-rest'*)
by(*rule attach-trace-eq'*)(*intro WT-intro IC1.WT-resource-of-oracle IC1 IC2.WT-resource-of-oracle IC2 IR1.WT-resource-of-oracle IR1 IR2.WT-resource-of-oracle IR2*)+
hence *trace-eq'* (*outs-$\mathcal{I}$?$\mathcal{I}$*) (*resource-of-oracle* (*fuse1.fuse rest1*) (*s-core*, *s-rest*)) (*resource-of-oracle* (*fuse2.fuse rest2*) (*s-core'*, *s-rest'*))
unfolding *fuse-converter* **by** *simp*
then show *?thesis* **by** *simp*
qed

inductive *trace-eq-simcl* :: (*'s1 spmf* ⇒ *'s2 spmf* ⇒ *bool*) ⇒ *'s1 spmf* ⇒ *'s2 spmf* ⇒ *bool*
for *S* **where**
base: *trace-eq-simcl S p q* **if** *S p q* **for** *p q*
| *bind-nat*: *trace-eq-simcl S* (*bind-spmf p f*) (*bind-spmf p g*)
if $\bigwedge$*x* :: *nat*. *x* ∈ *set-spmf p* ⟹ *S* (*f x*) (*g x*)

lemma *trace-eq-simcl-bindI* [*intro?*]: *trace-eq-simcl S* (*bind-spmf p f*) (*bind-spmf p g*)
if $\bigwedge$*x*. *x* ∈ *set-spmf p* ⟹ *S* (*f x*) (*g x*)
by(*subst* (*1 2*) *bind-spmf-to-nat-on*[*symmetric*])(*auto intro*!: *trace-eq-simcl.bind-nat simp add*: *that*)

lemma *trace-eq-simcl-bind*: *trace-eq-simcl S* (*bind-spmf p f*) (*bind-spmf p g*)
if ∗: $\bigwedge$*x* :: *'a*. *x* ∈ *set-spmf p* ⟹ *trace-eq-simcl S* (*f x*) (*g x*)
proof −
obtain *P* :: *'a* ⇒ *nat spmf* **and** *F G* **where**

**: ⋀x. x ∈ *set-spmf p* ⟹ *f x* = *bind-spmf* (*P x*) (*F x*) ∧ *g x* = *bind-spmf* (*P x*) (*G x*) ∧ (∀ *y*∈*set-spmf* (*P x*). *S* (*F x y*) (*G x y*))
apply(*atomize-elim*)
apply(*subst choice-iff*[*symmetric*])+
apply(*fastforce dest*!: * *elim*!: *trace-eq-simcl.cases intro*: *exI*[**where** *x*=*return-spmf* -])
done
have *bind-spmf p f* = *bind-spmf* (*bind-spmf p* (λ*x*. *map-spmf* (*Pair x*) (*P x*))) (λ(*x*, *y*). *F x y*)
by(*simp add*: *bind-map-spmf o-def* ** *cong*: *bind-spmf-cong*)
moreover have *bind-spmf p g* = *bind-spmf* (*bind-spmf p* (λ*x*. *map-spmf* (*Pair x*) (*P x*))) (λ(*x*, *y*). *G x y*)
by(*simp add*: *bind-map-spmf o-def* ** *cong*: *bind-spmf-cong*)
ultimately show *?thesis* **by**(*simp only*:)(*rule trace-eq-simcl-bindI*; *clarsimp simp add*: **)
qed

lemma *trace-eq-simcl-bind1-scale*: *trace-eq-simcl S* (*bind-spmf p f*) (*scale-spmf* (*weight-spmf p*) *q*)
if ∀ *x*∈*set-spmf p*. *trace-eq-simcl S* (*f x*) *q*
proof −
have *trace-eq-simcl S* (*bind-spmf p f*) (*bind-spmf p* (λ-. *q*))
by(*rule trace-eq-simcl-bind*)(*simp add*: *that*)
thus *?thesis* **by**(*simp add*: *bind-spmf-const*)
qed

lemma *trace-eq-simcl-bind1*: *trace-eq-simcl S* (*bind-spmf p f*) *q*
if ∀ *x*∈*set-spmf p*. *trace-eq-simcl S* (*f x*) *q lossless-spmf p*
using *trace-eq-simcl-bind1-scale*[*OF that*(*1*)] *that*(*2*) **by**(*simp add*: *lossless-weight-spmfD*)

lemma *trace-eq-simcl-bind2-scale*: *trace-eq-simcl S* (*scale-spmf* (*weight-spmf q*) *p*) (*bind-spmf q f*)
if ∀ *x*∈*set-spmf q*. *trace-eq-simcl S p* (*f x*)
proof −
have *trace-eq-simcl S* (*bind-spmf q* (λ-. *p*)) (*bind-spmf q f*)
by(*rule trace-eq-simcl-bind*)(*simp add*: *that*)
thus *?thesis* **by**(*simp add*: *bind-spmf-const*)
qed

lemma *trace-eq-simcl-bind2*: *trace-eq-simcl S p* (*bind-spmf q f*)
if ∀ *x*∈*set-spmf q*. *trace-eq-simcl S p* (*f x*) *lossless-spmf q*
using *trace-eq-simcl-bind2-scale*[*OF that*(*1*)] *that*(*2*) **by**(*simp add*: *lossless-weight-spmfD*)

lemma *trace-eq-simcl-return-pmf-None* [*simp*, *intro*!]: *trace-eq-simcl S* (*return-pmf None*) (*return-pmf None*)
for *S* :: ′*s1 spmf* ⇒ ′*s2 spmf* ⇒ *bool*
proof −
have *trace-eq-simcl S* (*bind-spmf* (*return-pmf None*) (*undefined* :: *nat* ⇒ ′*s1 spmf*)) (*bind-spmf* (*return-pmf None*) (*undefined* :: *nat* ⇒ ′*s2 spmf*))

by(*rule trace-eq-simcl-bindI*) *simp*
then show *?thesis* **by** *simp*
qed

lemma *trace-eq-simcl-map*: *trace-eq-simcl S* (*map-spmf f p*) (*map-spmf g p*)
if ∀ *x*∈*set-spmf p*. *S* (*return-spmf* (*f x*)) (*return-spmf* (*g x*))
unfolding *map-spmf-conv-bind-spmf*
by(*rule trace-eq-simcl-bindI*)(*simp add*: *that*)

lemma *trace-eq-simcl-map1*: *trace-eq-simcl S* (*map-spmf f p*) *q*
if ∀ *x*∈*set-spmf p*. *trace-eq-simcl S* (*return-spmf* (*f x*)) *q lossless-spmf p*
unfolding *map-spmf-conv-bind-spmf*
by(*rule trace-eq-simcl-bind1*)(*simp-all add*: *that*)

lemma *trace-eq-simcl-map2*: *trace-eq-simcl S p* (*map-spmf f q*)
if ∀ *x*∈*set-spmf q*. *trace-eq-simcl S p* (*return-spmf* (*f x*)) *lossless-spmf q*
unfolding *map-spmf-conv-bind-spmf*
by(*rule trace-eq-simcl-bind2*)(*simp-all add*: *that*)

lemma *trace-eq-simcl-return-spmf* [*simp*]: *trace-eq-simcl S* (*return-spmf x*) (*return-spmf y*) ⟷ *S* (*return-spmf x*) (*return-spmf y*)
apply(*rule iffI*)
subgoal by(*erule trace-eq-simcl.cases*; *clarsimp dest*!: *sym*[**where** *s*=*return-spmf* -])(*auto 4 4 simp add*: *bind-eq-return-spmf dest*!: *lossless-spmfD-set-spmf-nonempty*)
by(*simp add*: *trace-eq-simcl.base*)

lemma *trace-eq-simcl-callee*:
fixes *callee1* :: ('*a*, '*b*, '*s1*) *callee* **and** *callee2* :: ('*a*, '*b*, '*s2*) *callee*
assumes *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*) ⟧
⟹ *trace-eq-simcl 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*)
and *start*: *trace-eq-simcl S p q* **and** *a*: *a* ∈ *A*
shows *trace-eq-simcl-callee-step*: *bind-spmf p* (λ*s*. *map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (λ*s*. *map-spmf fst* (*callee2 s a*)) (**is** *?step*)
and *trace-eq-simcl-callee-sim*: ⋀*res b s*'. ⟦ *res* ∈ *set-spmf q*; (*b*, *s*') ∈ *set-spmf* (*callee2 res a*) ⟧
⟹ *trace-eq-simcl 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*) (**is** ⋀*res b s*'. ⟦ *?res res*; *?b res b s*' ⟧ ⟹ *?sim res b s*')
proof −
show *eq*: *?step* **using** *start a* **by** *cases*(*auto intro*!: *bind-spmf-cong intro*: *step*)
show *?sim res b s*' **if** *?res res ?b res b s*' **for** *res b s*' **using** *start*
proof *cases*
case *base* **then show** *?thesis* **using** *a that* **by**(*rule sim*)
next

case (*bind-nat X f g*)
let *?Y* = *cond-bind-spmf-fst X* (*λy. map-spmf fst* (*bind-spmf* (*f y*) (*λs. callee1 s a*))) *b*
let *?Y′* = *cond-bind-spmf-fst X* (*λy. map-spmf fst* (*bind-spmf* (*g y*) (*λs. callee2 s a*))) *b*
have *cond-spmf-fst* (*bind-spmf p* (*λs. callee1 s a*)) *b* = *bind-spmf ?Y* (*λx. cond-spmf-fst* (*bind-spmf* (*f x*) (*λs. callee1 s a*)) *b*)
unfolding *bind-nat* **by**(*simp add*: *cond-spmf-fst-bind o-def*)
moreover have *cond-spmf-fst* (*bind-spmf q* (*λs. callee2 s a*)) *b* = *bind-spmf ?Y′* (*λx. cond-spmf-fst* (*bind-spmf* (*g x*) (*λs. callee2 s a*)) *b*)
unfolding *bind-nat* **by**(*simp add*: *cond-spmf-fst-bind o-def*)
moreover have *?Y* = *?Y′* **using** *bind-nat eq*
by(*intro spmf-eqI*)(*fastforce simp add*: *map-bind-spmf o-def spmf-eq-0-set-spmf dest*: *step*[*OF - a*])
ultimately
show *trace-eq-simcl S* (*cond-spmf-fst* (*bind-spmf p* (*λs. callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q* (*λs. callee2 s a*)) *b*) **using** *bind-nat a*
by(*simp*)(*rule trace-eq-simcl-bind*; *auto intro*!: *sim simp add*: *bind-UNION*)
qed
qed

proposition *trace′-eqI-sim-upto*:
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 (*λs. map-spmf fst* (*callee1 s a*)) = *bind-spmf q* (*λs. 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*) ⟧
⟹ *trace-eq-simcl S* (*cond-spmf-fst* (*bind-spmf p* (*λs. callee1 s a*)) *b*)
(*cond-spmf-fst* (*bind-spmf q* (*λs. callee2 s a*)) *b*)
shows *trace-callee-eq callee1 callee2 A p q*
proof −
let *?S* = *trace-eq-simcl S*
from *start* **have** *?S p q* **by**(*rule trace-eq-simcl.base*)
then show *?thesis* **by**(*rule trace′-eqI-sim*)(*rule trace-eq-simcl-callee*[*OF step sim*]; *assumption*)+
qed

lemma *trace-core-eq-simI-upto*:
fixes *core1* :: (*′s-core*, *′event*, *′iadv-core*, *′iusr-core*, *′oadv-core*, *′ousr-core*) *core*
and *core2* :: (*′s-core′*, *′event*, *′iadv-core*, *′iusr-core*, *′oadv-core*, *′ousr-core*) *core*
and *S* :: *′s-core spmf* ⇒ *′s-core′ spmf* ⇒ *bool*
assumes *start*: *S p q*
and *step-cpoke*: ⋀*p q e*. ⟦ *S p q*; *e* ∈ *E* ⟧ ⟹
weight-spmf (*bind-spmf p* (*λs. cpoke core1 s e*)) = *weight-spmf* (*bind-spmf q* (*λs. cpoke core2 s e*))
and *sim-cpoke*: ⋀*p q e*. ⟦ *S p q*; *e* ∈ *E* ⟧ ⟹
trace-eq-simcl S (*mk-lossless* (*bind-spmf p* (*λs. cpoke core1 s e*))) (*mk-lossless*

(*bind-spmf q* (*λs. cpoke core2 s e*)))
  and *step-cfunc-adv*: $\bigwedge$*p q ia.* ⟦ *S p q*; *ia* ∈ *IA* ⟧ ⟹
    bind-spmf p (*λs1. map-spmf fst* (*cfunc-adv core1 s1 ia*)) = *bind-spmf q* (*λs2. map-spmf fst* (*cfunc-adv core2 s2 ia*))
  and *sim-cfunc-adv*: $\bigwedge$*p q ia s1 s2 s1′ s2′ oa.* ⟦ *S p q*; *ia* ∈ *IA*;
    s1 ∈ *set-spmf p*; *s2* ∈ *set-spmf q*; (*oa*, *s1′*) ∈ *set-spmf* (*cfunc-adv core1 s1 ia*); (*oa*, *s2′*) ∈ *set-spmf* (*cfunc-adv core2 s2 ia*) ⟧
    ⟹ *trace-eq-simcl S* (*cond-spmf-fst* (*bind-spmf p* (*λs1. cfunc-adv core1 s1 ia*)) *oa*) (*cond-spmf-fst* (*bind-spmf q* (*λs2. cfunc-adv core2 s2 ia*)) *oa*)
  and *step-cfunc-usr*: $\bigwedge$*p q iu.* ⟦ *S p q*; *iu* ∈ *IU* ⟧ ⟹
    bind-spmf p (*λs1. map-spmf fst* (*cfunc-usr core1 s1 iu*)) = *bind-spmf q* (*λs2. map-spmf fst* (*cfunc-usr core2 s2 iu*))
  and *sim-cfunc-usr*: $\bigwedge$*p q iu s1 s2 s1′ s2′ ou.* ⟦ *S p q*; *iu* ∈ *IU*;
    s1 ∈ *set-spmf p*; *s2* ∈ *set-spmf q*; (*ou*, *s1′*) ∈ *set-spmf* (*cfunc-usr core1 s1 iu*); (*ou*, *s2′*) ∈ *set-spmf* (*cfunc-usr core2 s2 iu*) ⟧
    ⟹ *trace-eq-simcl S* (*cond-spmf-fst* (*bind-spmf p* (*λs1. cfunc-usr core1 s1 iu*)) *ou*) (*cond-spmf-fst* (*bind-spmf q* (*λs2. cfunc-usr core2 s2 iu*)) *ou*)
 shows *trace-core-eq core1 core2 E IA IU p q*
proof −
 let *?S* = *trace-eq-simcl S*
 from *start* **have** *?S p q* **by**(*rule trace-eq-simcl.base*)
 then show *?thesis*
  proof(*rule trace-core-eq-simI*, *goal-cases Step-cpoke Sim-cpoke Step-cfunc-adv Sim-cfunc-adv Step-cfunc-usr Sim-cfunc-usr*)
  { **case** (*Step-cpoke p q e*)
   then show *?case* **using** *step-cpoke*
   by *cases*(*auto simp add*: *weight-bind-spmf o-def intro*!: *Bochner-Integration.integral-cong-AE*)
}
  note *eq* = *this*

  case (*Sim-cpoke p q e*) **then show** *?case*
  proof *cases*
   case *base* **then show** *?thesis* **using** *Sim-cpoke*(*2*) **by**(*rule sim-cpoke*)
  next
   case (*bind-nat X f g*)
    then have *cond-bind-spmf X* (*λy. f y* ≫= (*λs. cpoke core1 s e*)) *UNIV* = *cond-bind-spmf X* (*λy. g y* ≫= (*λs. cpoke core2 s e*)) *UNIV*
     using *eq*[*OF Sim-cpoke*] *step-cpoke Sim-cpoke*
   by(*intro spmf-eqI*)(*simp add*: *weight-spmf-def measure-spmf-zero-iff bind-UNION spmf-eq-0-set-spmf*)
   then show *?thesis* **using** *bind-nat Sim-cpoke sim-cpoke*
     by(*auto simp add*: *cond-bind-spmf cond-spmf-UNIV*[*symmetric*] *simp del*: *cond-spmf-UNIV intro*: *trace-eq-simcl-bind*)
  qed
 next
  { **case** (*Step-cfunc-adv p q ia*)
   then show *?case* **using** *step-cfunc-adv* **by** *cases*(*auto intro*!: *bind-spmf-cong*)
}
  note *eq* = *this*

case (*Sim-cfunc-adv p q ia s1 s2 s1′ s2′ oa*) **then show** *?case*
proof *cases*
case *base* **then show** *?thesis* **using** *Sim-cfunc-adv*(*2−*) **by**(*rule sim-cfunc-adv*)
next
case (*bind-nat X f g*)
then have *cond-bind-spmf-fst X* (*λy. map-spmf fst* (*f y* ≫= (*λs1. cfunc-adv core1 s1 ia*))) *oa* =
cond-bind-spmf-fst X (*λy. map-spmf fst* (*g y* ≫= (*λs2. cfunc-adv core2 s2 ia*))) *oa*
using *eq*[*OF Sim-cfunc-adv*(*1*,*2*)]
by(*intro spmf-eqI*)(*fastforce simp add*: *map-bind-spmf o-def spmf-eq-0-set-spmf dest*: *step-cfunc-adv*[*OF - Sim-cfunc-adv*(*2*)])
then show *?thesis* **using** *bind-nat*(*3−*) *Sim-cfunc-adv*(*1−2*)
unfolding *bind-nat*(*1*,*2*) *bind-spmf-assoc*
apply(*subst* (*1 2*) *cond-spmf-fst-bind*)
apply(*simp add*: *o-def*)
apply(*rule trace-eq-simcl-bind*)
apply *clarsimp*
apply(*frule step-cfunc-adv*[*OF bind-nat*(*3*) *Sim-cfunc-adv*(*2*), *THEN arg-cong*[**where** *f=set-spmf*], *THEN equalityD2*])
apply(*clarsimp simp add*: *o-def bind-UNION*)
apply(*drule subsetD*)
apply *fastforce*
apply(*auto intro*: *sim-cfunc-adv*)
done
qed
next
{ case (*Step-cfunc-usr p q iu*)
then show *?case* **using** *step-cfunc-usr* **by** *cases*(*auto intro*!: *bind-spmf-cong*)
}
note *eq* = *this*

case (*Sim-cfunc-usr p q iu s1 s2 s1′ s2′ ou*) **then show** *?case*
proof *cases*
case *base* **then show** *?thesis* **using** *Sim-cfunc-usr*(*2−*) **by**(*rule sim-cfunc-usr*)
next
case (*bind-nat X f g*)
then have *cond-bind-spmf-fst X* (*λy. map-spmf fst* (*f y* ≫= (*λs1. cfunc-usr core1 s1 iu*))) *ou* =
cond-bind-spmf-fst X (*λy. map-spmf fst* (*g y* ≫= (*λs2. cfunc-usr core2 s2 iu*))) *ou*
using *eq*[*OF Sim-cfunc-usr*(*1*,*2*)]
by(*intro spmf-eqI*)(*fastforce simp add*: *map-bind-spmf o-def spmf-eq-0-set-spmf dest*: *step-cfunc-usr*[*OF - Sim-cfunc-usr*(*2*)])
then show *?thesis* **using** *bind-nat*(*3−*) *Sim-cfunc-usr*(*1−2*)
unfolding *bind-nat*(*1*,*2*) *bind-spmf-assoc*
apply(*subst* (*1 2*) *cond-spmf-fst-bind*)
apply(*simp add*: *o-def*)

apply(*rule trace-eq-simcl-bind*)
apply *clarsimp*
apply(*frule step-cfunc-usr*[*OF bind-nat*(*3*) *Sim-cfunc-usr*(*2*), *THEN arg-cong*[**where** *f=set-spmf*], *THEN equalityD2*])
apply(*clarsimp simp add*: *o-def bind-UNION*)
apply(*drule subsetD*)
apply *fastforce*
apply(*auto intro*: *sim-cfunc-usr*)
done
qed
qed
qed

context
fixes *core* :: (*'s-core*, *'event1* + *'event2*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
and *rest* :: (*'s-rest*, *'event2*, *'iadv-rest*, *'iusr-rest*, *'oadv-rest*, *'ousr-rest*, *'more*) *rest-scheme*
begin

primcorec *core-with-rest* ::
(*'s-core* × *'s-rest*, *'event1*, *'iadv-core* + *'iadv-rest*, *'iusr-core* + *'iusr-rest*, *'oadv-core* + *'oadv-rest*, *'ousr-core* + *'ousr-rest*) *core*
where
cpoke core-with-rest = (λ(*s-core*, *s-rest*) *e*. *map-spmf* (λ*s-core'*. (*s-core'*, *s-rest*)) (*cpoke core s-core* (*Inl e*)))
| *cfunc-adv core-with-rest* = (λ(*s-core*, *s-rest*) *iadv*. *case iadv of*
Inl iadv-core ⇒ *map-spmf* (λ(*oadv-core*, *s-core'*). (*Inl oadv-core*, (*s-core'*, *s-rest*))) (*cfunc-adv core s-core iadv-core*)
| *Inr iadv-rest* ⇒
bind-spmf (*rfunc-adv rest s-rest iadv-rest*) (λ((*oadv-rest*, *es*), *s-rest'*).
map-spmf (λ*s-core'*. (*Inr oadv-rest*, (*s-core'*, *s-rest'*))) (*foldl-spmf* (*cpoke core*) (*return-spmf s-core*) (*map Inr es*))))
| *cfunc-usr core-with-rest* = (λ(*s-core*, *s-rest*) *iusr*. *case iusr of*
Inl iusr-core ⇒ *map-spmf* (λ(*ousr-core*, *s-core'*). (*Inl ousr-core*, (*s-core'*, *s-rest*))) (*cfunc-usr core s-core iusr-core*)
| *Inr iusr-rest* ⇒
bind-spmf (*rfunc-usr rest s-rest iusr-rest*) (λ((*ousr-rest*, *es*), *s-rest'*).
map-spmf (λ*s-core'*. (*Inr ousr-rest*, (*s-core'*, *s-rest'*))) (*foldl-spmf* (*cpoke core*) (*return-spmf s-core*) (*map Inr es*))))

end

lemma *fuse-core-with-rest*:
fixes *core* :: (*'s-core*, *'event1* + *'event2*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
and *rest1* :: (*'s-rest1*, *'event1*, *'iadv-rest1*, *'iusr-rest1*, *'oadv-rest1*, *'ousr-rest1*,

'more1) rest-scheme
and *rest2* :: (*'s-rest2*, *'event2*, *'iadv-rest2*, *'iusr-rest2*, *'oadv-rest2*, *'ousr-rest2*, *'more2) rest-scheme*
shows
fused-resource.fuse core (*parallel-rest rest1 rest2*) (*s-core*, (*s-rest1*, *s-rest2*)) =
map-fun (*map-sum* (*lsumr* ∘ *map-sum id swap-sum*) (*lsumr* ∘ *map-sum id swap-sum*)) (*map-spmf* (*map-prod* (*map-sum* (*map-sum id swap-sum* ∘ *rsuml*) (*map-sum id swap-sum* ∘ *rsuml*)) (*map-prod id prod.swap* ∘ *rprodl*)))
(*fused-resource.fuse* (*core-with-rest core rest2*) *rest1* ((*s-core*, *s-rest2*), *s-rest1*))
apply(*rule ext*)
subgoal for *x*
apply(*cases* (*parallel-rest rest1 rest2*, (*s-core*, (*s-rest1*, *s-rest2*)), *x*) *rule*: *fused-resource.fuse.cases*)
apply(*auto simp add*: *fused-resource.fuse.simps map-bind-spmf bind-map-spmf map-prod-def split-def o-def parallel-eoracle-def parallel-oracle-def split*!: *sum.split intro*!: *bind-spmf-cong*)
apply(*subst foldl-spmf-pair-left*[*simplified split-def*]; *simp add*: *map-fun-def o-def bind-map-spmf*)+
done
done

end
theory *State-Isomorphism*
imports
More-CC
begin

6 State Isomorphism

type-synonym
(*'a*, *'b*) *state-iso* = (*'a* ⇒ *'b*) × (*'b* ⇒ *'a*)

definition
state-iso :: (*'a*, *'b*) *state-iso* ⇒ *bool*
where
state-iso ≡ (λ(*f*, *g*). *type-definition f g UNIV*)

definition
apply-state-iso :: (*'s1*, *'s2*) *state-iso* ⇒ (*'s1*, *'i*, *'o*) *oracle'* ⇒ (*'s2*, *'i*, *'o*) *oracle'*
where
apply-state-iso ≡ (λ(*f*, *g*). *map-fun g* (*map-fun id* (*map-spmf* (*map-prod id f*))))

lemma *apply-state-iso-id*: *apply-state-iso* (*id*, *id*) = *id*
by (*auto simp add*: *apply-state-iso-def map-prod.id spmf.map-id0 map-fun-id*)

lemma *apply-state-iso-compose*: *apply-state-iso si1* (*apply-state-iso si2 oracle*) =
apply-state-iso (*map-prod* (λ*f*. *f o* (*fst si2*)) ((*o*) (*snd si2*)) *si1*) *oracle*
unfolding *apply-state-iso-def*
by (*auto simp add*: *split-def id-def o-def map-prod-def map-fun-def map-spmf-conv-bind-spmf*)

lemma *apply-wiring-state-iso-assoc*:
apply-wiring wr (*apply-state-iso si oracle*) = *apply-state-iso si* (*apply-wiring wr oracle*)
unfolding *apply-state-iso-def apply-wiring-def*
by (*auto simp add*: *split-def id-def o-def map-prod-def map-fun-def map-spmf-conv-bind-spmf*)

lemma
resource-of-oracle-state-iso:
assumes *state-iso fg*
shows *resource-of-oracle* (*apply-state-iso fg oracle*) *s* = *resource-of-oracle oracle* (*snd fg s*)
proof −
have [*simp*]: *snd fg* (*fst fg x*) = *x* **for** *x*
using *assms* **by**(*simp add*: *state-iso-def split-beta type-definition.Rep-inverse*)
show *?thesis*
by(*coinduction arbitrary*: *s*)
(*auto 4 3 simp add*: *rel-fun-def spmf-rel-map apply-state-iso-def split-def intro*!: *rel-spmf-reflI*)
qed

6.1 Parallel State Isomorphism

definition
parallel-state-iso :: (('s-core1 × 's-core2) × ('s-rest1 × 's-rest2), ('s-core1 × 's-rest1) × ('s-core2 × 's-rest2)) *state-iso*
where
parallel-state-iso =
(λ((*s11*, *s12*), (*s21*, *s22*)). ((*s11*, *s21*), (*s12*, *s22*)),
λ((*s11*, *s21*), (*s12*, *s22*)). ((*s11*, *s12*), (*s21*, *s22*)))

lemma
state-iso-parallel-state-iso [*simp*]: *state-iso parallel-state-iso*
by (*auto simp add*: *type-definition-def state-iso-def parallel-state-iso-def*)

6.2 Trisplit State Isomorphism

definition
iso-trisplit
where
iso-trisplit =
(λ(((*s11*, *s12*), *s13*), (*s21*, *s22*), *s23*). (((*s11*, *s21*), *s12*, *s22*), *s13*, *s23*),
λ(((*s11*, *s21*), *s12*, *s22*), *s13*, *s23*). (((*s11*, *s12*), *s13*), (*s21*, *s22*), *s23*))

lemma
state-iso-fuse-par [*simp*]: *state-iso iso-trisplit*
by(*simp add*: *state-iso-def iso-trisplit-def*; *unfold-locales*; *simp add*: *split-def*)

6.3 Assocl-Swap State Isomorphism

definition

iso-swapar
where
iso-swapar = (λ((*sm*, *s1*), *s2*). (*s1*, *sm*, *s2*), λ(*s1*, *sm*, *s2*). ((*sm*, *s1*), *s2*))

lemma
state-iso-swapar [*simp*]: *state-iso iso-swapar*
by(*simp add*: *state-iso-def iso-swapar-def*; *unfold-locales*; *simp add*: *split-def*)

end
theory *Construction-Utility*
imports
Fused-Resource
State-Isomorphism
begin

— Dummy converters that return a constant value on their external interface

primcorec
ldummy-converter :: (*'a* ⇒ *'b*) ⇒ (*'i-cnv*, *'o-cnv*, *'i-res*, *'o-res*) *converter* ⇒
(*'a* + *'i-cnv*, *'b* + *'o-cnv*, *'i-res*, *'o-res*) *converter*
where
run-converter (*ldummy-converter f conv*) = (λ*inp*. *case inp of*
Inl x ⇒ *map-gpv* (*map-prod Inl* (λ-. *ldummy-converter f conv*)) *id* (*Done* (*f x*, ()))
| *Inr x* ⇒ *map-gpv* (*map-prod Inr* (λ*c*. *ldummy-converter f c*)) *id* (*run-converter conv x*))

primcorec
rdummy-converter :: (*'a* ⇒ *'b*) ⇒ (*'i-cnv*, *'o-cnv*, *'i-res*, *'o-res*) *converter* ⇒
(*'i-cnv* + *'a*, *'o-cnv* + *'b*, *'i-res*, *'o-res*) *converter*
where
run-converter (*rdummy-converter f conv*) = (λ*inp*. *case inp of*
Inl x ⇒ *map-gpv* (*map-prod Inl* (λ*c*. *rdummy-converter f c*)) *id* (*run-converter conv x*)
| *Inr x* ⇒ *map-gpv* (*map-prod Inr* (λ-. *rdummy-converter f conv*)) *id* (*Done* (*f x*, ())))

lemma *ldummy-converter-of-callee*:
ldummy-converter f (*converter-of-callee callee state*) =
converter-of-callee (λ*s q*. *case-sum* (λ*ql*. *Done* (*Inl* (*f ql*), *s*)) (λ*qr*. *map-gpv* (*map-prod Inr id*) *id* (*callee s qr*)) *q*) *state*
apply (*coinduction arbitrary*: *callee state*)
apply(*clarsimp intro*!:*rel-funI split*!:*sum.splits*)
subgoal by *blast*
apply (*simp add*: *gpv.rel-map map-prod-def split-def*)
by (*rule gpv.rel-mono-strong0*[*of* (=) (=)]) (*auto simp add*: *gpv.rel-eq*)

lemma *rdummy-converter-of-callee*:
rdummy-converter f (*converter-of-callee callee state*) =

converter-of-callee (*λs q. case-sum* (*λql. map-gpv* (*map-prod Inl id*) *id* (*callee s ql*)) (*λqr. Done* (*Inr* (*f qr*), *s*)) *q*) *state*
apply (*coinduction arbitrary*: *callee state*)
apply(*clarsimp intro*!:*rel-funI split*!:*sum.splits*)
defer
subgoal by *blast*
apply (*simp add*: *gpv.rel-map map-prod-def split-def*)
by (*rule gpv.rel-mono-strong0*[*of* (=) (=)]) (*auto simp add*: *gpv.rel-eq*)

— Commonly used wirings

context
fixes
cnv1 :: (*'icnv-usr1*, *'ocnv-usr1*, *'iusr1-res1* + *'iusr1-res2*, *'ousr1-res1* + *'ousr1-res2*) *converter* **and**
cnv2 :: (*'icnv-usr2*, *'ocnv-usr2*, *'iusr2-res1* + *'iusr2-res2*, *'ousr2-res1* + *'ousr2-res2*) *converter*
begin

— c1r22: a converter that has 1 interface and sends queries to two resources, where the first and second resources have 2 and 2 interfaces respectively

definition
wiring-c1r22-c1r22 :: (*'icnv-usr1* + *'icnv-usr2*, *'ocnv-usr1* + *'ocnv-usr2*,
(*'iusr1-res1* + *'iusr2-res1*) + *'iusr1-res2* + *'iusr2-res2*,
(*'ousr1-res1* + *'ousr2-res1*) + *'ousr1-res2* + *'ousr2-res2*) *converter*
where
wiring-c1r22-c1r22 ≡ (*cnv1* $|_=$ *cnv2*) ⊙ *parallel-wiring*

end

— Special wiring converters used for the parallel composition of Fused resources

definition
fused-wiring ::
(((*'iadv-core1* + *'iadv-core2*) + (*'iadv-rest1* + *'iadv-rest2*)) +
((*'iusr-core1* + *'iusr-core2*) + (*'iusr-rest1* + *'iusr-rest2*)),
((*'oadv-core1* + *'oadv-core2*) + (*'oadv-rest1* + *'oadv-rest2*)) +
((*'ousr-core1* + *'ousr-core2*) + (*'ousr-rest1* + *'ousr-rest2*)),
((*'iadv-core1* + *'iadv-rest1*) + (*'iusr-core1* + *'iusr-rest1*)) +
((*'iadv-core2* + *'iadv-rest2*) + (*'iusr-core2* + *'iusr-rest2*)),
((*'oadv-core1* + *'oadv-rest1*) + (*'ousr-core1* + *'ousr-rest1*)) +
((*'oadv-core2* + *'oadv-rest2*) + (*'ousr-core2* + *'ousr-rest2*))) *converter*
where
fused-wiring ≡ (*parallel-wiring* $|_=$ *parallel-wiring*) ⊙ *parallel-wiring*

definition
fused-wiring$_w$
where
fused-wiring$_w$ ≡ (*parallel-wiring*$_w$ $|_w$ *parallel-wiring*$_w$) $\circ_w$ *parallel-wiring*$_w$

schematic-goal
wiring-fused-wiring[*wiring-intro*]: *wiring ?ℐ1 ?ℐ2 fused-wiring fused-wiring*$_w$
unfolding *fused-wiring-def fused-wiring*$_w$*-def*
by(*rule wiring-intro*)+

schematic-goal *WT-fused-wiring* [*WT-intro*]: *?ℐ1*, *?ℐ2* $\vdash_C$ *fused-wiring* $\surd$
unfolding *fused-wiring-def*
by(*rule WT-intro*)+

— Commonlu used attachments

context
fixes
cnv1 :: (*'icnv-usr1*, *'ocnv-usr1*, *'iusr1-core1* + *'iusr1-core2*, *'ousr1-core1* + *'ousr1-core2*) *converter* **and**
cnv2 :: (*'icnv-usr2*, *'ocnv-usr2*, *'iusr2-core1* + *'iusr2-core2*, *'ousr2-core1* + *'ousr2-core2*) *converter* **and**
res1 :: ((*'iadv-core1* + *'iadv-rest1*) + (*'iusr1-core1* + *'iusr2-core1*) + *'iusr-rest1*, (*'oadv-core1* + *'oadv-rest1*) + (*'ousr1-core1* + *'ousr2-core1*) + *'ousr-rest1*) *resource* **and**
res2 :: ((*'iadv-core2* + *'iadv-rest2*) + (*'iusr1-core2* + *'iusr2-core2*) + *'iusr-rest2*, (*'oadv-core2* + *'oadv-rest2*) + (*'ousr1-core2* + *'ousr2-core2*) + *'ousr-rest2*) *resource*
begin

— Attachement of two c1f22 ('f' instead of 'r' to indicate Fused Resources) converters to two 2-interface Fused Resources, the results will be a new 2-interface Fused Resource

definition
attach-c1f22-c1f22 :: (((*'iadv-core1* + *'iadv-core2*) + *'iadv-rest1* + *'iadv-rest2*) + (*'icnv-usr1* + *'icnv-usr2*) + *'iusr-rest1* + *'iusr-rest2*,
((*'oadv-core1* + *'oadv-core2*) + *'oadv-rest1* + *'oadv-rest2*) + (*'ocnv-usr1* + *'ocnv-usr2*) + *'ousr-rest1* + *'ousr-rest2*) *resource*
where
attach-c1f22-c1f22 = (((1_C $|_=$ 1_C) $|_=$ ((*wiring-c1r22-c1r22 cnv1 cnv2*) $|_=$ 1_C)) ⊙ *fused-wiring*) ▷ (*res1* ∥ *res2*)
end

— Properties of Converters attaching to Fused resources
context

fixes
core1 :: (*'s-core1*, *'e1*, *'iadv-core1*, *'iusr-core1*, *'oadv-core1*, *'ousr-core1*) *core* **and**
core2 :: (*'s-core2*, *'e2*, *'iadv-core2*, *'iusr-core2*, *'oadv-core2*, *'ousr-core2*) *core* **and**
rest1 :: (*'s-rest1*, *'e1*, *'iadv-rest1*, *'iusr-rest1*, *'oadv-rest1*, *'ousr-rest1*, *'m1*) *rest-scheme* **and**
rest2 :: (*'s-rest2*, *'e2*, *'iadv-rest2*, *'iusr-rest2*, *'oadv-rest2*, *'ousr-rest2*, *'m2*) *rest-scheme*
begin

lemma *parallel-oracle-fuse*:
apply-wiring fused-wiring$_w$ (*parallel-oracle* (*fused-resource.fuse core1 rest1*) (*fused-resource.fuse core2 rest2*)) =
apply-state-iso parallel-state-iso (*fused-resource.fuse* (*parallel-core core1 core2*) (*parallel-rest rest1 rest2*))
supply *fused-resource.fuse.simps*[*simp*]
apply(*rule ext*)+
apply(*clarsimp simp add*: *fused-wiring*$_w$*-def apply-state-iso-def parallel-state-iso-def parallel-wiring*$_w$*-def*)
apply(*simp add*: *apply-wiring-def comp-wiring-def parallel2-wiring-def lassocr*$_w$*-def swap-lassocr*$_w$*-def rassocl*$_w$*-def swap*$_w$*-def*)
subgoal for *s-core1 s-rest1 s-core2 s-rest2 i*
apply(*cases* (*parallel-rest rest1 rest2*, ((*s-core1*, *s-core2*), (*s-rest1*, *s-rest2*)), *i*) *rule*: *fused-resource.fuse.cases*)
apply(*auto split*!: *sum.splits*)
subgoal for *iadv-core*
by (*cases iadv-core*) (*auto simp add*: *map-spmf-bind-spmf bind-map-spmf intro*!: *bind-spmf-cong split*!: *sum.splits*)
subgoal for *iadv-rest*
by (*cases iadv-rest*) (*auto simp add*: *parallel-handler-left parallel-handler-right foldl-spmf-pair-left*
parallel-eoracle-def foldl-spmf-pair-right split-beta o-def map-spmf-bind-spmf bind-map-spmf)
subgoal for *iusr-core*
by (*cases iusr-core*) (*auto simp add*: *map-spmf-bind-spmf bind-map-spmf intro*!: *bind-spmf-cong split*!: *sum.splits*)
subgoal for *iusr-rest*
by (*cases iusr-rest*) (*auto simp add*: *parallel-handler-left parallel-handler-right foldl-spmf-pair-left*
parallel-eoracle-def foldl-spmf-pair-right split-beta o-def map-spmf-bind-spmf bind-map-spmf)
done
done
end

lemma *attach-callee-fuse*:
attach-callee ((*cnv-adv-core* $\ddagger_I$ *cnv-adv-rest*) $\ddagger_I$ *cnv-usr-core* $\ddagger_I$ *cnv-usr-rest*) (*fused-resource.fuse core rest*) =
apply-state-iso iso-trisplit (*fused-resource.fuse* (*attach-core cnv-adv-core cnv-usr-core*

```
core) (attach-rest cnv-adv-rest cnv-usr-rest f-init rest))
  (is ?lhs = ?rhs)
proof(intro ext; clarify)
  note fused-resource.fuse.simps [simp]
  let ?tri = λ(((s11, s12), s13), (s21, s22), s23). (((s11, s21), s12, s22), s13, s23)
  fix q :: ('g + 'h) + 'i + 'j
  consider (ACore) qac where q = Inl (Inl qac)
    | (ARest) qar where q = Inl (Inr qar)
    | (UCore) quc where q = Inr (Inl quc)
    | (URest) qur where q = Inr (Inr qur)
    using fuse-callee.cases by blast
  then show ?lhs (((sac, sar), (suc, sur)), (sc, sr)) q = ?rhs (((sac, sar), (suc,
sur)), (sc, sr)) q
    for sac sar suc sur sc sr
  proof cases
    case ACore
    have map-spmf rprodl (exec-gpv (fused-resource.fuse core rest)
      (left-gpv (map-gpv (map-prod Inl (λs1'. (s1', suc, sur))) id (left-gpv (map-gpv
(map-prod Inl (λs1'. (s1', sar))) id (cnv-adv-core sac qac)))))
       (sc, sr)) =
     (map-spmf (map-prod (Inl ∘ Inl) (?tri ∘ prod.swap ∘ Pair ((sar, sur), sr)))
       (map-spmf (λ((oadv, s-adv'), s-core'). (oadv, (s-adv', suc), s-core'))
         (exec-gpv (cfunc-adv core) (cnv-adv-core sac qac) sc)))
    proof(induction arbitrary: sc cnv-adv-core rule: exec-gpv-fixp-parallel-induct)
      case adm show ?case by simp
      case bottom show ?case by simp
      case (step execl execr)
      show ?case
       apply(clarsimp simp add: gpv.map-sel map-bind-spmf bind-map-spmf intro!:
bind-spmf-cong[OF refl] split!: generat.split)
        apply(subst step.IH[unfolded id-def])
        apply(simp add: spmf.map-comp o-def)
        done
    qed
    then show ?thesis using ACore
    by(simp add: apply-state-iso-def iso-trisplit-def map-spmf-conv-bind-spmf[symmetric]
spmf.map-comp o-def split-def)
  next
    case ARest
    have bind-spmf (foldl-spmf (cpoke core) (return-spmf sc) es) (λsc'.
      map-spmf rprodl (exec-gpv (fused-resource.fuse core rest)
      (left-gpv (map-gpv (map-prod Inl (λs1'. (s1', suc, sur))) id (right-gpv (map-gpv
(map-prod Inr (Pair sac)) id (cnv-adv-rest sar qar)))))
        (sc', sr))) =
      bind-spmf
       (exec-gpv (λ(s, es) q. map-spmf (λ((out, e), s'). (out, s', es @ e)) (rfunc-adv
rest s q)) (cnv-adv-rest sar qar) (sr, es))
        (map-spmf (map-prod id ?tri) ∘
            ((λ((o-rfunc, e-lst), s-rfunc). map-spmf (λs-notify. (Inl (Inr o-rfunc),
```

s-notify, *s-rfunc*))
(*map-spmf* (*Pair* (*sac*, *suc*)) (*foldl-spmf* (*cpoke core*) (*return-spmf sc*) *e-lst*))) ∘
(λ((*oadv*, *s-adv′*), *s-rest′*, *es*). ((*oadv*, *es*), (*s-adv′*, *sur*), *s-rest′*))))
for *es*
proof(*induction arbitrary*: *sc sr es cnv-adv-rest rule*: *exec-gpv-fixp-parallel-induct*)
case *adm* **then show** *?case* **by** *simp*
case *bottom* **then show** *?case* **by** *simp*
case (*step execl execr*)
show *?case*
apply(*clarsimp simp add*: *gpv.map-sel map-bind-spmf bind-map-spmf*)
apply(*subst bind-commute-spmf*)
apply(*clarsimp simp add*: *gpv.map-sel map-bind-spmf bind-map-spmf spmf.map-comp o-def map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *generat.split*)
apply(*subst bind-commute-spmf*)
apply(*clarsimp simp add*: *gpv.map-sel map-bind-spmf bind-map-spmf spmf.map-comp o-def map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *generat.split*)
apply(*simp add*: *bind-spmf-assoc*[*symmetric*] *bind-foldl-spmf-return foldl-spmf-append*[*symmetric*] *del*: *bind-spmf-assoc*)
apply(*subst step.IH*[*unfolded id-def*])
apply(*simp add*: *split-def o-def spmf.map-comp*)
done
qed
from *this*[*of* []]
show *?thesis* **using** *ARest*
by(*simp add*: *apply-state-iso-def iso-trisplit-def map-bind-spmf bind-map-spmf map-spmf-conv-bind-spmf*[*symmetric*] *foldl-spmf-pair-right*)
next
case *UCore*
have *map-spmf rprodl* (*exec-gpv* (*fused-resource.fuse core rest*)
(*right-gpv* (*map-gpv* (*map-prod Inr* (*Pair* (*sac*, *sar*))) *id* (*left-gpv* (*map-gpv* (*map-prod Inl* (λ*s1′*. (*s1′*, *sur*))) *id* (*cnv-usr-core suc quc*)))))
(*sc*, *sr*)) =
(*map-spmf* (*map-prod* (*Inr* ∘ *Inl*) (*?tri* ∘ *prod.swap* ∘ *Pair* ((*sar*, *sur*), *sr*)))
(*map-spmf* (λ((*ousr*, *s-usr′*), *s-core′*). (*ousr*, (*sac*, *s-usr′*), *s-core′*))
(*exec-gpv* (*cfunc-usr core*) (*cnv-usr-core suc quc*) *sc*)))
proof(*induction arbitrary*: *sc cnv-usr-core rule*: *exec-gpv-fixp-parallel-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step execl execr*)
show *?case*
apply(*clarsimp simp add*: *gpv.map-sel map-bind-spmf bind-map-spmf intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *generat.split*)
apply(*subst step.IH*[*unfolded id-def*])
apply(*simp add*: *spmf.map-comp o-def*)
done
qed

then show *?thesis* **using** *UCore*
by(*simp add*: *apply-state-iso-def iso-trisplit-def map-spmf-conv-bind-spmf*[*symmetric*] *spmf.map-comp o-def split-def*)
next
case *URest*
have *bind-spmf* (*foldl-spmf* (*cpoke core*) (*return-spmf sc*) *es*) (λ*sc′*.
map-spmf rprodl (*exec-gpv* (*fused-resource.fuse core rest*)
(*right-gpv* (*map-gpv* (*map-prod Inr* (*Pair* (*sac*, *sar*))) *id* (*right-gpv* (*map-gpv* (*map-prod Inr* (*Pair suc*)) *id* (*cnv-usr-rest sur qur*)))))
(*sc′*, *sr*))) =
bind-spmf
(*exec-gpv* (λ(*s*, *es*) *q*. *map-spmf* (λ((*out*, *e*), *s′*). (*out*, *s′*, *es* @ *e*)) (*rfunc-usr rest s q*)) (*cnv-usr-rest sur qur*) (*sr*, *es*))
(*map-spmf* (*map-prod id ?tri*) ∘
((λ((*o-rfunc*, *e-lst*), *s-rfunc*). *map-spmf* (λ*s-notify*. (*Inr* (*Inr o-rfunc*), *s-notify*, *s-rfunc*))
(*map-spmf* (*Pair* (*sac*, *suc*)) (*foldl-spmf* (*cpoke core*) (*return-spmf sc*) *e-lst*))) ∘
(λ((*ousr*, *s-usr′*), *s-rest′*, *es*). ((*ousr*, *es*), (*sar*, *s-usr′*), *s-rest′*))))
for *es*
proof(*induction arbitrary*: *sc sr es cnv-usr-rest rule*: *exec-gpv-fixp-parallel-induct*)
case *adm* **then show** *?case* **by** *simp*
case *bottom* **then show** *?case* **by** *simp*
case (*step execl execr*)
show *?case*
apply(*clarsimp simp add*: *gpv.map-sel map-bind-spmf bind-map-spmf*)
apply(*subst bind-commute-spmf*)
apply(*clarsimp simp add*: *gpv.map-sel map-bind-spmf bind-map-spmf spmf.map-comp o-def map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *generat.split*)
apply(*subst bind-commute-spmf*)
apply(*clarsimp simp add*: *gpv.map-sel map-bind-spmf bind-map-spmf spmf.map-comp o-def map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *generat.split*)
apply(*simp add*: *bind-spmf-assoc*[*symmetric*] *bind-foldl-spmf-return foldl-spmf-append*[*symmetric*] *del*: *bind-spmf-assoc*)
apply(*subst step.IH*[*unfolded id-def*])
apply(*simp add*: *split-def o-def spmf.map-comp*)
done
qed
from *this*[*of* []] **show** *?thesis* **using** *URest*
by(*simp add*: *apply-state-iso-def iso-trisplit-def map-bind-spmf bind-map-spmf map-spmf-conv-bind-spmf*[*symmetric*] *foldl-spmf-pair-right*)
qed
qed

lemma *attach-parallel-fuse′*:
(*CNV cnv-adv-core s-a-c* $|_=$ *CNV cnv-adv-rest s-a-r*) $|_=$ (*CNV cnv-usr-core s-u-c* $|_=$ *CNV cnv-usr-rest s-u-r*) ▷

RES (*fused-resource.fuse core rest*) (*s-r-c*, *s-r-r*) =
RES (*fused-resource.fuse* (*attach-core cnv-adv-core cnv-usr-core core*) (*attach-rest cnv-adv-rest cnv-usr-rest f-init rest*)) (((*s-a-c*, *s-u-c*), *s-r-c*), ((*s-a-r*, *s-u-r*), *s-r-r*))
apply(*fold conv-callee-parallel*)
apply(*unfold attach-CNV-RES*)
apply(*subst attach-callee-fuse*)
apply(*subst resource-of-oracle-state-iso*)
apply *simp*
apply(*simp add*: *iso-trisplit-def*)
done

— Moving event translators from rest to the core

context
fixes
einit :: *'s-event* **and**
etran :: (*'s-event*, *'ievent*, *'oevent list*) *oracle'* **and**
rest :: (*'s-rest*, *'ievent*, *'iadv-rest*, *'iusr-rest*, *'oadv-rest*, *'ousr-rest*) *rest-wstate* **and**
core :: (*'s-core*, *'oevent*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
begin

primcorec
translate-rest :: (*'s-event* × *'s-rest*, *'oevent*, *'iadv-rest*, *'iusr-rest*, *'oadv-rest*, *'ousr-rest*) *rest-wstate*
where
rinit translate-rest = (*einit*, *rinit rest*)
| *rfunc-adv translate-rest* = *translate-eoracle etran* (*extend-state-oracle* (*rfunc-adv rest*))
| *rfunc-usr translate-rest* = *translate-eoracle etran* (*extend-state-oracle* (*rfunc-usr rest*))

primcorec
translate-core :: (*'s-event* × *'s-core*, *'ievent*, *'iadv-core*, *'iusr-core*, *'oadv-core*, *'ousr-core*) *core*
where
cpoke translate-core = (λ(*s-event*, *s-core*) *event*.
bind-spmf (*etran s-event event*) (λ(*events*, *s-event'*).
map-spmf (λ*s-core'*. (*s-event'*, *s-core'*)) (*foldl-spmf* (*cpoke core*) (*return-spmf s-core*) *events*)))
| *cfunc-adv translate-core* = *extend-state-oracle* (*cfunc-adv core*)
| *cfunc-usr translate-core* = *extend-state-oracle* (*cfunc-usr core*)

lemma *WT-translate-rest* [*WT-intro*]:
assumes *WT-rest* *ℐ-adv* *ℐ-usr* *I-rest rest*
shows *WT-rest* *ℐ-adv* *ℐ-usr* (*pred-prod* (λ-. *True*) *I-rest*) *translate-rest*
by(*rule WT-rest.intros*)(*auto simp add*: *translate-eoracle-def simp add*: *WT-restD-rinit*[*OF*

assms] *dest*!: *WT-restD*(*1*,*2*)[*OF assms*])

lemma *fused-resource-move-translate*:
fused-resource.fuse core translate-rest = *apply-state-iso iso-swapar* (*fused-resource.fuse translate-core rest*)
proof −
note [*simp*] = *exec-gpv-bind spmf.map-comp o-def map-bind-spmf bind-map-spmf bind-spmf-const*

```
show ?thesis
  apply (rule ext)
  apply (rule ext)
  subgoal for s query
    apply (cases s)
    subgoal for s-core s-event s-rest
      apply (cases query)
      subgoal for q-adv
        apply (cases q-adv)
        subgoal for q-acore
         by (simp add: apply-state-iso-def iso-swapar-def fused-resource.fuse.simps
split-def map-prod-def)
        subgoal for q-arest
      apply (simp add: apply-state-iso-def iso-swapar-def fused-resource.fuse.simps)
          apply (simp add: translate-eoracle-def split-def)
          apply(rule bind-spmf-cong[OF refl])
          apply(subst foldl-spmf-chain[simplified split-def])
          by simp
        done
      subgoal for q-usr
        apply (cases q-usr)
        subgoal for q-ucore
         by (simp add: apply-state-iso-def iso-swapar-def fused-resource.fuse.simps
split-def map-prod-def)
        subgoal for q-urest
      apply (simp add: apply-state-iso-def iso-swapar-def fused-resource.fuse.simps)
          apply (simp add: translate-eoracle-def split-def)
          apply(rule bind-spmf-cong[OF refl])
          apply(subst foldl-spmf-chain[simplified split-def])
          by simp
        done
      done
    done
  done
qed
```

end

— Moving interfaces between rest and core

lemma
fuse-ishift-core-to-rest:
assumes *cpoke core′* = (*λs. case-sum* (*λq. fn s q*) (*cpoke core s*))
and *cfunc-adv core* = *cfunc-adv core′*
and *cfunc-usr core* = *cfunc-usr core′* $\oplus_O$ (*λs i. map-spmf* (*Pair* (*h-out i*)) (*fn s i*))
and *rfunc-adv rest′* = (*λs q. map-spmf* (*apfst* (*apsnd* (*map Inr*))) (*rfunc-adv rest s q*))
and *rfunc-usr rest′* = *plus-eoracle* (*λs i. return-spmf* ((*h-out i*, [*i*]), *s*)) (*rfunc-usr rest*)
shows *fused-resource.fuse core rest* = *apply-wiring* (1_w $|_w$ *lassocr*$_w$) (*fused-resource.fuse core′ rest′*) (**is** *?L* = *?R*)
proof –
note [*simp*] = *fused-resource.fuse.simps apply-wiring-def lassocr*$_w$*-def parallel2-wiring-def*

plus-eoracle-def map-spmf-conv-bind-spmf map-prod-def map-fun-def split-def o-def

have *?L s q* = *?R s q* **for** *s q*
apply (*cases q*; *cases s*)
subgoal for *q-adv*
by (*cases q-adv*) (*simp-all add*: *assms*(*1*, *2*, *4*))
subgoal for *q-usr*
apply (*cases q-usr*)
subgoal for *q-usr-core*
apply (*cases q-usr-core*)
subgoal for *q-nrm*
by (*simp add*: *assms*(*3*))
by (*simp add*: *assms*(*1*, *3*, *5*))
by (*simp add*: *assms*(*1*, *5*))
done

then show *?thesis*
by *blast*
qed

lemma *move-simulator-interface*:
defines *x-ifunc* ≡ (*λifunc core* (*se*, *sc*) *q. do* {
((*out*, *es*), *se′*) ← *ifunc se q*;
sc′ ← *foldl-spmf* (*cpoke core*) (*return-spmf sc*) *es*;
return-spmf (*out*, *se′*, *sc′*) })
assumes *cpoke core′* = *cpoke* (*translate-core etran core*)
and *cfunc-adv core′* = †(*cfunc-adv core*) $\oplus_O$ *x-ifunc ifunc core*
and *cfunc-usr core′* = *cfunc-usr* (*translate-core etran core*)

and *rinit rest* = (*einit*, *rinit rest'*)
and *rfunc-adv rest* = (λ*s q*. *case q of*
Inl ql ⇒ *map-spmf* (*apfst* (*map-prod Inl id*)) ((*ifunc*†) *s ql*)
| *Inr qr* ⇒ *map-spmf* (*apfst* (*map-prod Inr id*)) ((*translate-eoracle etran* (†(*rfunc-adv rest'*))) *s qr*))
and *rfunc-usr rest* = *translate-eoracle etran* (†(*rfunc-usr rest'*))
shows *fused-resource.fuse core rest* = *apply-wiring* (*rassocl*$_w$ $|_w$ (*id*, *id*))
(*apply-state-iso* (*rprodl o* (*apfst prod.swap*), (*apfst prod.swap*) *o lprodr*)
(*fused-resource.fuse core' rest'*))
(**is** *?L* = *?R*)
proof −
note [*simp*] = *fused-resource.fuse.simps apply-wiring-def rassocl*$_w$*-def parallel2-wiring-def apply-state-iso-def*
exec-gpv-bind spmf.map-comp map-bind-spmf bind-map-spmf bind-spmf-const o-def split-def

have *?L* (*sc*, *st*, *sr*) *q* = *?R* (*sc*, *st*, *sr*) *q* **for** *sc st sr q*
apply (*simp add*: *map-fun-def map-prod-def prod.swap-def apfst-def lprodr-def rprodl-def id-def*)
using *assms* **apply** (*cases q*)
subgoal for *q-adv*
apply (*cases q-adv*)
subgoal for *q-adv-core*
by (*simp add*: *map-prod-def*)
subgoal for *q-adv-rest*
apply (*cases q-adv-rest*)
subgoal for *q-adv-rest-ifunc*
by *simp*
subgoal for *q-adv-rest-etran*
apply (*simp add*: *translate-eoracle-def*)
apply(*rule bind-spmf-cong*[*OF refl*])
apply(*subst foldl-spmf-chain*[*simplified split-def*])
by *simp*
done
done
subgoal for *q-usr*
apply (*cases q-usr*)
subgoal for *q-usr-core*
by (*simp add*: *map-prod-def*)
subgoal for *q-usr-rest*
apply (*simp add*: *translate-eoracle-def extend-state-oracle-def*)
apply(*rule bind-spmf-cong*[*OF refl*])
apply(*subst foldl-spmf-chain*[*simplified split-def*])
by *simp*
done
done

then show *?thesis*
by *force*

qed

end
theory *Concrete-Security*
imports
Observe-Failure
Construction-Utility
begin

7 Concrete security definition

locale *constructive-security-aux-obsf* =
fixes *real-resource* :: ('*a* + '*e*, '*b* + '*f*) *resource*
and *ideal-resource* :: ('*c* + '*e*, '*d* + '*f*) *resource*
and *sim* :: ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$-*real* :: ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$-*ideal* :: ('*c*, '*d*) $\mathcal{I}$
and $\mathcal{I}$-*common* :: ('*e*, '*f*) $\mathcal{I}$
and *adv* :: *real*
assumes *WT-real* [*WT-intro*]: $\mathcal{I}$-*real* $\oplus_\mathcal{I}$ $\mathcal{I}$-*common* ⊢*res real-resource* $\surd$
and *WT-ideal* [*WT-intro*]: $\mathcal{I}$-*ideal* $\oplus_\mathcal{I}$ $\mathcal{I}$-*common* ⊢*res ideal-resource* $\surd$
and *WT-sim* [*WT-intro*]: $\mathcal{I}$-*real*, $\mathcal{I}$-*ideal* $\vdash_C$ *sim* $\surd$
and *pfinite-sim* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$-*real* $\mathcal{I}$-*ideal sim*
and *adv-nonneg*: $0 \leq adv$

locale *constructive-security-sim-obsf* =
fixes *real-resource* :: ('*a* + '*e*, '*b* + '*f*) *resource*
and *ideal-resource* :: ('*c* + '*e*, '*d* + '*f*) *resource*
and *sim* :: ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$-*real* :: ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$-*common* :: ('*e*, '*f*) $\mathcal{I}$
and $\mathcal{A}$ :: ('*a* + '*e*, '*b* + '*f*) *distinguisher-obsf*
and *adv* :: *real*
assumes *adv*: ⟦ *exception*-$\mathcal{I}$ ($\mathcal{I}$-*real* $\oplus_\mathcal{I}$ $\mathcal{I}$-*common*) ⊢*g* $\mathcal{A}$ $\surd$ ⟧
⟹ *advantage* $\mathcal{A}$ (*obsf-resource* (*sim* $|_=$ 1_C ⊳ *ideal-resource*)) (*obsf-resource* (*real-resource*)) ≤ *adv*

locale *constructive-security-obsf* = *constructive-security-aux-obsf real-resource ideal-resource sim* $\mathcal{I}$-*real* $\mathcal{I}$-*ideal* $\mathcal{I}$-*common adv*
+ *constructive-security-sim-obsf real-resource ideal-resource sim* $\mathcal{I}$-*real* $\mathcal{I}$-*common* $\mathcal{A}$ *adv*
for *real-resource* :: ('*a* + '*e*, '*b* + '*f*) *resource*
and *ideal-resource* :: ('*c* + '*e*, '*d* + '*f*) *resource*
and *sim* :: ('*a*, '*b*, '*c*, '*d*) *converter*
and $\mathcal{I}$-*real* :: ('*a*, '*b*) $\mathcal{I}$
and $\mathcal{I}$-*ideal* :: ('*c*, '*d*) $\mathcal{I}$
and $\mathcal{I}$-*common* :: ('*e*, '*f*) $\mathcal{I}$
and $\mathcal{A}$ :: ('*a* + '*e*, '*b* + '*f*) *distinguisher-obsf*

and *adv* :: *real*
begin

lemma *constructive-security-aux-obsf*: *constructive-security-aux-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common adv* **..**
lemma *constructive-security-sim-obsf*: *constructive-security-sim-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-common $\mathcal{A}$ adv* **..**

end

context *constructive-security-aux-obsf* **begin**

lemma *constructive-security-obsf-refl*:
constructive-security-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common $\mathcal{A}$
(*advantage $\mathcal{A}$* (*obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) (*obsf-resource* (*real-resource*)))
by *unfold-locales*(*simp-all add*: *advantage-def WT-intro pfinite-intro*)

end

lemma *constructive-security-obsf-absorb-cong*:
assumes *sec*: *constructive-security-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common* (*absorb $\mathcal{A}$ cnv*) *adv*
and [*WT-intro*]: *exception-$\mathcal{I}$ $\mathcal{I}$*, *exception-$\mathcal{I}$* (*$\mathcal{I}$-real* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common*) $\vdash_C$ *cnv* $\surd$ *exception-$\mathcal{I}$ $\mathcal{I}$*, *exception-$\mathcal{I}$* (*$\mathcal{I}$-real* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common*) $\vdash_C$ *cnv'* $\surd$ *exception-$\mathcal{I}$ $\mathcal{I}$* $\vdash g$ $\mathcal{A}$ $\surd$
and *cong*: *exception-$\mathcal{I}$ $\mathcal{I}$*, *exception-$\mathcal{I}$* (*$\mathcal{I}$-real* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common*) $\vdash_C$ *cnv* $\sim$ *cnv'*
shows *constructive-security-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common* (*absorb $\mathcal{A}$ cnv'*) *adv*
proof −
interpret *constructive-security-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common absorb $\mathcal{A}$ cnv adv* **by** *fact*
show *?thesis*
proof
have *connect-obsf $\mathcal{A}$* (*cnv'* $\rhd$ *obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) = *connect-obsf $\mathcal{A}$* (*cnv* $\rhd$ *obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*))
connect-obsf $\mathcal{A}$ (*cnv'* $\rhd$ *obsf-resource real-resource*) = *connect-obsf $\mathcal{A}$* (*cnv* $\rhd$ *obsf-resource real-resource*)
by(*rule connect-eq-resource-cong eq-$\mathcal{I}$-attach-on' WT-intro cong*[*symmetric*] *order-refl*)+
then have *advantage* (*absorb $\mathcal{A}$ cnv'*) (*obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) (*obsf-resource real-resource*) =
advantage (*absorb $\mathcal{A}$ cnv*) (*obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) (*obsf-resource real-resource*)
unfolding *advantage-def distinguish-attach*[*symmetric*] **by** *simp*
also have . . . $\leq$ *adv* **by**(*rule adv*)(*rule WT-intro*)+
finally show *advantage* (*absorb $\mathcal{A}$ cnv'*) (*obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) (*obsf-resource real-resource*) $\leq$ *adv* **.**

qed
qed

lemma *constructive-security-obsf-sim-cong*:
assumes *sec*: *constructive-security-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common $\mathcal{A}$ adv*
and *cong*: $\mathcal{I}$*-real*, $\mathcal{I}$*-ideal* $\vdash_C$ *sim* $\sim$ *sim'*
and *pfinite* [*pfinite-intro*]: *pfinite-converter $\mathcal{I}$-real $\mathcal{I}$-ideal sim'*
shows *constructive-security-obsf real-resource ideal-resource sim' $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common $\mathcal{A}$ adv*
proof
interpret *constructive-security-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common $\mathcal{A}$ adv* **by** *fact*
show $\mathcal{I}$*-real* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* $\vdash$*res real-resource* $\surd$ $\mathcal{I}$*-ideal* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* $\vdash$*res ideal-resource* $\surd$ **by**(*rule WT-intro*)+
from *cong* **show** [*WT-intro*]: $\mathcal{I}$*-real*, $\mathcal{I}$*-ideal* $\vdash_C$ *sim'* $\surd$ **by**(*rule eq-$\mathcal{I}$-converterD-WT1*)(*rule WT-intro*)
show *pfinite-converter $\mathcal{I}$-real $\mathcal{I}$-ideal sim'* **by** *fact*
show *0* $\leq$ *adv* **by**(*rule adv-nonneg*)

assume *WT* [*WT-intro*]: *exception-$\mathcal{I}$* ($\mathcal{I}$*-real* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common*) $\vdash$*g* $\mathcal{A}$ $\surd$
have *connect-obsf* $\mathcal{A}$ (*obsf-resource* (*sim'* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) = *connect-obsf* $\mathcal{A}$ (*obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*))
by(*rule connect-eq-resource-cong WT-intro obsf-resource-eq-$\mathcal{I}$-cong eq-$\mathcal{I}$-attach-on' parallel-converter2-eq-$\mathcal{I}$-cong cong*[*symmetric*] *eq-$\mathcal{I}$-converter-reflI* | *simp*)+
with *adv*[*OF WT*]
show *advantage* $\mathcal{A}$ (*obsf-resource* (*sim'* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) (*obsf-resource real-resource*) $\leq$ *adv*
unfolding *advantage-def* **by** *simp*
qed

lemma *constructive-security-obsfI-core-rest* [*locale-witness*]:
assumes *constructive-security-aux-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal* ($\mathcal{I}$*-common-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common-rest*) *adv*
and *adv*: $[\![$ *exception-$\mathcal{I}$* ($\mathcal{I}$*-real* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-common-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common-rest*)) $\vdash$*g* $\mathcal{A}$ $\surd$ $]\!]$
$\Longrightarrow$ *advantage* $\mathcal{A}$ (*obsf-resource* (*sim* $|_=$ (*1*$_C$ $|_=$ *1*$_C$) $\rhd$ *ideal-resource*)) (*obsf-resource* (*real-resource*)) $\leq$ *adv*
shows *constructive-security-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal* ($\mathcal{I}$*-common-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common-rest*) $\mathcal{A}$ *adv*
proof –
interpret *constructive-security-aux-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-ideal $\mathcal{I}$-common-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common-rest* **by** *fact*
show *?thesis*
proof
assume *A* [*WT-intro*]: *exception-$\mathcal{I}$* ($\mathcal{I}$*-real* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-common-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common-rest*)) $\vdash$*g* $\mathcal{A}$ $\surd$
hence *outs*: *outs-gpv* (*exception-$\mathcal{I}$* ($\mathcal{I}$*-real* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-common-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common-rest*))) $\mathcal{A}$ $\subseteq$ *outs-$\mathcal{I}$* ($\mathcal{I}$*-real* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-common-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common-rest*))

unfolding *WT-gpv-iff-outs-gpv* **by** *simp*
have *connect-obsf* $\mathcal{A}$ (*obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) = *connect-obsf* $\mathcal{A}$ (*obsf-resource* (*sim* $|_=$ *1*$_C$ $|_=$ *1*$_C$ $\rhd$ *ideal-resource*))
by(*rule connect-cong-trace trace-eq-obsf-resourceI eq-resource-on-imp-trace-eq eq-*$\mathcal{I}$*-attach-on'*)+
(*rule WT-intro parallel-converter2-eq-*$\mathcal{I}$*-cong eq-*$\mathcal{I}$*-converter-reflI parallel-converter2-id-id*[*symmetric*] *order-refl outs*)+
then show *advantage* $\mathcal{A}$ (*obsf-resource* (*sim* $|_=$ *1*$_C$ $\rhd$ *ideal-resource*)) (*obsf-resource real-resource*) $\leq$ *adv*
using *adv*[*OF A*] **by**(*simp add*: *advantage-def*)
qed
qed

7.1 Composition theorems

theorem *constructive-security-obsf-composability*:
fixes *real*
assumes *constructive-security-obsf middle ideal sim-inner* $\mathcal{I}$*-middle* $\mathcal{I}$*-inner* $\mathcal{I}$*-common* (*absorb* $\mathcal{A}$ (*obsf-converter* (*sim-outer* $|_=$ *1*$_C$))) *adv1*
assumes *constructive-security-obsf real middle sim-outer* $\mathcal{I}$*-real* $\mathcal{I}$*-middle* $\mathcal{I}$*-common* $\mathcal{A}$ *adv2*
shows *constructive-security-obsf real ideal* (*sim-outer* $\odot$ *sim-inner*) $\mathcal{I}$*-real* $\mathcal{I}$*-inner* $\mathcal{I}$*-common* $\mathcal{A}$ (*adv1* + *adv2*)
proof
let *?*$\mathcal{A}$ = *absorb* $\mathcal{A}$ (*obsf-converter* (*sim-outer* $|_=$ *1*$_C$))
interpret *inner*: *constructive-security-obsf middle ideal sim-inner* $\mathcal{I}$*-middle* $\mathcal{I}$*-inner* $\mathcal{I}$*-common ?*$\mathcal{A}$ *adv1* **by** *fact*
interpret *outer*: *constructive-security-obsf real middle sim-outer* $\mathcal{I}$*-real* $\mathcal{I}$*-middle* $\mathcal{I}$*-common* $\mathcal{A}$ *adv2* **by** *fact*

show $\mathcal{I}$*-real* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* $\vdash$*res real* $\surd$
and $\mathcal{I}$*-inner* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* $\vdash$*res ideal* $\surd$
and $\mathcal{I}$*-real*, $\mathcal{I}$*-inner* $\vdash_C$ *sim-outer* $\odot$ *sim-inner* $\surd$ **by**(*rule WT-intro*)+
show *pfinite-converter* $\mathcal{I}$*-real* $\mathcal{I}$*-inner* (*sim-outer* $\odot$ *sim-inner*) **by**(*rule pfinite-intro WT-intro*)+
show *0* $\leq$ *adv1* + *adv2* **using** *inner.adv-nonneg outer.adv-nonneg* **by** *simp*

assume *WT-adv*[*WT-intro*]: *exception-*$\mathcal{I}$ ($\mathcal{I}$*-real* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common*) $\vdash$*g* $\mathcal{A}$ $\surd$
have *eq1*: *connect-obsf* (*absorb* $\mathcal{A}$ (*obsf-converter* (*sim-outer* $|_=$ *1*$_C$))) (*obsf-resource* (*sim-inner* $|_=$ *1*$_C$ $\rhd$ *ideal*)) =
connect-obsf $\mathcal{A}$ (*obsf-resource* (*sim-outer* $\odot$ *sim-inner* $|_=$ *1*$_C$ $\rhd$ *ideal*))
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del*: *outs-plus-*$\mathcal{I}$ *add*: *parallel-converter2-comp1-out attach-compose*)
apply(*rule obsf-attach*)
apply(*rule pfinite-intro WT-intro*)+
done
have *eq2*: *connect-obsf* (*absorb* $\mathcal{A}$ (*obsf-converter* (*sim-outer* $|_=$ *1*$_C$))) (*obsf-resource*

middle) =
connect-obsf $\mathcal{A}$ (*obsf-resource* (*sim-outer* $|_=$ *1*$_C$ $\rhd$ *middle*))
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del*: *outs-plus-*$\mathcal{I}$ *add*: *parallel-converter2-comp1-out attach-compose*)
apply(*rule obsf-attach*)
apply(*rule pfinite-intro WT-intro*)+
done

have *advantage ?*$\mathcal{A}$ (*obsf-resource* (*sim-inner* $|_=$ *1*$_C$ $\rhd$ *ideal*)) (*obsf-resource* *middle*) $\leq$ *adv1*
by(*rule inner.adv*)(*rule WT-intro*)+
moreover have *advantage* $\mathcal{A}$ (*obsf-resource* (*sim-outer* $|_=$ *1*$_C$ $\rhd$ *middle*)) (*obsf-resource* *real*) $\leq$ *adv2*
by(*rule outer.adv*)(*rule WT-intro*)+
ultimately
show *advantage* $\mathcal{A}$ (*obsf-resource* (*sim-outer* $\odot$ *sim-inner* $|_=$ *1*$_C$ $\rhd$ *ideal*)) (*obsf-resource real*) $\leq$ *adv1* + *adv2*
by(*auto simp add*: *advantage-def eq1 eq2 abs-diff-triangle-ineq2*)
qed

theorem *constructive-security-obsf-lifting*:
assumes *sec*: *constructive-security-aux-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common adv*
and *sec2*: *exception-*$\mathcal{I}$ ($\mathcal{I}$*-real'* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common'*) $\vdash$*g* $\mathcal{A}$ $\surd$
$\Longrightarrow$ *constructive-security-sim-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-common* (*absorb* $\mathcal{A}$ (*obsf-converter* (*w-adv-real* $|_=$ *w-usr*))) *adv*
(**is** - $\Longrightarrow$ *constructive-security-sim-obsf* - - - - - *?*$\mathcal{A}$ -)
assumes *WT-usr* [*WT-intro*]: $\mathcal{I}$*-common'*, $\mathcal{I}$*-common* $\vdash_C$ *w-usr* $\surd$
and *pfinite* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-common'* $\mathcal{I}$*-common w-usr*
and *WT-adv-real* [*WT-intro*]: $\mathcal{I}$*-real'*, $\mathcal{I}$*-real* $\vdash_C$ *w-adv-real* $\surd$
and *WT-w-adv-ideal* [*WT-intro*]: $\mathcal{I}$*-ideal'*, $\mathcal{I}$*-ideal* $\vdash_C$ *w-adv-ideal* $\surd$
and *WT-adv-ideal-inv* [*WT-intro*]: $\mathcal{I}$*-ideal*, $\mathcal{I}$*-ideal'* $\vdash_C$ *w-adv-ideal-inv* $\surd$
and *ideal-inverse*: $\mathcal{I}$*-ideal*, $\mathcal{I}$*-ideal* $\vdash_C$ *w-adv-ideal-inv* $\odot$ *w-adv-ideal* $\sim$ *1*$_C$
and *pfinite-real* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-real'* $\mathcal{I}$*-real w-adv-real*
and *pfinite-ideal* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-ideal* $\mathcal{I}$*-ideal' w-adv-ideal-inv*
shows *constructive-security-obsf* (*w-adv-real* $|_=$ *w-usr* $\rhd$ *real-resource*) (*w-adv-ideal* $|_=$ *w-usr* $\rhd$ *ideal-resource*) (*w-adv-real* $\odot$ *sim* $\odot$ *w-adv-ideal-inv*) $\mathcal{I}$*-real'* $\mathcal{I}$*-ideal'* $\mathcal{I}$*-common'* $\mathcal{A}$ *adv*
(**is** *constructive-security-obsf ?real ?ideal ?sim ?*$\mathcal{I}$*-real ?*$\mathcal{I}$*-ideal* - - -)
proof
interpret *constructive-security-aux-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* **by** *fact*
show $\mathcal{I}$*-real'* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common'* $\vdash$*res ?real* $\surd$
and $\mathcal{I}$*-ideal'* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common'* $\vdash$*res ?ideal* $\surd$
and $\mathcal{I}$*-real'*, $\mathcal{I}$*-ideal'* $\vdash_C$ *?sim* $\surd$ **by**(*rule WT-intro*)+
show *pfinite-converter* $\mathcal{I}$*-real'* $\mathcal{I}$*-ideal' ?sim* **by**(*rule pfinite-intro WT-intro*)+
show *0* $\leq$ *adv* **by**(*rule adv-nonneg*)

assume *WT-adv* [*WT-intro*]: *exception-$\mathcal{I}$* (*$\mathcal{I}$-real′* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common′*) $\vdash g$ $\mathcal{A}$ $\surd$
then interpret *constructive-security-sim-obsf real-resource ideal-resource sim $\mathcal{I}$-real $\mathcal{I}$-common ?$\mathcal{A}$ adv* **by**(*rule sec2*)

have ∗: *advantage ?$\mathcal{A}$* (*obsf-resource* (*sim* $|_=$ 1_C ⊳ *ideal-resource*)) (*obsf-resource real-resource*) ≤ *adv*
by(*rule adv*)(*rule WT-intro*)+

have *ideal*: *connect-obsf ?$\mathcal{A}$* (*obsf-resource* (*sim* $|_=$ 1_C ⊳ *ideal-resource*)) =
connect-obsf $\mathcal{A}$ (*obsf-resource* (*?sim* $|_=$ 1_C ⊳ *?ideal*))
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del*: *outs-plus-$\mathcal{I}$*)
apply(*rule eq-resource-on-trans*[*OF obsf-attach*])
apply(*rule pfinite-intro WT-intro*)+
apply(*rule obsf-resource-eq-$\mathcal{I}$-cong*)
apply(*fold attach-compose*)
apply(*unfold comp-converter-parallel2*)
apply(*rule eq-$\mathcal{I}$-attach-on′*)
apply(*rule WT-intro*)
apply(*rule parallel-converter2-eq-$\mathcal{I}$-cong*)
apply(*unfold comp-converter-assoc*)
apply(*rule eq-$\mathcal{I}$-comp-cong*)
apply(*rule eq-$\mathcal{I}$-converter-reflI*; *rule WT-intro*)
apply(*rule eq-$\mathcal{I}$-converter-trans*[*rotated*])
apply(*rule eq-$\mathcal{I}$-comp-cong*)
apply(*rule eq-$\mathcal{I}$-converter-reflI*; *rule WT-intro*)
apply(*rule ideal-inverse*[*symmetric*])
apply(*unfold comp-converter-id-right comp-converter-id-left*)
apply(*rule eq-$\mathcal{I}$-converter-reflI*; *rule WT-intro*)+
apply *simp*
apply(*rule WT-intro*)+
done
have *real*: *connect-obsf ?$\mathcal{A}$* (*obsf-resource real-resource*) = *connect-obsf* $\mathcal{A}$ (*obsf-resource ?real*)
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del*: *outs-plus-$\mathcal{I}$*)
apply(*rule obsf-attach*)
apply(*rule pfinite-intro WT-intro*)+
done
show *advantage* $\mathcal{A}$ (*obsf-resource* ((*?sim* $|_=$ 1_C) ⊳ *?ideal*)) (*obsf-resource ?real*) ≤ *adv* **using** ∗
unfolding *advantage-def ideal*[*symmetric*] *real*[*symmetric*] **.**
qed

corollary *constructive-security-obsf-lifting-*:
assumes *sec*: *constructive-security-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* (*absorb* $\mathcal{A}$ (*obsf-converter* (*w-adv-real* $|_=$ *w-usr*))) *adv*
assumes *WT-usr* [*WT-intro*]: $\mathcal{I}$*-common′*, $\mathcal{I}$*-common* $\vdash_C$ *w-usr* $\surd$
and *pfinite* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-common′* $\mathcal{I}$*-common w-usr*
and *WT-adv-real* [*WT-intro*]: $\mathcal{I}$*-real′*, $\mathcal{I}$*-real* $\vdash_C$ *w-adv-real* $\surd$
and *WT-w-adv-ideal* [*WT-intro*]: $\mathcal{I}$*-ideal′*, $\mathcal{I}$*-ideal* $\vdash_C$ *w-adv-ideal* $\surd$
and *WT-adv-ideal-inv* [*WT-intro*]: $\mathcal{I}$*-ideal*, $\mathcal{I}$*-ideal′* $\vdash_C$ *w-adv-ideal-inv* $\surd$
and *ideal-inverse*: $\mathcal{I}$*-ideal*, $\mathcal{I}$*-ideal* $\vdash_C$ *w-adv-ideal-inv* $\odot$ *w-adv-ideal* $\sim$ 1_C
and *pfinite-real* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-real′* $\mathcal{I}$*-real w-adv-real*
and *pfinite-ideal* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-ideal* $\mathcal{I}$*-ideal′ w-adv-ideal-inv*
shows *constructive-security-obsf* (*w-adv-real* $|_=$ *w-usr* $\triangleright$ *real-resource*) (*w-adv-ideal* $|_=$ *w-usr* $\triangleright$ *ideal-resource*) (*w-adv-real* $\odot$ *sim* $\odot$ *w-adv-ideal-inv*) $\mathcal{I}$*-real′* $\mathcal{I}$*-ideal′* $\mathcal{I}$*-common′* $\mathcal{A}$ *adv*
proof −
interpret *constructive-security-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common absorb* $\mathcal{A}$ (*obsf-converter* (*w-adv-real* $|_=$ *w-usr*)) *adv* **by** *fact*
from *constructive-security-aux-obsf constructive-security-sim-obsf assms*(*2*−)
show *?thesis* **by**(*rule constructive-security-obsf-lifting*)
qed

theorem *constructive-security-obsf-lifting-usr*:
assumes *sec*: *constructive-security-aux-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common adv*
and *sec2*: *exception-*$\mathcal{I}$ ($\mathcal{I}$*-real* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common′*) $\vdash g$ $\mathcal{A}$ $\surd$
$\Longrightarrow$ *constructive-security-sim-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-common* (*absorb* $\mathcal{A}$ (*obsf-converter* (1_C $|_=$ *conv*))) *adv*
and *WT-conv* [*WT-intro*]: $\mathcal{I}$*-common′*, $\mathcal{I}$*-common* $\vdash_C$ *conv* $\surd$
and *pfinite* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-common′* $\mathcal{I}$*-common conv*
shows *constructive-security-obsf* (1_C $|_=$ *conv* $\triangleright$ *real-resource*) (1_C $|_=$ *conv* $\triangleright$ *ideal-resource*) *sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common′* $\mathcal{A}$ *adv*
by(*rule constructive-security-obsf-lifting*[*OF sec sec2*, **where** *?w-adv-ideal*=1_C **and** *?w-adv-ideal-inv*=1_C, *simplified comp-converter-id-left comp-converter-id-right*])
(*rule WT-intro pfinite-intro id-converter-eq-self order-refl* | *assumption*)+

theorem *constructive-security-obsf-lifting2*:
assumes *sec*: *constructive-security-aux-obsf real-resource ideal-resource sim* ($\mathcal{I}$*-real1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-real2*) ($\mathcal{I}$*-ideal1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-ideal2*) $\mathcal{I}$*-common adv*
and *sec2*: *exception-*$\mathcal{I}$ (($\mathcal{I}$*-real1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-real2*) $\oplus_\mathcal{I}$ $\mathcal{I}$*-common′*) $\vdash g$ $\mathcal{A}$ $\surd$
$\Longrightarrow$ *constructive-security-sim-obsf real-resource ideal-resource sim* ($\mathcal{I}$*-real1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-real2*) $\mathcal{I}$*-common* (*absorb* $\mathcal{A}$ (*obsf-converter* ((1_C $|_=$ 1_C) $|_=$ *conv*))) *adv*
assumes *WT-conv* [*WT-intro*]: $\mathcal{I}$*-common′*, $\mathcal{I}$*-common* $\vdash_C$ *conv* $\surd$
and *pfinite* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}$*-common′* $\mathcal{I}$*-common conv*
shows *constructive-security-obsf* ((1_C $|_=$ 1_C) $|_=$ *conv* $\triangleright$ *real-resource*) ((1_C $|_=$ 1_C) $|_=$ *conv* $\triangleright$ *ideal-resource*) *sim* ($\mathcal{I}$*-real1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-real2*) ($\mathcal{I}$*-ideal1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-ideal2*) $\mathcal{I}$*-common′* $\mathcal{A}$ *adv*
(**is** *constructive-security-obsf ?real ?ideal - ?*$\mathcal{I}$*-real ?*$\mathcal{I}$*-ideal - - -*)
proof −
interpret *constructive-security-aux-obsf real-resource ideal-resource sim* $\mathcal{I}$*-real1*

$\oplus_\mathcal{I}$ *ℐ-real2 ℐ-ideal1* $\oplus_\mathcal{I}$ *ℐ-ideal2 ℐ-common adv* **by** *fact*
have *sim* [*unfolded comp-converter-id-left*]: *ℐ-real1* $\oplus_\mathcal{I}$ *ℐ-real2*,*ℐ-ideal1* $\oplus_\mathcal{I}$ *ℐ-ideal2* $\vdash_C$ (1_C $|_=$ 1_C) ⊙ *sim* ∼ 1_C ⊙ *sim*
by(*rule eq-ℐ-comp-cong*)(*rule parallel-converter2-id-id eq-ℐ-converter-reflI WT-intro*)+
show *?thesis*
apply(*rule constructive-security-obsf-sim-cong*)
apply(*rule constructive-security-obsf-lifting*[*OF sec sec2*, **where** *?w-adv-ideal*=1_C $|_=$ 1_C **and** *?w-adv-ideal-inv*=1_C, *unfolded comp-converter-id-left comp-converter-id-right*])
apply(*assumption*|*rule WT-intro sim pfinite-intro parallel-converter2-id-id*)+
done
qed

theorem *constructive-security-obsf-trivial*:
fixes *res*
assumes [*WT-intro*]: ℐ $\oplus_\mathcal{I}$ *ℐ-common* ⊢*res res* √
shows *constructive-security-obsf res res* 1_C ℐ ℐ *ℐ-common* 𝒜 *0*
proof
show ℐ $\oplus_\mathcal{I}$ *ℐ-common* ⊢*res res* √ **and** ℐ, ℐ $\vdash_C$ 1_C √ **by**(*rule WT-intro*)+
show *pfinite-converter* ℐ ℐ 1_C **by**(*rule pfinite-intro*)

assume *WT* [*WT-intro*]: *exception-ℐ* (ℐ $\oplus_\mathcal{I}$ *ℐ-common*) ⊢*g* 𝒜 √
have *connect-obsf* 𝒜 (*obsf-resource* (1_C $|_=$ 1_C ⊳ *res*)) = *connect-obsf* 𝒜 (*obsf-resource* (1_C ⊳ *res*))
by(*rule connect-eq-resource-cong*[*OF WT*])(*fastforce intro*: *WT-intro eq-ℐ-attach-on′ obsf-resource-eq-ℐ-cong parallel-converter2-id-id*)+
then show *advantage* 𝒜 (*obsf-resource* (1_C $|_=$ 1_C ⊳ *res*)) (*obsf-resource res*) ≤ *0*
unfolding *advantage-def* **by** *simp*
qed *simp*

lemma *parallel-constructive-security-aux-obsf* [*locale-witness*]:
assumes *constructive-security-aux-obsf real1 ideal1 sim1 ℐ-real1 ℐ-inner1 ℐ-common1 adv1*
assumes *constructive-security-aux-obsf real2 ideal2 sim2 ℐ-real2 ℐ-inner2 ℐ-common2 adv2*
shows *constructive-security-aux-obsf* (*parallel-wiring* ⊳ *real1* ∥ *real2*) (*parallel-wiring* ⊳ *ideal1* ∥ *ideal2*) (*sim1* $|_=$ *sim2*)
(*ℐ-real1* $\oplus_\mathcal{I}$ *ℐ-real2*) (*ℐ-inner1* $\oplus_\mathcal{I}$ *ℐ-inner2*) (*ℐ-common1* $\oplus_\mathcal{I}$ *ℐ-common2*)
(*adv1* + *adv2*)
proof
interpret *sec1*: *constructive-security-aux-obsf real1 ideal1 sim1 ℐ-real1 ℐ-inner1 ℐ-common1 adv1* **by** *fact*
interpret *sec2*: *constructive-security-aux-obsf real2 ideal2 sim2 ℐ-real2 ℐ-inner2 ℐ-common2 adv2* **by** *fact*

show (*ℐ-real1* $\oplus_\mathcal{I}$ *ℐ-real2*) $\oplus_\mathcal{I}$ (*ℐ-common1* $\oplus_\mathcal{I}$ *ℐ-common2*) ⊢*res parallel-wiring* ⊳ *real1* ∥ *real2* √
and (*ℐ-inner1* $\oplus_\mathcal{I}$ *ℐ-inner2*) $\oplus_\mathcal{I}$ (*ℐ-common1* $\oplus_\mathcal{I}$ *ℐ-common2*) ⊢*res parallel-wiring* ⊳ *ideal1* ∥ *ideal2* √

and $\mathcal{I}$-*real1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*real2*, $\mathcal{I}$-*inner1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*inner2* $\vdash_C$ *sim1* $|_=$ *sim2* $\surd$ **by**(*rule WT-intro*)+
show *pfinite-converter* ($\mathcal{I}$-*real1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*real2*) ($\mathcal{I}$-*inner1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*inner2*) (*sim1* $|_=$ *sim2*) **by**(*rule pfinite-intro*)+
show *0* $\leq$ *adv1* + *adv2* **using** *sec1.adv-nonneg sec2.adv-nonneg* **by** *simp*
qed

theorem *parallel-constructive-security-obsf*:
assumes *constructive-security-obsf real1 ideal1 sim1* $\mathcal{I}$-*real1* $\mathcal{I}$-*inner1* $\mathcal{I}$-*common1* (*absorb* $\mathcal{A}$ (*obsf-converter* (*parallel-wiring* $\odot$ *parallel-converter* 1_C (*converter-of-resource* (*sim2* $|_=$ 1_C $\rhd$ *ideal2*))))) *adv1*
(**is** *constructive-security-obsf - - - - - - ?*$\mathcal{A}$*1 -*)
assumes *constructive-security-obsf real2 ideal2 sim2* $\mathcal{I}$-*real2* $\mathcal{I}$-*inner2* $\mathcal{I}$-*common2* (*absorb* $\mathcal{A}$ (*obsf-converter* (*parallel-wiring* $\odot$ *parallel-converter* (*converter-of-resource real1*) 1_C))) *adv2*
(**is** *constructive-security-obsf - - - - - - ?*$\mathcal{A}$*2 -*)
shows *constructive-security-obsf* (*parallel-wiring* $\rhd$ *real1* $\parallel$ *real2*) (*parallel-wiring* $\rhd$ *ideal1* $\parallel$ *ideal2*) (*sim1* $|_=$ *sim2*)
($\mathcal{I}$-*real1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*real2*) ($\mathcal{I}$-*inner1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*inner2*) ($\mathcal{I}$-*common1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*common2*)
$\mathcal{A}$ (*adv1* + *adv2*)
proof −
interpret *sec1*: *constructive-security-obsf real1 ideal1 sim1* $\mathcal{I}$-*real1* $\mathcal{I}$-*inner1* $\mathcal{I}$-*common1 ?*$\mathcal{A}$*1 adv1* **by** *fact*
interpret *sec2*: *constructive-security-obsf real2 ideal2 sim2* $\mathcal{I}$-*real2* $\mathcal{I}$-*inner2* $\mathcal{I}$-*common2 ?*$\mathcal{A}$*2 adv2* **by** *fact*

show *?thesis*
proof
assume *WT* [*WT-intro*]: *exception-*$\mathcal{I}$ (($\mathcal{I}$-*real1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*real2*) $\oplus_\mathcal{I}$ ($\mathcal{I}$-*common1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*common2*)) $\vdash g$ $\mathcal{A}$ $\surd$

have ∗∗: *outs-*$\mathcal{I}$ (($\mathcal{I}$-*real1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*real2*) $\oplus_\mathcal{I}$ ($\mathcal{I}$-*common1* $\oplus_\mathcal{I}$ $\mathcal{I}$-*common2*)) $\vdash_R$
((1_C $|_=$ *sim2*) $|_=$ 1_C $|_=$ 1_C) $\odot$ *parallel-wiring* $\rhd$ *real1* $\parallel$ *ideal2* $\sim$
parallel-wiring $\odot$ (*converter-of-resource real1* $|_\propto$ 1_C) $\rhd$ *sim2* $|_=$ 1_C $\rhd$ *ideal2*
unfolding *comp-parallel-wiring*
by(*rule eq-resource-on-trans*, *rule eq-*$\mathcal{I}$*-attach-on*[**where** *conv'*=*parallel-wiring* $\odot$ (1_C $|_=$ *sim2* $|_=$ 1_C)]
, (*rule WT-intro*)+, *rule eq-*$\mathcal{I}$*-comp-cong*, *rule eq-*$\mathcal{I}$*-converter-mono*)
(*auto simp add*: *le-*$\mathcal{I}$*-def attach-compose attach-parallel2 attach-converter-of-resource-conv-parallel-resource*
intro: *WT-intro parallel-converter2-eq-*$\mathcal{I}$*-cong parallel-converter2-id-id eq-*$\mathcal{I}$*-converter-reflI*)

have *ideal2*:
connect-obsf ?$\mathcal{A}$*2* (*obsf-resource* (*sim2* $|_=$ 1_C $\rhd$ *ideal2*)) =
connect-obsf $\mathcal{A}$ (*obsf-resource* ((1_C $|_=$ *sim2*) $|_=$ (1_C $|_=$ 1_C) $\rhd$ *parallel-wiring* $\rhd$ *real1* $\parallel$ *ideal2*))
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)

apply(*simp del*: *outs-plus-$\mathcal{I}$*)
apply(*rule eq-resource-on-trans*[*OF obsf-attach*])
apply(*rule pfinite-intro WT-intro*)+
apply(*rule obsf-resource-eq-$\mathcal{I}$-cong*)
apply(*rule eq-resource-on-sym*)
apply(*subst attach-compose*[*symmetric*])
apply(*rule* $**$)
apply(*rule WT-intro*)+
done

have *real2*: *connect-obsf ?$\mathcal{A}$2* (*obsf-resource real2*) = *connect-obsf* $\mathcal{A}$ (*obsf-resource* (*parallel-wiring* $\rhd$ *real1* $\parallel$ *real2*))
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del*: *outs-plus-$\mathcal{I}$*)
apply(*rule eq-resource-on-trans*[*OF obsf-attach*])
apply(*rule pfinite-intro WT-intro*)+
apply(*rule obsf-resource-eq-$\mathcal{I}$-cong*)
apply(*rule eq-resource-on-sym*)
by(*simp add*: *attach-compose attach-converter-of-resource-conv-parallel-resource*)(*rule WT-intro*)+

have *adv2*: *advantage* $\mathcal{A}$
(*obsf-resource* ((*1*$_C$ $|_=$ *sim2*) $|_=$ (*1*$_C$ $|_=$ *1*$_C$) $\rhd$ *parallel-wiring* $\rhd$ *real1* $\parallel$ *ideal2*))
(*obsf-resource* (*parallel-wiring* $\rhd$ *real1* $\parallel$ *real2*)) $\leq$ *adv2*
unfolding *advantage-def ideal2*[*symmetric*] *real2*[*symmetric*] **by**(*rule sec2.adv*[*unfolded advantage-def*])(*rule WT-intro*)+

have *ideal1*:
connect-obsf ?$\mathcal{A}$1 (*obsf-resource* (*sim1* $|_=$ *1*$_C$ $\rhd$ *ideal1*)) =
connect-obsf $\mathcal{A}$ (*obsf-resource* ((*sim1* $|_=$ *sim2*) $|_=$ (*1*$_C$ $|_=$ *1*$_C$) $\rhd$ *parallel-wiring* $\rhd$ *ideal1* $\parallel$ *ideal2*))
proof −
have $*$:((*outs-$\mathcal{I}$ $\mathcal{I}$-real1* <+> *outs-$\mathcal{I}$ $\mathcal{I}$-real2*) <+> *outs-$\mathcal{I}$ $\mathcal{I}$-common1* <+> *outs-$\mathcal{I}$ $\mathcal{I}$-common2*) $\vdash_R$
(*sim1* $|_=$ *sim2*) $|_=$ (*1*$_C$ $|_=$ *1*$_C$) $\rhd$ *parallel-wiring* $\rhd$ *ideal1* $\parallel$ *ideal2* $\sim$
parallel-wiring $\odot$ (*1*$_C$ $|_\propto$ *converter-of-resource* (*sim2* $|_=$ *1*$_C$ $\rhd$ *ideal2*)) $\rhd$ *sim1* $|_=$ *1*$_C$ $\rhd$ *ideal1*
by(*auto simp add*: *le-$\mathcal{I}$-def comp-parallel-wiring$'$ attach-compose attach-parallel2 attach-converter-of-resource-conv-parallel-resource2 intro*: *WT-intro*)
show *?thesis*
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del*: *outs-plus-$\mathcal{I}$*)
apply(*rule eq-resource-on-trans*[*OF obsf-attach*])
apply(*rule pfinite-intro WT-intro*)+
apply(*rule obsf-resource-eq-$\mathcal{I}$-cong*)

apply(*rule eq-resource-on-sym*)
by(*simp add*: ∗, (*rule WT-intro*)+)
qed

have *real1*: *connect-obsf ?A1* (*obsf-resource real1*) = *connect-obsf* $\mathcal{A}$ (*obsf-resource* ((*1C* $|_=$ *sim2*) $|_=$ (*1C* $|_=$ *1C*) ⊳ *parallel-wiring* ⊳ *real1* ∥ *ideal2*))
proof −
have ∗: *outs-*$\mathcal{I}$ ((*$\mathcal{I}$-real1* $\oplus_\mathcal{I}$ *$\mathcal{I}$-real2*) $\oplus_\mathcal{I}$ (*$\mathcal{I}$-common1* $\oplus_\mathcal{I}$ *$\mathcal{I}$-common2*)) $\vdash_R$ *parallel-wiring* ⊙ ((*1C* $|_=$ *1C*) $|_=$ *sim2* $|_=$ *1C*) ⊳ *real1* ∥ *ideal2* ∼ *parallel-wiring* ⊙ (*1C* $|_\propto$ *converter-of-resource* (*sim2* $|_=$ *1C* ⊳ *ideal2*)) ⊳ *real1*
by(*rule eq-resource-on-trans*, *rule eq-$\mathcal{I}$-attach-on*[**where** *conv′*=*parallel-wiring* ⊙ (*1C* $|_=$ *sim2* $|_=$ *1C*)]
, (*rule WT-intro*)+, *rule eq-$\mathcal{I}$-comp-cong*, *rule eq-$\mathcal{I}$-converter-mono*)
(*auto simp add*: *le-$\mathcal{I}$-def attach-compose attach-converter-of-resource-conv-parallel-resource2 attach-parallel2*
intro: *WT-intro parallel-converter2-eq-$\mathcal{I}$-cong parallel-converter2-id-id eq-$\mathcal{I}$-converter-reflI*)

show *?thesis*
unfolding *distinguish-attach*[*symmetric*]
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del*: *outs-plus-$\mathcal{I}$*)
apply(*rule eq-resource-on-trans*[*OF obsf-attach*])
apply(*rule pfinite-intro WT-intro*)+
apply(*rule obsf-resource-eq-$\mathcal{I}$-cong*)
apply(*rule eq-resource-on-sym*)
apply(*fold attach-compose*)
apply(*subst comp-parallel-wiring*)
apply(*rule* ∗)
apply(*rule WT-intro*)+
done
qed

have *adv1*: *advantage* $\mathcal{A}$
(*obsf-resource* ((*sim1* $|_=$ *sim2*) $|_=$ (*1C* $|_=$ *1C*) ⊳ *parallel-wiring* ⊳ *ideal1* ∥ *ideal2*))
(*obsf-resource* ((*1C* $|_=$ *sim2*) $|_=$ (*1C* $|_=$ *1C*) ⊳ *parallel-wiring* ⊳ *real1* ∥ *ideal2*))
≤ *adv1*
unfolding *advantage-def ideal1*[*symmetric*] *real1*[*symmetric*] **by**(*rule sec1.adv*[*unfolded advantage-def*])(*rule WT-intro*)+

from *adv1 adv2* **show** *advantage* $\mathcal{A}$ (*obsf-resource* ((*sim1* $|_=$ *sim2*) $|_=$ (*1C* $|_=$ *1C*) ⊳ *parallel-wiring* ⊳ *ideal1* ∥ *ideal2*))
(*obsf-resource* (*parallel-wiring* ⊳ *real1* ∥ *real2*)) ≤ *adv1* + *adv2*
by(*auto simp add*: *advantage-def*)
qed
qed

theorem *parallel-constructive-security-obsf-fuse*:
assumes *1*: *constructive-security-obsf real1 ideal1 sim1* ($\mathcal{I}$*-real1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real1-rest*) ($\mathcal{I}$*-ideal1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-ideal1-rest*) ($\mathcal{I}$*-common1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common1-rest*) (*absorb* $\mathcal{A}$ (*obsf-converter* (*fused-wiring* $\odot$ *parallel-converter* 1_C (*converter-of-resource* (*sim2* $|_=$ 1_C $\rhd$ *ideal2*))))) *adv1*
(**is** *constructive-security-obsf - - - ?*$\mathcal{I}$*-real1 ?*$\mathcal{I}$*-ideal1 ?*$\mathcal{I}$*-common1 ?*$\mathcal{A}$*1 -*)
assumes *2*: *constructive-security-obsf real2 ideal2 sim2* ($\mathcal{I}$*-real2-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2-rest*) ($\mathcal{I}$*-ideal2-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-ideal2-rest*) ($\mathcal{I}$*-common2-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2-rest*) (*absorb* $\mathcal{A}$ (*obsf-converter* (*fused-wiring* $\odot$ *parallel-converter* (*converter-of-resource real1*) 1_C))) *adv2*
(**is** *constructive-security-obsf - - - ?*$\mathcal{I}$*-real2 ?*$\mathcal{I}$*-ideal2 ?*$\mathcal{I}$*-common2 ?*$\mathcal{A}$*2 -*)
shows *constructive-security-obsf* (*fused-wiring* $\rhd$ *real1* $\|$ *real2*) (*fused-wiring* $\rhd$ *ideal1* $\|$ *ideal2*)
(*parallel-wiring* $\odot$ (*sim1* $|_=$ *sim2*) $\odot$ *parallel-wiring*)
(($\mathcal{I}$*-real1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2-core*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-real1-rest* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2-rest*))
(($\mathcal{I}$*-ideal1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-ideal2-core*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-ideal1-rest* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-ideal2-rest*))
(($\mathcal{I}$*-common1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2-core*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1-rest* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2-rest*))
$\mathcal{A}$ (*adv1* + *adv2*)
proof −
interpret *sec1*: *constructive-security-obsf real1 ideal1 sim1 ?*$\mathcal{I}$*-real1 ?*$\mathcal{I}$*-ideal1 ?*$\mathcal{I}$*-common1 ?*$\mathcal{A}$*1 adv1* **by** *fact*
interpret *sec2*: *constructive-security-obsf real2 ideal2 sim2 ?*$\mathcal{I}$*-real2 ?*$\mathcal{I}$*-ideal2 ?*$\mathcal{I}$*-common2 ?*$\mathcal{A}$*2 adv2* **by** *fact*

have *aux1*: *constructive-security-aux-obsf real1 ideal1 sim1 ?*$\mathcal{I}$*-real1 ?*$\mathcal{I}$*-ideal1 ?*$\mathcal{I}$*-common1 adv1* **..**
have *aux2*: *constructive-security-aux-obsf real2 ideal2 sim2 ?*$\mathcal{I}$*-real2 ?*$\mathcal{I}$*-ideal2 ?*$\mathcal{I}$*-common2 adv2* **..**

have *sim*: *constructive-security-sim-obsf* (*parallel-wiring* $\rhd$ *real1* $\|$ *real2*) (*parallel-wiring* $\rhd$ *ideal1* $\|$ *ideal2*) (*sim1* $|_=$ *sim2*)
(*?*$\mathcal{I}$*-real1* $\oplus_{\mathcal{I}}$ *?*$\mathcal{I}$*-real2*) (*?*$\mathcal{I}$*-common1* $\oplus_{\mathcal{I}}$ *?*$\mathcal{I}$*-common2*)
(*absorb* $\mathcal{A}$ (*obsf-converter* (*parallel-wiring* $|_=$ *parallel-wiring*)))
(*adv1* + *adv2*)
if [*WT-intro*]: *exception-*$\mathcal{I}$ ((($\mathcal{I}$*-real1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2-core*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-real1-rest* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-real2-rest*)) $\oplus_{\mathcal{I}}$ (($\mathcal{I}$*-common1-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2-core*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-common1-rest* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common2-rest*))) $\vdash g$ $\mathcal{A}$ $\surd$
proof −
interpret *constructive-security-obsf*
parallel-wiring $\rhd$ *real1* $\|$ *real2*
parallel-wiring $\rhd$ *ideal1* $\|$ *ideal2*
sim1 $|_=$ *sim2*
?$\mathcal{I}$*-real1* $\oplus_{\mathcal{I}}$ *?*$\mathcal{I}$*-real2 ?*$\mathcal{I}$*-ideal1* $\oplus_{\mathcal{I}}$ *?*$\mathcal{I}$*-ideal2 ?*$\mathcal{I}$*-common1* $\oplus_{\mathcal{I}}$ *?*$\mathcal{I}$*-common2*
absorb $\mathcal{A}$ (*obsf-converter* (*parallel-wiring* $|_=$ *parallel-wiring*))
adv1 + *adv2*
apply(*rule parallel-constructive-security-obsf*)
apply(*fold absorb-comp-converter*)
apply(*rule constructive-security-obsf-absorb-cong*[*OF 1*])
apply(*rule WT-intro*)+

apply(*unfold fused-wiring-def comp-converter-assoc*)
apply(*rule obsf-comp-converter*)
apply(*rule WT-intro pfinite-intro*)+
apply(*rule constructive-security-obsf-absorb-cong*[*OF 2*])
apply(*rule WT-intro*)+
apply(*subst fused-wiring-def*)
apply(*unfold comp-converter-assoc*)
apply(*rule obsf-comp-converter*)
apply(*rule WT-intro pfinite-intro wiring-intro parallel-wiring-inverse*)+
done
show *?thesis* **..**
qed
show *?thesis*
unfolding *fused-wiring-def attach-compose*
apply(*rule constructive-security-obsf-lifting*[**where** *w-adv-ideal-inv=parallel-wiring*])
apply(*rule parallel-constructive-security-aux-obsf*[*OF aux1 aux2*])
apply(*erule sim*)
apply(*rule WT-intro pfinite-intro parallel-wiring-inverse*)+
done
qed

end
theory *Asymptotic-Security* **imports** *Concrete-Security* **begin**

8 Asymptotic security definition

locale *constructive-security-obsf′* =
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 *ℐ-real* :: *security* ⇒ (*′a*, *′b*) *ℐ*
and *ℐ-ideal* :: *security* ⇒ (*′c*, *′d*) *ℐ*
and *ℐ-common* :: *security* ⇒ (*′e*, *′f*) *ℐ*
and *𝒜* :: *security* ⇒ (*′a* + *′e*, *′b* + *′f*) *distinguisher-obsf*
assumes *constructive-security-aux-obsf*: ⋀*η*.
constructive-security-aux-obsf (*real-resource η*) (*ideal-resource η*) (*sim η*) (*ℐ-real η*) (*ℐ-ideal η*) (*ℐ-common η*) *0*
and *adv*: ⟦ ⋀*η*. *exception-ℐ* (*ℐ-real η* $\oplus_{\mathcal{I}}$ *ℐ-common η*) ⊢g *𝒜 η* √ ⟧
⟹ *negligible* (*λη*. *advantage* (*𝒜 η*) (*obsf-resource* (*sim η* $|_=$ 1_C ⊳ *ideal-resource η*)) (*obsf-resource* (*real-resource η*)))
begin

sublocale *constructive-security-aux-obsf*
real-resource η
ideal-resource η
sim η
ℐ-real η
ℐ-ideal η
ℐ-common η

0
for *η* **by**(*rule constructive-security-aux-obsf*)

lemma *constructive-security-obsf′D*:
constructive-security-obsf (*real-resource η*) (*ideal-resource η*) (*sim η*) (*ℐ-real η*) (*ℐ-ideal η*) (*ℐ-common η*) (*𝒜 η*)
(*advantage* (*𝒜 η*) (*obsf-resource* (*sim η* $|_=$ 1_C ⊳ *ideal-resource η*)) (*obsf-resource* (*real-resource η*)))
by(*rule constructive-security-obsf-refl*)

end

lemma *constructive-security-obsf′I*:
assumes ⋀*η*. *constructive-security-obsf* (*real-resource η*) (*ideal-resource η*) (*sim η*) (*ℐ-real η*) (*ℐ-ideal η*) (*ℐ-common η*) (*𝒜 η*) (*adv η*)
and (⋀*η*. *exception-ℐ* (*ℐ-real η* $\oplus_{\mathcal{I}}$ *ℐ-common η*) ⊢*g* *𝒜 η* √) ⟹ *negligible adv*
shows *constructive-security-obsf′ real-resource ideal-resource sim ℐ-real ℐ-ideal ℐ-common 𝒜*
proof −
interpret *constructive-security-obsf*
real-resource η
ideal-resource η
sim η
ℐ-real η
ℐ-ideal η
ℐ-common η
𝒜 η
adv η
for *η* **by** *fact*
show *?thesis*
proof
show *negligible* (*λη*. *advantage* (*𝒜 η*) (*obsf-resource* (*sim η* $|_=$ 1_C ⊳ *ideal-resource η*)) (*obsf-resource* (*real-resource η*)))
if ⋀*η*. *exception-ℐ* (*ℐ-real η* $\oplus_{\mathcal{I}}$ *ℐ-common η*) ⊢*g* *𝒜 η* √ **using** *assms*(*2*)[*OF that*]
by(*rule negligible-mono*)(*auto intro*!: *eventuallyI landau-o.big-mono simp add*: *advantage-nonneg adv-nonneg adv*[*OF that*])
qed(*rule WT-intro pfinite-intro order-refl*)+
qed

lemma *constructive-security-obsf′-into-constructive-security*:
assumes ⋀*𝒜* :: *security* ⇒ (*′a* + *′b*, *′c* + *′d*) *distinguisher-obsf*.
⟦ ⋀*η*. *interaction-bounded-by* (*λ-. True*) (*𝒜 η*) (*bound η*);
⋀*η*. *lossless* ⟹ *plossless-gpv* (*exception-ℐ* (*ℐ-real η* $\oplus_{\mathcal{I}}$ *ℐ-common η*)) (*𝒜 η*)
⟧
⟹ *constructive-security-obsf′ real-resource ideal-resource sim ℐ-real ℐ-ideal ℐ-common 𝒜*
and *correct*: ∃ *cnv*. ∀ *𝒟*. (∀ *η*. *ℐ-ideal η* $\oplus_{\mathcal{I}}$ *ℐ-common η* ⊢*g* *𝒟 η* √) ⟶
(∀ *η*. *interaction-any-bounded-by* (*𝒟 η*) (*bound η*)) ⟶

($\forall\,\eta$. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η)) $\longrightarrow$
($\forall\,\eta$. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η)) $\wedge$
Negligible.negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))
shows *constructive-security real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common bound lossless w*
proof
interpret *constructive-security-obsf′ real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* ‹λ-. *Done undefined*›
by(*rule assms*) *simp-all*
show $\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res real-resource* η $\surd$
and $\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*res ideal-resource* η $\surd$
and $\mathcal{I}$*-real* η, $\mathcal{I}$*-ideal* $\eta \vdash_C$ *sim* η $\surd$ **for** η **by**(*rule WT-intro*)+

show $\exists$ *cnv*. $\forall\,\mathcal{D}$. ($\forall\,\eta$. $\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*g* $\mathcal{D}$ η $\surd$) $\longrightarrow$
($\forall\,\eta$. *interaction-any-bounded-by* ($\mathcal{D}$ η) (*bound* η)) $\longrightarrow$
($\forall\,\eta$. *lossless* $\longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-ideal* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{D}$ η)) $\longrightarrow$
($\forall\,\eta$. *wiring* ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-real* η) (*cnv* η) (*w* η)) $\wedge$
Negligible.negligible ($\lambda\eta$. *advantage* ($\mathcal{D}$ η) (*ideal-resource* η) (*cnv* η $|_=$ 1_C $\rhd$ *real-resource* η))
by *fact*
next
fix $\mathcal{A}$ :: *security* $\Rightarrow$ (*′a* + *′b*, *′c* + *′d*) *distinguisher*
assume *WT-adv* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η $\vdash$*g* $\mathcal{A}$ η $\surd$
and *bound* [*interaction-bound*]: $\bigwedge\eta$. *interaction-any-bounded-by* ($\mathcal{A}$ η) (*bound* η)
and *lossless*: $\bigwedge\eta$. *lossless* $\Longrightarrow$ *plossless-gpv* ($\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η) ($\mathcal{A}$ η)
let *?*$\mathcal{A}$ = $\lambda\eta$. *obsf-distinguisher* ($\mathcal{A}$ η)
interpret *constructive-security-obsf′ real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common ?*$\mathcal{A}$
proof(*rule assms*)
show *interaction-any-bounded-by* (*?*$\mathcal{A}$ η) (*bound* η) **for** η **by**(*rule interaction-bound*)+
show *plossless-gpv* (*exception-*$\mathcal{I}$ ($\mathcal{I}$*-real* $\eta \oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* η)) (*?*$\mathcal{A}$ η) **if** *lossless* **for** η
using *WT-adv*[*of* η] *lossless that* **by**(*simp*)
qed
have *negligible* ($\lambda\eta$. *advantage* (*?*$\mathcal{A}$ η) (*obsf-resource* (*sim* η $|_=$ 1_C $\rhd$ *ideal-resource* η)) (*obsf-resource* (*real-resource* η)))
by(*rule adv*)(*rule WT-intro*)+
then show *negligible* ($\lambda\eta$. *advantage* ($\mathcal{A}$ η) (*sim* η $|_=$ 1_C $\rhd$ *ideal-resource* η) (*real-resource* η))
unfolding *advantage-obsf-distinguisher* **.**
qed

8.1 Composition theorems

theorem *constructive-security-obsf′-composability*:
fixes *real*

assumes *constructive-security-obsf′ middle ideal sim-inner* $\mathcal{I}$*-middle* $\mathcal{I}$*-inner* $\mathcal{I}$*-common* ($\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*obsf-converter* (*sim-outer* η $|_=$ 1_C)))
assumes *constructive-security-obsf′ real middle sim-outer* $\mathcal{I}$*-real* $\mathcal{I}$*-middle* $\mathcal{I}$*-common* $\mathcal{A}$
shows *constructive-security-obsf′ real ideal* ($\lambda\eta$. *sim-outer* η $\odot$ *sim-inner* η) $\mathcal{I}$*-real* $\mathcal{I}$*-inner* $\mathcal{I}$*-common* $\mathcal{A}$
proof(*rule constructive-security-obsf′I*)
let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*obsf-converter* (*sim-outer* η $|_=$ 1_C))
interpret *inner*: *constructive-security-obsf′ middle ideal sim-inner* $\mathcal{I}$*-middle* $\mathcal{I}$*-inner* $\mathcal{I}$*-common ?*$\mathcal{A}$ **by** *fact*
interpret *outer*: *constructive-security-obsf′ real middle sim-outer* $\mathcal{I}$*-real* $\mathcal{I}$*-middle* $\mathcal{I}$*-common* $\mathcal{A}$ **by** *fact*

let *?adv1* = $\lambda\eta$. *advantage* (*?*$\mathcal{A}$ η) (*obsf-resource* (*sim-inner* η $|_=$ 1_C $\rhd$ *ideal* η)) (*obsf-resource* (*middle* η))
let *?adv2* = $\lambda\eta$. *advantage* ($\mathcal{A}$ η) (*obsf-resource* (*sim-outer* η $|_=$ 1_C $\rhd$ *middle* η)) (*obsf-resource* (*real* η))
let *?adv* = $\lambda\eta$. *?adv1* η + *?adv2* η
show *constructive-security-obsf* (*real* η) (*ideal* η) (*sim-outer* η $\odot$ *sim-inner* η) ($\mathcal{I}$*-real* η) ($\mathcal{I}$*-inner* η) ($\mathcal{I}$*-common* η) ($\mathcal{A}$ η) (*?adv* η) **for** η
using *inner.constructive-security-obsf′D outer.constructive-security-obsf′D*
by(*rule constructive-security-obsf-composability*)
assume [*WT-intro*]: *exception-*$\mathcal{I}$ ($\mathcal{I}$*-real* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* η) $\vdash g$ $\mathcal{A}$ η $\surd$ **for** η
have *negligible ?adv1* **by**(*rule inner.adv*)(*rule WT-intro*)+
also have *negligible ?adv2* **by**(*rule outer.adv*)(*rule WT-intro*)+
finally (*negligible-plus*) **show** *negligible ?adv* **.**
qed

theorem *constructive-security-obsf′-lifting*:
assumes *sec*: *constructive-security-obsf′ real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* ($\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*obsf-converter* (1_C $|_=$ *conv* η)))
assumes *WT-conv* [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$*-common′* η, $\mathcal{I}$*-common* η $\vdash_C$ *conv* η $\surd$
and *pfinite* [*pfinite-intro*]: $\bigwedge\eta$. *pfinite-converter* ($\mathcal{I}$*-common′* η) ($\mathcal{I}$*-common* η) (*conv* η)
shows *constructive-security-obsf′*
($\lambda\eta$. 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′* $\mathcal{A}$
proof(*rule constructive-security-obsf′I*)
let *?*$\mathcal{A}$ = $\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*obsf-converter* (1_C $|_=$ *conv* η))
interpret *constructive-security-obsf′ real-resource ideal-resource sim* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common ?*$\mathcal{A}$ **by** *fact*
let *?adv* = $\lambda\eta$. *advantage* (*?*$\mathcal{A}$ η) (*obsf-resource* (*sim* η $|_=$ 1_C $\rhd$ *ideal-resource* η)) (*obsf-resource* (*real-resource* η))

fix η :: *security*
show *constructive-security-obsf* (1_C $|_=$ *conv* η $\rhd$ *real-resource* η) (1_C $|_=$ *conv* η $\rhd$ *ideal-resource* η) (*sim* η)
($\mathcal{I}$*-real* η) ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-common′* η) ($\mathcal{A}$ η)
(*?adv* η)

using *constructive-security-obsf.constructive-security-aux-obsf*[*OF constructive-security-obsf'D*]
constructive-security-obsf.constructive-security-sim-obsf[*OF constructive-security-obsf'D*]
by(*rule constructive-security-obsf-lifting-usr*)(*rule WT-intro pfinite-intro*)+
show *negligible ?adv* **if** [*WT-intro*]: $\bigwedge\eta$. *exception-$\mathcal{I}$* (*$\mathcal{I}$-real* $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-common'* η) $\vdash g$ $\mathcal{A}$ η $\surd$
by(*rule adv*)(*rule WT-intro*)+
qed

theorem *constructive-security-obsf'-trivial*:
fixes *res*
assumes [*WT-intro*]: $\bigwedge\eta$. $\mathcal{I}$ $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-common* η $\vdash$*res res* η $\surd$
shows *constructive-security-obsf' res res* (λ-. 1_C) $\mathcal{I}$ $\mathcal{I}$ *$\mathcal{I}$-common* $\mathcal{A}$
proof(*rule constructive-security-obsf'I*)
show *constructive-security-obsf* (*res* η) (*res* η) 1_C ($\mathcal{I}$ η) ($\mathcal{I}$ η) (*$\mathcal{I}$-common* η) ($\mathcal{A}$ η) *0* **for** η
using *assms* **by**(*rule constructive-security-obsf-trivial*)
qed *simp*

theorem *parallel-constructive-security-obsf'*:
assumes *constructive-security-obsf' real1 ideal1 sim1 $\mathcal{I}$-real1 $\mathcal{I}$-inner1 $\mathcal{I}$-common1* ($\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*obsf-converter* (*parallel-wiring* $\odot$ *parallel-converter* 1_C (*converter-of-resource* (*sim2* η $|_=$ 1_C $\triangleright$ *ideal2* η))))))
(**is** *constructive-security-obsf'* - - - - - - *?$\mathcal{A}$1*)
assumes *constructive-security-obsf' real2 ideal2 sim2 $\mathcal{I}$-real2 $\mathcal{I}$-inner2 $\mathcal{I}$-common2* ($\lambda\eta$. *absorb* ($\mathcal{A}$ η) (*obsf-converter* (*parallel-wiring* $\odot$ *parallel-converter* (*converter-of-resource* (*real1* η)) 1_C)))
(**is** *constructive-security-obsf'* - - - - - - *?$\mathcal{A}$2*)
shows *constructive-security-obsf'* ($\lambda\eta$. *parallel-wiring* $\triangleright$ *real1* η $\|$ *real2* η) ($\lambda\eta$. *parallel-wiring* $\triangleright$ *ideal1* η $\|$ *ideal2* η) ($\lambda\eta$. *sim1* η $|_=$ *sim2* η)
($\lambda\eta$. *$\mathcal{I}$-real1* $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) ($\lambda\eta$. *$\mathcal{I}$-inner1* $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-inner2* η) ($\lambda\eta$. *$\mathcal{I}$-common1* $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η) $\mathcal{A}$
proof(*rule constructive-security-obsf'I*)
interpret *sec1*: *constructive-security-obsf' real1 ideal1 sim1 $\mathcal{I}$-real1 $\mathcal{I}$-inner1 $\mathcal{I}$-common1 ?$\mathcal{A}$1* **by** *fact*
interpret *sec2*: *constructive-security-obsf' real2 ideal2 sim2 $\mathcal{I}$-real2 $\mathcal{I}$-inner2 $\mathcal{I}$-common2 ?$\mathcal{A}$2* **by** *fact*
let *?adv1* = $\lambda\eta$. *advantage* (*?$\mathcal{A}$1* η) (*obsf-resource* (*sim1* η $|_=$ 1_C $\triangleright$ *ideal1* η)) (*obsf-resource* (*real1* η))
let *?adv2* = $\lambda\eta$. *advantage* (*?$\mathcal{A}$2* η) (*obsf-resource* (*sim2* η $|_=$ 1_C $\triangleright$ *ideal2* η)) (*obsf-resource* (*real2* η))
let *?adv* = $\lambda\eta$. *?adv1* η + *?adv2* η
show *constructive-security-obsf* (*parallel-wiring* $\triangleright$ *real1* η $\|$ *real2* η) (*parallel-wiring* $\triangleright$ *ideal1* η $\|$ *ideal2* η)
(*sim1* η $|_=$ *sim2* η) (*$\mathcal{I}$-real1* $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) (*$\mathcal{I}$-inner1* $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-inner2* η) (*$\mathcal{I}$-common1* $\eta \oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η) ($\mathcal{A}$ η)
(*?adv* η) **for** η
using *sec1.constructive-security-obsf'D sec2.constructive-security-obsf'D*
by(*rule parallel-constructive-security-obsf*)

assume [*WT-intro*]: *exception-$\mathcal{I}$* ((*$\mathcal{I}$-real1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-real2* η) $\oplus_{\mathcal{I}}$ (*$\mathcal{I}$-common1* η $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-common2* η)) $\vdash g$ $\mathcal{A}$ η $\surd$ **for** η
have *negligible ?adv1* **by**(*rule sec1.adv*)(*rule WT-intro*)+
also have *negligible ?adv2* **by**(*rule sec2.adv*)(*rule WT-intro*)+
finally (*negligible-plus*) **show** *negligible ?adv* **.**
qed

end
theory *Key*
imports
../Fused-Resource
begin

9 Key specification

locale *ideal-key* =
fixes *valid-keys* :: *'key set*
begin

9.1 Data-types for Parties, State, Events, Input, and Output

datatype *party* = *Alice* | *Bob*

type-synonym *s-shell* = *party set*
datatype *'key' s-kernel* = *PState-Store* | *State-Store 'key'*
type-synonym *'key' state* = *'key' s-kernel* × *s-shell*

datatype *event* = *Event-Shell party* | *Event-Kernel*

datatype *iadv* = *Inp-Adversary*

datatype *iusr-alice* = *Inp-Alice*
datatype *iusr-bob* = *Inp-Bob*
type-synonym *iusr* = *iusr-alice* + *iusr-bob*

datatype *oadv* = *Out-Adversary*

datatype *'key' ousr-alice* = *Out-Alice 'key'*
datatype *'key' ousr-bob* = *Out-Bob 'key'*
type-synonym *'key' ousr* = *'key' ousr-alice* + *'key' ousr-bob*

9.1.1 Basic lemmas for automated handling of party sets (i.e. *s-shell*)

lemma *Alice-neq-iff* [*simp*]: *Alice* ≠ *x* ⟷ *x* = *Bob*
by(*cases x*) *simp-all*

lemma *neq-Alice-iff* [*simp*]: *x* ≠ *Alice* ⟷ *x* = *Bob*
by(*cases x*) *simp-all*

lemma *Bob-neq-iff* [*simp*]: *Bob* ≠ *x* ⟷ *x* = *Alice*
by(*cases x*) *simp-all*

lemma *neq-Bob-iff* [*simp*]: *x* ≠ *Bob* ⟷ *x* = *Alice*
by(*cases x*) *simp-all*

lemma *Alice-in-iff-nonempty*: *Alice* ∈ *A* ⟷ *A* ≠ {} **if** *Bob* ∉ *A*
using *that* **by**(*auto*)(*metis* (*full-types*) *party.exhaust*)

lemma *Bob-in-iff-nonempty*: *Bob* ∈ *A* ⟷ *A* ≠ {} **if** *Alice* ∉ *A*
using *that* **by**(*auto*)(*metis* (*full-types*) *party.exhaust*)

9.2 Defining the event handler

fun *poke* :: (*'key state*, *event*) *handler*
where
poke (*s-kernel*, *parties*) (*Event-Shell party*) =
(*if party* ∈ *parties then*
return-pmf None
else
return-spmf (*s-kernel*, *insert party parties*))
| *poke* (*PState-Store*, *s-shell*) (*Event-Kernel*) = *do* {
key ← *spmf-of-set valid-keys*;
return-spmf (*State-Store key*, *s-shell*) }
| *poke* - - = *return-pmf None*

lemma *in-set-spmf-poke*:
s' ∈ *set-spmf* (*poke s x*) ⟷
(∃ *s-kernel parties party*. *s* = (*s-kernel*, *parties*) ∧ *x* = *Event-Shell party* ∧ *party* ∉ *parties* ∧ *s'* = (*s-kernel*, *insert party parties*)) ∨
(∃ *s-shell key*. *s* = (*PState-Store*, *s-shell*) ∧ *x* = *Event-Kernel* ∧ *key* ∈ *valid-keys* ∧ *finite valid-keys* ∧ *s'* = (*State-Store key*, *s-shell*))
by(*cases* (*s*, *x*) *rule*: *poke.cases*)(*auto simp add*: *set-spmf-of-set*)

lemma *foldl-poke-invar*:
⟦ (*s-kernel'*, *parties'*) ∈ *set-spmf* (*foldl-spmf poke p events*); ∀ (*s-kernel*, *parties*)∈*set-spmf p*. *set-s-kernel s-kernel* ⊆ *valid-keys* ⟧
⟹ *set-s-kernel s-kernel'* ⊆ *valid-keys*
by(*induction events arbitrary*: *parties' rule*: *rev-induct*)
(*auto 4 3 simp add*: *split-def foldl-spmf-append in-set-spmf-poke dest*: *bspec*)

9.3 Defining the adversary interface

fun *iface-adv* :: (*'key state*, *iadv*, *oadv*) *oracle'*
where
iface-adv state - = *return-spmf* (*Out-Adversary*, *state*)

9.4 Defining the user interfaces

context
begin

private fun *iface-usr-func* :: *party* ⇒ - ⇒ - ⇒ *'inp* ⇒ (*'wrap-key* × *'key state*) *spmf*
  where
    iface-usr-func party wrap (*State-Store key*, *parties*) *inp* =
      (*if party* ∈ *parties then*
        return-spmf (*wrap key*, *State-Store key*, *parties*)
      else
        return-pmf None)
  | *iface-usr-func* - - - - = *return-pmf None*

abbreviation *iface-alice* :: (*'key state*, *iusr-alice*, *'key ousr-alice*) *oracle'*
  where
    iface-alice ≡ *iface-usr-func Alice Out-Alice*

abbreviation *iface-bob* :: (*'key state*, *iusr-bob*, *'key ousr-bob*) *oracle'*
  where
    iface-bob ≡ *iface-usr-func Bob Out-Bob*

abbreviation *iface-usr* :: (*'key state*, *iusr*, *'key ousr*) *oracle'*
  where
    iface-usr ≡ *plus-oracle iface-alice iface-bob*

lemma *in-set-iface-usr-func* [*simp*]:
  x ∈ *set-spmf* (*iface-usr-func party wrap state inp*) ⟷
  (∃ *key parties*. *state* = (*State-Store key*, *parties*) ∧ *party* ∈ *parties* ∧ *x* = (*wrap key*, *State-Store key*, *parties*))
  by(*cases* (*party*, *wrap*, *state*, *inp*) *rule*: *iface-usr-func.cases*) *auto*

end

9.5 Defining the Fuse Resource

primcorec *core* :: (*'key state*, *event*, *iadv*, *iusr*, *oadv*,*'key ousr*) *core*
  where
    cpoke core = *poke*
  | *cfunc-adv core* = *iface-adv*
  | *cfunc-usr core* = *iface-usr*

sublocale *fused-resource core* (*PState-Store*, {}) **.**

9.5.1 Lemma showing that the resulting resource is well-typed

lemma *WT-core* [*WT-intro*]:
  WT-core $\mathcal{I}$*-full* ($\mathcal{I}$*-uniform UNIV* (*Out-Alice* ‘ *valid-keys*) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*Out-Bob* ‘ *valid-keys*))

(*pred-prod* (*pred-s-kernel* (λ*key*. *key* ∈ *valid-keys*)) (λ-. *True*)) *core*
apply (*rule WT-core.intros*)
subgoal for *s e s'* **by**(*cases* (*s*, *e*) *rule*: *poke.cases*)(*auto split*: *if-split-asm simp add*: *set-spmf-of-set*)
by *auto*

lemma *WT-fuse* [*WT-intro*]:
assumes [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-rest* $\mathcal{I}$*-usr-rest* *I-rest rest*
shows ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-rest*) $\oplus_{\mathcal{I}}$ (($\mathcal{I}$*-uniform UNIV* (*Out-Alice* ' *valid-keys*) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*Out-Bob* ' *valid-keys*)) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr-rest*) ⊢*res resource rest* √
by(*rule WT-intro*)+ *simp*

end

end
theory *Channel*
imports
../*Fused-Resource*
begin

10 Channel specification

locale *ideal-channel* =
fixes

leak :: '*msg* ⇒ '*leak* **and**
editable :: *bool*
begin

10.1 Data-types for Parties, State, Events, Input, and Output

datatype *party* = *Alice* | *Bob*

type-synonym *s-shell* = *party set*
datatype '*msg*' *s-kernel* = *State-Void* | *State-Store* '*msg*' | *State-Collect* '*msg*' | *State-Collected*
type-synonym '*msg*' *state* = '*msg*' *s-kernel* × *s-shell*

datatype *event* = *Event-Shell party*

datatype *iadv-drop* = *Inp-Drop*
datatype *iadv-look* = *Inp-Look*
datatype '*msg*' *iadv-fedit* = *Inp-Fedit* '*msg*'
type-synonym '*msg*' *iadv* = *iadv-drop* + *iadv-look* + '*msg*' *iadv-fedit*

datatype '*msg*' *iusr-alice* = *Inp-Send* '*msg*'
datatype *iusr-bob* = *Inp-Recv*

type-synonym *'msg' iusr = 'msg' iusr-alice + iusr-bob*

datatype *oadv-drop = Out-Drop*
datatype *'leak' oadv-look = Out-Look 'leak'*
datatype *oadv-fedit = Out-Fedit*
type-synonym *'leak' oadv = oadv-drop + 'leak' oadv-look + oadv-fedit*

datatype *ousr-alice = Out-Send*
datatype *'msg' ousr-bob = Out-Recv 'msg'*
type-synonym *'msg' ousr = ousr-alice + 'msg' ousr-bob*

10.1.1 Basic lemmas for automated handling of party sets (i.e. *s-shell*)

lemma *Alice-neq-iff* [*simp*]: *Alice* $\neq$ *x* $\longleftrightarrow$ *x* = *Bob*
by(*cases x*) *simp-all*

lemma *neq-Alice-iff* [*simp*]: *x* $\neq$ *Alice* $\longleftrightarrow$ *x* = *Bob*
by(*cases x*) *simp-all*

lemma *Bob-neq-iff* [*simp*]: *Bob* $\neq$ *x* $\longleftrightarrow$ *x* = *Alice*
by(*cases x*) *simp-all*

lemma *neq-Bob-iff* [*simp*]: *x* $\neq$ *Bob* $\longleftrightarrow$ *x* = *Alice*
by(*cases x*) *simp-all*

lemma *Alice-in-iff-nonempty*: *Alice* $\in$ *A* $\longleftrightarrow$ *A* $\neq$ {} **if** *Bob* $\notin$ *A*
using *that* **by**(*auto*)(*metis* (*full-types*) *party.exhaust*)

lemma *Bob-in-iff-nonempty*: *Bob* $\in$ *A* $\longleftrightarrow$ *A* $\neq$ {} **if** *Alice* $\notin$ *A*
using *that* **by**(*auto*)(*metis* (*full-types*) *party.exhaust*)

10.2 Defining the event handler

fun *poke* :: (*'msg state*, *event*) *handler*
where
poke (*s-kernel*, *parties*) (*Event-Shell party*) =
(*if party* $\in$ *parties then*
return-pmf None
else
return-spmf (*s-kernel*, *insert party parties*))

lemma *poke-alt-def*:
poke = (λ(*s*, *ps*) *e*. *map-spmf* (*Pair s*) (*case e of Event-Shell party* $\Rightarrow$ *if party* $\in$ *ps then return-pmf None else return-spmf* (*insert party ps*)))
by(*simp add*: *fun-eq-iff split*: *event.split*)

10.3 Defining the adversary interfaces

fun *iface-drop* :: (*'msg state*, *iadv-drop*, *oadv-drop*) *oracle'*

where
iface-drop - - = return-pmf None

fun *iface-look* :: (*'msg state*, *iadv-look*, *'leak oadv-look*) *oracle'*
where
iface-look (*State-Store msg*, *parties*) - =
return-spmf (*Out-Look* (*leak msg*), *State-Store msg*, *parties*)
| *iface-look - - = return-pmf None*

fun *iface-fedit* :: (*'msg state*, *'msg iadv-fedit*, *oadv-fedit*) *oracle'*
where
iface-fedit (*State-Store msg*, *parties*) (*Inp-Fedit msg'*) =
(*if editable then*
return-spmf (*Out-Fedit*, *State-Collect msg'*, *parties*)
else
return-spmf (*Out-Fedit*, *State-Collect msg*, *parties*))
| *iface-fedit - - = return-pmf None*

abbreviation *iface-adv* :: (*'msg state*, *'msg iadv*, *'leak oadv*) *oracle'*
where
iface-adv ≡ *plus-oracle iface-drop* (*plus-oracle iface-look iface-fedit*)

lemma *in-set-spmf-iface-drop*: *ys'* ∈ *set-spmf* (*iface-drop s x*) ⟷ *False*
by *simp*

lemma *in-set-spmf-iface-look*: *ys'* ∈ *set-spmf* (*iface-look s x*) ⟷
(∃ *msg parties*. *s* = (*State-Store msg*, *parties*) ∧ *ys'* = (*Out-Look* (*leak msg*), *State-Store msg*, *parties*))
by(*cases* (*s*, *x*) *rule*: *iface-look.cases*) *simp-all*

lemma *in-set-spmf-iface-fedit*: *ys'* ∈ *set-spmf* (*iface-fedit s x*) ⟷
(∃ *msg parties msg'*. *s* = (*State-Store msg*, *parties*) ∧ *x* = (*Inp-Fedit msg'*) ∧
ys' = (*if editable then* (*Out-Fedit*, *State-Collect msg'*, *parties*) *else* (*Out-Fedit*, *State-Collect msg*, *parties*)))
by(*cases* (*s*, *x*) *rule*: *iface-fedit.cases*) *simp-all*

10.4 Defining the user interfaces

fun *iface-alice* :: (*'msg state*, *'msg iusr-alice*, *ousr-alice*) *oracle'*
where
iface-alice (*State-Void*, *parties*) (*Inp-Send msg*) =
(*if Alice* ∈ *parties then*
return-spmf (*Out-Send*, *State-Store msg*, *parties*)
else
return-pmf None)
| *iface-alice - - = return-pmf None*

fun *iface-bob* :: (*'msg state*, *iusr-bob*, *'msg ousr-bob*) *oracle'*
where

iface-bob (*State-Collect msg*, *parties*) - =
(*if Bob* ∈ *parties then*
return-spmf (*Out-Recv msg*, *State-Collected*, *parties*)
else
return-pmf None)
| *iface-bob* - - = *return-pmf None*

abbreviation *iface-usr* :: (*'msg state*, *'msg iusr*, *'msg ousr*) *oracle'*
where
iface-usr ≡ *plus-oracle iface-alice iface-bob*

lemma *in-set-spmf-iface-alice*: *ys'* ∈ *set-spmf* (*iface-alice s x*) ⟷
(∃ *parties msg*. *s* = (*State-Void*, *parties*) ∧ *x* = *Inp-Send msg* ∧ *Alice* ∈ *parties* ∧ *ys'* = (*Out-Send*, *State-Store msg*, *parties*))
by(*cases* (*s*, *x*) *rule*: *iface-alice.cases*) *simp-all*

lemma *in-set-spmf-iface-bob*: *ys'* ∈ *set-spmf* (*iface-bob s x*) ⟷
(∃ *msg parties*. *s* = (*State-Collect msg*, *parties*) ∧ *Bob* ∈ *parties* ∧ *ys'* = (*Out-Recv msg*, *State-Collected*, *parties*))
by(*cases* (*s*, *x*) *rule*: *iface-bob.cases*) *simp-all*

10.5 Defining the Fused Resource

primcorec *core* :: (*'msg state*, *event*, *'msg iadv*, *'msg iusr*, *'leak oadv*, *'msg ousr*) *core*
where
cpoke core = *poke*
| *cfunc-adv core* = *iface-adv*
| *cfunc-usr core* = *iface-usr*

sublocale *fused-resource core* (*State-Void*, {}) **.**

10.5.1 Lemma showing that the resulting resource is well-typed

lemma *WT-core* [*WT-intro*]:
WT-core ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*Inp-Fedit* ‘ *valid-messages*) *UNIV*)) ($\mathcal{I}$*-uniform* (*Inp-Send* ‘ *valid-messages*) *UNIV* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-uniform UNIV* (*Out-Recv* ‘ *valid-messages*)))
(*pred-prod* (*pred-s-kernel* (λ*msg*. *msg* ∈ *valid-messages*)) (λ-. *True*)) *core*
apply(*rule WT-core.intros*)
subgoal for *s e s'* **by**(*cases* (*s*, *e*) *rule*: *poke.cases*)(*auto split*: *if-split-asm*)
subgoal for *s x y s'* **by**(*cases* (*s*, *projl* (*projr x*)) *rule*: *iface-look.cases*)(*auto split*: *if-split-asm*)
subgoal for *s x y s'* **by**(*cases* (*s*, *projl x*) *rule*: *iface-alice.cases*)(*auto split*: *if-split-asm*)
done

lemma *WT-fuse* [*WT-intro*]:
assumes [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-rest* $\mathcal{I}$*-usr-rest I-rest rest*

shows $((\mathcal{I}\text{-}full \oplus_{\mathcal{I}} (\mathcal{I}\text{-}full \oplus_{\mathcal{I}} \mathcal{I}\text{-}uniform\ (Inp\text{-}Fedit\ `\ valid\text{-}messages)\ UNIV))$
$\oplus_{\mathcal{I}} \mathcal{I}\text{-}adv\text{-}rest) \oplus_{\mathcal{I}}$
$((\mathcal{I}\text{-}uniform\ (Inp\text{-}Send\ `\ valid\text{-}messages)\ UNIV \oplus_{\mathcal{I}} \mathcal{I}\text{-}uniform\ UNIV\ (Out\text{-}Recv$
$`\ valid\text{-}messages)) \oplus_{\mathcal{I}} \mathcal{I}\text{-}usr\text{-}rest) \vdash res\ resource\ rest\ \surd$
by(*rule WT-intro*)+ *simp*

end

end
theory *One-Time-Pad*
imports
Sigma-Commit-Crypto.Xor
../Asymptotic-Security
../Construction-Utility
../Specifications/Key
../Specifications/Channel
begin

11 One-time-pad construction

locale *one-time-pad* =
key: *ideal-key carrier* $\mathcal{L}$ +
auth: *ideal-channel id* :: *'msg* ⇒ *'msg False* +
sec: *ideal-channel* λ- :: *'msg. carrier* $\mathcal{L}$ *False* +
boolean-algebra $\mathcal{L}$
for
$\mathcal{L}$ :: (*'msg*, *'more*) *boolean-algebra-scheme* (**structure**) +
assumes
nempty-carrier: *carrier* $\mathcal{L} \neq \{\}$ **and**
finite-carrier: *finite* (*carrier* $\mathcal{L}$)
begin

11.1 Defining user callees

definition *enc-callee* :: *unit* ⇒ *'msg sec.iusr-alice*
⇒ (*sec.ousr-alice* × *unit*, *key.iusr-alice* + *'msg sec.iusr-alice*, *'msg key.ousr-alice*
+ *auth.ousr-alice*) *gpv*
where
enc-callee ≡ *stateless-callee* (λ*inp. case inp of sec.Inp-Send msg* ⇒
if msg ∈ *carrier* $\mathcal{L}$ *then*
Pause
(*Inl key.Inp-Alice*)
(λ*kout. case projl kout of key.Out-Alice key* ⇒
let cipher = *key* ⊕ *msg in*
Pause (*Inr* (*auth.Inp-Send cipher*)) (λ-. *Done sec.Out-Send*))
else
Fail)

definition *dec-callee* :: *unit* ⇒ *sec.iusr-bob*
⇒ (′*msg sec.ousr-bob* × *unit*, *key.iusr-bob* + *auth.iusr-bob*, ′*msg key.ousr-bob* + ′*msg auth.ousr-bob*) *gpv*
where
dec-callee ≡ *stateless-callee* (λ-.
Pause
(*Inr auth.Inp-Recv*)
(λ*cout. case cout of*
Inr (*auth.Out-Recv cipher*) ⇒
Pause
(*Inl key.Inp-Bob*)
(λ*kout. case projl kout of key.Out-Bob key* ⇒
Done (*sec.Out-Recv* (*key* ⊕ *cipher*)))
| - ⇒ *Fail*))

11.2 Defining adversary converter

type-synonym ′*msg*′ *astate* = ′*msg*′ *option*

definition *look-callee* :: ′*msg astate* ⇒ *sec.iadv-look*
⇒ (′*msg sec.oadv-look* × ′*msg astate*, *sec.iadv-look*, ′*msg set sec.oadv-look*) *gpv*
where
look-callee ≡ λ*state inp.*
Pause
sec.Inp-Look
(λ*cout. case cout of*
sec.Out-Look msg-set ⇒
(*case state of*
None ⇒ *do* {
msg ← *lift-spmf* (*spmf-of-set* (*msg-set*));
Done (*auth.Out-Look msg*, *Some msg*) }
| *Some msg* ⇒ *Done* (*auth.Out-Look msg*, *Some msg*)))

definition *sim* ::
(*key.iadv* + *auth.iadv-drop* + *auth.iadv-look* + ′*msg auth.iadv-fedit*,
key.oadv + *auth.oadv-drop* + ′*msg auth.oadv-look* + *auth.oadv-fedit*,
sec.iadv-drop + *sec.iadv-look* + ′*msg sec.iadv-fedit*,
sec.oadv-drop + ′*msg set sec.oadv-look* + *sec.oadv-fedit*) *converter*
where
sim ≡
let look-converter = *converter-of-callee look-callee None in*
ldummy-converter (λ-. *key.Out-Adversary*) (1_C $|_=$ *look-converter* $|_=$ 1_C)

11.3 Defining event-translator

type-synonym *estate* = *bool* × (*key.party* + *auth.party*) *set*

abbreviation *einit* :: *estate*
where
einit ≡ (*False*, {})

definition *sec-party-of-key-party* :: *key.party* ⇒ *sec.party*
  where
    sec-party-of-key-party ≡ *key.case-party sec.Alice sec.Bob*

abbreviation *etran-base-helper* :: *estate* ⇒ *key.party* + *auth.party* ⇒ *sec.event list*
  where
    etran-base-helper ≡ (λ(*s-flg*, *s-kap*) *item.*
      let sp-of = *case-sum sec-party-of-key-party id in*
      let se-of = (λ*chk out. if s-flg* ∧ *chk then* [*out*] *else* []) *in*
      let chk-alice = *Inl key.Alice* ∈ *s-kap* ∧ *Inr auth.Alice* ∈ *s-kap in*
      let chk-bob = *Inl key.Bob* ∈ *s-kap* ∧ *Inr auth.Bob* ∈ *s-kap in*
      sec.case-party
        (*se-of chk-alice* (*sec.Event-Shell sec.Alice*))
        (*se-of chk-bob* (*sec.Event-Shell sec.Bob*))
        (*sp-of item*))

abbreviation *etran-base* :: (*estate*, *key.party* + *auth.party*, *sec.event list*) *oracle′*
  where
    etran-base ≡ (λ(*s-flg*, *s-kap*) *item.*
      let s-kap′ = *insert item s-kap in*
      let event = *etran-base-helper* (*s-flg*, *s-kap′*) *item in*
      if item ∉ *s-kap then return-spmf* (*event*, *s-flg*, *s-kap′*) *else return-pmf None*)

fun *etran* :: (*estate*, *key.event* + *auth.event*, *sec.event list*) *oracle′*
  where
    etran state (*Inl* (*key.Event-Shell party*)) = *etran-base state* (*Inl party*)
  | *etran* (*False*, *s-kap*) (*Inl key.Event-Kernel*) =
      (*let check-alice* = *Inl key.Alice* ∈ *s-kap* ∧ *Inr auth.Alice* ∈ *s-kap in*
      let check-bob = *Inl key.Bob* ∈ *s-kap* ∧ *Inr auth.Bob* ∈ *s-kap in*
      let e-alice = *if check-alice then* [*sec.Event-Shell sec.Alice*] *else* [] *in*
      let e-bob = *if check-bob then* [*sec.Event-Shell sec.Bob*] *else* [] *in*
      return-spmf (*e-alice* @ *e-bob*, *True*, *s-kap*))
  | *etran state* (*Inr* (*auth.Event-Shell party*)) = *etran-base state* (*Inr party*)
  | *etran* - - = *return-pmf None*

11.3.1 Basic lemmas for automated handling of *sec-party-of-key-party*

lemma *sec-party-of-key-party-simps* [*simp*]:
  sec-party-of-key-party key.Alice = *sec.Alice*
  sec-party-of-key-party key.Bob = *sec.Bob*
  by(*simp-all add*: *sec-party-of-key-party-def*)

lemma *sec-party-of-key-party-eq-simps* [*simp*]:
  sec-party-of-key-party p = *sec.Alice* ⟷ *p* = *key.Alice*
  sec-party-of-key-party p = *sec.Bob* ⟷ *p* = *key.Bob*
  by(*simp-all add*: *sec-party-of-key-party-def split*: *key.party.split*)

lemma *key-case-party-collapse* [*simp*]: *key.case-party x x p = x*
by(*simp split*: *key.party.split*)

lemma *sec-case-party-collapse* [*simp*]: *sec.case-party x x p = x*
by(*simp split*: *sec.party.split*)

lemma *Alice-in-sec-party-of-key-party* [*simp*]:
sec.Alice ∈ *sec-party-of-key-party* ‘ *P* ⟷ *key.Alice* ∈ *P*
by(*auto simp add*: *sec-party-of-key-party-def split*: *key.party.splits*)

lemma *Bob-in-sec-party-of-key-party* [*simp*]:
sec.Bob ∈ *sec-party-of-key-party* ‘ *P* ⟷ *key.Bob* ∈ *P*
by(*auto simp add*: *sec-party-of-key-party-def split*: *key.party.splits*)

lemma *case-sec-party-of-key-party* [*simp*]: *sec.case-party a b* (*sec-party-of-key-party x*) = *key.case-party a b x*
by(*simp add*: *sec-party-of-key-party-def split*: *sec.party.split key.party.split*)

11.4 Defining Ideal and Real constructions

context
fixes
key-rest :: (*'key-s-rest*, *key.event*, *'key-iadv-rest*, *'key-iusr-rest*, *'key-oadv-rest*, *'key-ousr-rest*) *rest-wstate* **and**
auth-rest :: (*'auth-s-rest*, *auth.event*, *'auth-iadv-rest*, *'auth-iusr-rest*, *'auth-oadv-rest*, *'auth-ousr-rest*) *rest-wstate*
begin

definition *ideal-rest*
where
ideal-rest ≡ *translate-rest einit etran* (*parallel-rest key-rest auth-rest*)

definition *ideal-resource*
where
ideal-resource ≡ (*sim* $|_=$ 1_C) $|_=$ 1_C $|_=$ 1_C ▷ (*sec.resource ideal-rest*)

definition *real-resource*
where
real-resource ≡ *attach-c1f22-c1f22* (*CNV enc-callee* ()) (*CNV dec-callee* ()) (*key.resource key-rest*) (*auth.resource auth-rest*)

11.5 Wiring and simplifying the Ideal construction

definition *ideal-s-core'* :: ((- × *'msg astate* × -) × -) × *estate* × *'msg sec.state*
where
ideal-s-core' ≡ ((((), *None*, ()), ()), (*False*, {}), *sec.State-Void*, {})

definition *ideal-s-rest'* :: - × *'key-s-rest* × *'auth-s-rest*
where
ideal-s-rest' ≡ (((), ()), *rinit key-rest*, *rinit auth-rest*)

primcorec *ideal-core′* :: (((*unit* × - × *unit*) × *unit*) × -, -, *key.iadv* + -, -, -, -) *core*
where
cpoke ideal-core′ = (λ(*s-advusr*, *s-event*, *s-core*) *event*. *do* {
(*events*, *s-event′*) ← (*etran s-event event*);
s-core′ ← *foldl-spmf sec.poke* (*return-spmf s-core*) *events*;
return-spmf (*s-advusr*, *s-event′*, *s-core′*)
})
| *cfunc-adv ideal-core′* = (λ((*s-adv*, *s-usr*), *s-core*) *iadv*.
let handle-l = (λ-. *Done* (*Inl key.Out-Adversary*, *s-adv*)) *in*
let handle-r = (λ*qr*. *map-gpv* (*map-prod Inr id*) *id* ((1_I $\ddagger_I$ *look-callee* $\ddagger_I$ 1_I) *s-adv qr*)) *in*
map-spmf
(λ((*oadv*, *s-adv′*), *s-core′*). (*oadv*, (*s-adv′*, *s-usr*), *s-core′*))
(*exec-gpv* †*sec.iface-adv* (*case-sum handle-l handle-r iadv*) *s-core*))
| *cfunc-usr ideal-core′* = ††*sec.iface-usr*

primcorec *ideal-rest′* :: ((*unit* × *unit*) × -, -, -, -, -, -, -) *rest-scheme*
where
rinit ideal-rest′ = (((), ()), *rinit key-rest*, *rinit auth-rest*)
| *rfunc-adv ideal-rest′* = †(*parallel-eoracle* (*rfunc-adv key-rest*) (*rfunc-adv auth-rest*))
| *rfunc-usr ideal-rest′* = †(*parallel-eoracle* (*rfunc-usr key-rest*) (*rfunc-usr auth-rest*))

11.5.1 The ideal attachment lemma

lemma *attach-ideal*: *ideal-resource* = *RES* (*fused-resource.fuse ideal-core′ ideal-rest′*) (*ideal-s-core′*, *ideal-s-rest′*)
proof −

have *fact1*: *ideal-rest′* = *attach-rest* 1_I 1_I (*Pair* ((), ())) (*parallel-rest key-rest auth-rest*) (**is** *?L* = *?R*)
proof −

have *rinit ?L* = *rinit ?R*
by *simp*

moreover have *rfunc-adv ?L* = *rfunc-adv ?R*
unfolding *attach-rest-id-oracle-adv parallel-eoracle-def*
by (*simp add*: *extend-state-oracle-def*)

moreover have *rfunc-usr ?L* = *rfunc-usr ?R*
unfolding *attach-rest-id-oracle-usr parallel-eoracle-def*
by (*simp add*: *extend-state-oracle-def*)

ultimately show *?thesis*
by (*coinduction*) *blast*
qed

have *fact2*: *ideal-core'* =
(*let handle-l* = (λ*s ql. Generative-Probabilistic-Value.Done* (*Inl key.Out-Adversary*, *s*)) *in*
let handle-r = (λ*s qr. map-gpv* (*map-prod Inr id*) *id* ((1_I ‡$_I$ *look-callee* ‡$_I$ 1_I) *s qr*)) *in*
let tcore = *translate-core etran sec.core in*
attach-core (λ*s. case-sum* (*handle-l s*) (*handle-r s*)) 1_I *tcore*) (**is** *?L* = *?R*)
proof −

have *cpoke ?L* = *cpoke ?R*
by (*simp add*: *split-def map-spmf-conv-bind-spmf*)

moreover have *cfunc-adv ?L* = *cfunc-adv ?R*
unfolding *attach-core-def*
by (*simp add*: *split-def*)

moreover have *cfunc-usr ?L* = *cfunc-usr ?R*
unfolding *Let-def attach-core-id-oracle-usr*
by (*clarsimp simp add*: *extend-state-oracle-def*[*symmetric*])

ultimately show *?thesis*
by (*coinduction*) *blast*
qed

show *?thesis*
unfolding *ideal-resource-def sec.resource-def sim-def ideal-rest-def ideal-s-core'-def ideal-s-rest'-def*
apply(*simp add*: *conv-callee-parallel-id-right*[*symmetric*, **where** *s'*=()])
apply(*simp add*: *conv-callee-parallel-id-left*[*symmetric*, **where** *s*=()])
apply(*simp add*: *ldummy-converter-of-callee*)
apply(*subst fused-resource-move-translate*[*of* - *einit etran*])
apply(*simp add*: *resource-of-oracle-state-iso*)
apply(*simp add*: *iso-swapar-def split-beta ideal-rest-def*)
apply(*subst* (*1 2 3*) *converter-of-callee-id-oracle*[*symmetric*, *of* ()])
apply(*subst attach-parallel-fuse'*[**where** *f-init*=*Pair* ((), ())])
apply(*simp add*: *fact1*[*symmetric*] *fact2*[*symmetric*, *simplified Let-def*])
done
qed

11.6 Wiring and simplifying the Real construction

definition *real-s-core'* :: - × *'msg key.state* × *'msg auth.state*
where
real-s-core' ≡ (((), (), ()), (*key.PState-Store*, {}), (*auth.State-Void*, {}))

definition *real-s-rest'*
where
real-s-rest' ≡ *ideal-s-rest'*

primcorec *real-core′* :: ((*unit* × -) × -, -, -, -, -, -) *core*
  where
    cpoke real-core′ = (λ(*s-advusr*, *s-core*) *event*.
      map-spmf (*Pair s-advusr*) (*parallel-handler key.poke auth.poke s-core event*))
  | *cfunc-adv real-core′* = †(*key.iface-adv* $\ddagger_O$ *auth.iface-adv*)
  | *cfunc-usr real-core′* = (λ((*s-adv*, *s-usr*), *s-core*) *iusr*.
    let handle-req = *lsumr* ∘ *map-sum id* (*rsuml* ∘ *map-sum swap-sum id* ∘ *lsumr*) ∘ *rsuml in*
      let handle-ret = *lsumr* ∘ (*map-sum id* (*rsuml* ∘ (*map-sum swap-sum id* ∘ *lsumr*)) ∘ *rsuml*) *in*
      map-spmf
        (λ((*ousr*, *s-usr′*), *s-core′*). (*ousr*, (*s-adv*, *s-usr′*), *s-core′*))
        (*exec-gpv*
          (*key.iface-usr* $\ddagger_O$ *auth.iface-usr*)
          (*map-gpv′ id handle-req handle-ret* ((*enc-callee* $\ddagger_I$ *dec-callee*) *s-usr iusr*)) *s-core*))

definition *real-rest′*
  where
    real-rest′ ≡ *ideal-rest′*

11.6.1 The real attachment lemma

private lemma *WT-callee-real1*: (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*)) $\oplus_\mathcal{I}$ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*)) $\vdash c$
  (*key.fuse key-rest* $\ddagger_O$ *auth.fuse auth-rest*) *s* √
  apply(*rule WT-calleeI*)
  apply(*cases s*)
  apply(*case-tac call*)
   apply(*rename-tac* [!] *x*)
   apply(*case-tac* [!] *x*)
     apply(*rename-tac* [!] *y*)
     apply(*case-tac* [!] *y*)
  by(*auto simp add*: *fused-resource.fuse.simps*)

private lemma *WT-callee-real2*: ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ((($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*)) $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\vdash c$
 fused-resource.fuse (*parallel-core key.core auth.core*) (*parallel-rest key-rest auth-rest*) *s* √
  apply(*rule WT-calleeI*)
  apply(*cases s*)
  apply(*case-tac call*)
   apply(*rename-tac* [!] *x*)
   apply(*case-tac* [!] *x*)
     apply(*rename-tac* [!] *y*)
     apply(*case-tac* [!] *y*)
         apply(*rename-tac* [5] *z*)
         apply(*rename-tac* [6] *z*)
         apply(*case-tac* [5] *z*)

apply(*case-tac* [7] *z*)
by(*auto simp add*: *fused-resource.fuse.simps*)

lemma *attach-real*: *real-resource* = *RES* (*fused-resource.fuse real-core′ real-rest′*) (*real-s-core′*, *real-s-rest′*)
proof −

have *fact1*: *real-core′* = *attach-core* 1_I (*attach-wiring-right parallel-wiring*$_w$ (*enc-callee* $‡_I$ *dec-callee*))
(*parallel-core key.core auth.core*) (**is** *?L* = *?R*)
proof−

have *cpoke ?L* = *cpoke ?R*
by *simp*

moreover have *cfunc-adv ?L* = *cfunc-adv ?R*
unfolding *attach-core-id-oracle-adv*
by (*simp add*: *extend-state-oracle-def*)

moreover have *cfunc-usr ?L* = *cfunc-usr ?R*
unfolding *parallel-wiring*$_w$*-def swap-lassocr*$_w$*-def swap*$_w$*-def lassocr*$_w$*-def rassocl*$_w$*-def*
by (*simp add*: *attach-wiring-right-simps parallel2-wiring-simps comp-wiring-simps*)

ultimately show *?thesis*
by (*coinduction*) *blast*
qed

have *fact2*: *real-rest′* = *attach-rest* 1_I 1_I (*Pair* ((), ())) (*parallel-rest key-rest auth-rest*) (**is** *?L* = *?R*)
proof −
have *rinit ?L* = *rinit ?R*
unfolding *real-rest′-def ideal-rest′-def*
by *simp*

moreover have *rfunc-adv ?L* = *rfunc-adv ?R*
unfolding *real-rest′-def ideal-rest′-def attach-rest-id-oracle-adv*
by (*simp add*: *extend-state-oracle-def*)

moreover have *rfunc-usr ?L* = *rfunc-usr ?R*
unfolding *real-rest′-def ideal-rest′-def attach-rest-id-oracle-usr*
by (*simp add*: *extend-state-oracle-def*)

ultimately show *?thesis*
by (*coinduction*) *blast*
qed

show *?thesis*

unfolding *real-resource-def attach-c1f22-c1f22-def wiring-c1r22-c1r22-def key.resource-def auth.resource-def*
apply(*subst resource-of-parallel-oracle*[*symmetric*])
apply(*subst attach-compose*)
apply(*subst attach-wiring-resource-of-oracle*)
apply(*rule wiring-intro*)
apply (*rule WT-resource-of-oracle*[*OF WT-callee-real1*])
apply *simp*
subgoal
apply(*subst parallel-oracle-fuse*)
apply(*subst resource-of-oracle-state-iso*)
apply *simp*
apply(*simp add*: *parallel-state-iso-def*)
apply(*subst conv-callee-parallel*[*symmetric*])
apply(*subst eq-resource-on-UNIV-iff*[*symmetric*])
apply(*rule eq-resource-on-trans*)
apply(*rule eq-$\mathcal{I}$-attach-on$'$*)
apply (*rule WT-resource-of-oracle*[*OF WT-callee-real2*])
apply(*rule parallel-converter2-eq-$\mathcal{I}$-cong*)
apply(*rule eq-$\mathcal{I}$-converter-reflI*)
apply(*rule WT-intro*)+
apply(*rule parallel-converter2-eq-$\mathcal{I}$-cong*)
apply(*rule comp-converter-of-callee-wiring*)
apply(*rule wiring-intro*)
apply(*subst conv-callee-parallel*)
apply(*rule WT-intro*)
apply (*rule WT-converter-of-callee*[**where** $\mathcal{I}$=*$\mathcal{I}$-full* **and** $\mathcal{I}'$=*$\mathcal{I}$-full* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-full*])
apply (*rule WT-gpv-$\mathcal{I}$-mono*)
apply (*rule WT-gpv-full*)
apply (*rule $\mathcal{I}$-full-le-plus-$\mathcal{I}$*)
apply(*rule order-refl*)
apply(*rule order-refl*)
apply (*clarsimp simp add*: *enc-callee-def stateless-callee-def split*!: *sec.iusr-alice.splits key.ousr-alice.splits*)
apply (*rule WT-converter-of-callee*[**where** $\mathcal{I}$=*$\mathcal{I}$-full* **and** $\mathcal{I}'$=*$\mathcal{I}$-full* $\oplus_{\mathcal{I}}$ *$\mathcal{I}$-full*])
apply (*rule WT-gpv-$\mathcal{I}$-mono*)
apply (*rule WT-gpv-full*)
apply (*rule $\mathcal{I}$-full-le-plus-$\mathcal{I}$*)
apply(*rule order-refl*)
apply(*rule order-refl*)
apply (*clarsimp simp add*: *enc-callee-def stateless-callee-def split*!: *sec.iusr-alice.splits key.ousr-alice.splits*)
apply(*subst id-converter-eq-self*)
apply(*rule order-refl*)
apply *simp*
apply *simp*
apply(*subst eq-resource-on-UNIV-iff*)

apply(*subst* (*1 2 3*) *converter-of-callee-id-oracle*[*symmetric*, *of* ()])
apply(*subst attach-parallel-fuse'*)
apply(*simp add*: *fact1 fact2 real-s-core'-def real-s-rest'-def ideal-s-rest'-def*)
done
done
qed

11.7 Proving the trace-equivalence of simplified Ideal and Real constructions

context
begin

11.7.1 Proving the trace-equivalence of cores

private abbreviation
a-I ≡ λ(*x*, *y*). ((((), *x*, ()), ()), *y*)

private abbreviation
a-R ≡ λ*x*. (((), (), ()), *x*)

abbreviation
asm-act ≡ (λ*flg pset-sec pset-key pset-auth pset-union*.
pset-union = *pset-key* <+> *pset-auth* ∧
(*flg* ⟶ *pset-sec* = *sec-party-of-key-party* ‘ *pset-key* ∩ *pset-auth*))

private inductive *S* :: (((- × *'msg option* × -) × -) × *estate* × *'msg sec.state*) *spmf*
⇒ (- × *'msg key.state* × *'msg auth.state*) *spmf* ⇒ *bool*
where
— (Auth =a)@(Key =0)
s-0-0: *S* (*return-spmf* (*a-I* (*None*, (*False*, *s-act-ka*), *sec.State-Void*, *s-act-s*)))
(*return-spmf* (*a-R* ((*key.PState-Store*, *s-act-k*), *auth.State-Void*, *s-act-a*)))
if *asm-act False s-act-s s-act-k s-act-a s-act-ka* **and** *s-act-s* = {}
— (Auth =a)@(Key =1)
| *s-0-1*: *S* (*return-spmf* (*a-I* (*None*, (*True*, *s-act-ka*), *sec.State-Void*, *s-act*)))
(*map-spmf* (λ*key*. *a-R* ((*key.State-Store key*, *s-act-k*), *auth.State-Void*, *s-act-a*)) (*spmf-of-set* (*carrier* ℒ)))
if *asm-act True s-act s-act-k s-act-a s-act-ka*
— ../(Auth =a)@(Key =1) # wl
| *s-1-1*: *S* (*return-spmf* (*a-I* (*None*, (*True* ,*s-act-ka*), *sec.State-Store msg*, *s-act-s*)))
(*map-spmf* (λ*key*. *a-R* ((*key.State-Store key*, *s-act-k*), *auth.State-Store* (*key* ⊕ *msg*), *s-act-a*)) (*spmf-of-set* (*carrier* ℒ)))
if *asm-act True s-act-s s-act-k s-act-a s-act-ka* **and** *key.Alice* ∈ *s-act-k* **and** *auth.Alice* ∈ *s-act-a* **and** *msg* ∈ *carrier* ℒ
| *s-2-1*: *S* (*return-spmf* (*a-I* (*None*, (*True* ,*s-act-ka*), *sec.State-Collect msg*, *s-act-s*)))
(*map-spmf* (λ*key*. *a-R* ((*key.State-Store key*, *s-act-k*), *auth.State-Collect* (*key* ⊕ *msg*), *s-act-a*)) (*spmf-of-set* (*carrier* ℒ)))

if *asm-act True s-act-s s-act-k s-act-a s-act-ka* **and** *key.Alice* ∈ *s-act-k* **and** *auth.Alice* ∈ *s-act-a* **and** *msg* ∈ *carrier* $\mathcal{L}$
| *s-3-1*: *S* (*return-spmf* (*a-I* (*None*, (*True* ,*s-act-ka*), *sec.State-Collected*, *s-act-s*)))
(*map-spmf* (λ*key*. *a-R* ((*key.State-Store key*, *s-act-k*), *auth.State-Collected*, *s-act-a*)) (*spmf-of-set* (*carrier* $\mathcal{L}$)))
if *asm-act True s-act-s s-act-k s-act-a s-act-ka* **and** *s-act-k* = {*key.Alice*, *key.Bob*} **and** *s-act-a* = {*auth.Alice*, *auth.Bob*}
— ../(Auth =a)@(Key =1) # look
| *s-1′-1*: *S* (*return-spmf* (*a-I* (*Some* (*key* ⊕ *msg*), (*True* ,*s-act-ka*), *sec.State-Store msg*, *s-act-s*)))
(*return-spmf* (*a-R* ((*key.State-Store key*, *s-act-k*), *auth.State-Store* (*key* ⊕ *msg*), *s-act-a*)))
if *asm-act True s-act-s s-act-k s-act-a s-act-ka* **and** *key.Alice* ∈ *s-act-k* **and** *auth.Alice* ∈ *s-act-a* **and** *msg* ∈ *carrier* $\mathcal{L}$ **and** *key* ∈ *carrier* $\mathcal{L}$
| *s-2′-1*: *S* (*return-spmf* (*a-I* (*Some* (*key* ⊕ *msg*), (*True* ,*s-act-ka*), *sec.State-Collect msg*, *s-act-s*)))
(*return-spmf* (*a-R* ((*key.State-Store key*, *s-act-k*), *auth.State-Collect* (*key* ⊕ *msg*), *s-act-a*)))
if *asm-act True s-act-s s-act-k s-act-a s-act-ka* **and** *key.Alice* ∈ *s-act-k* **and** *auth.Alice* ∈ *s-act-a* **and** *msg* ∈ *carrier* $\mathcal{L}$ **and** *key* ∈ *carrier* $\mathcal{L}$
| *s-3′-1*: *S* (*return-spmf* (*a-I* (*Some* (*key* ⊕ *msg*), (*True* ,*s-act-ka*), *sec.State-Collected*, *s-act-s*)))
(*return-spmf* (*a-R* ((*key.State-Store key*, *s-act-k*), *auth.State-Collected*, *s-act-a*)))
if *asm-act True s-act-s s-act-k s-act-a s-act-ka* **and** *s-act-k* = {*key.Alice*, *key.Bob*} **and** *s-act-a* = {*auth.Alice*, *auth.Bob*} **and** *msg* ∈ *carrier* $\mathcal{L}$ **and** *key* ∈ *carrier* $\mathcal{L}$

private lemma *trace-eq-core*: *trace-core-eq ideal-core′ real-core′*
UNIV (*UNIV* <+> *UNIV* <+> *UNIV* <+> (*auth.Inp-Fedit* ‘ *carrier* $\mathcal{L}$)) ((*sec.Inp-Send* ‘ *carrier* $\mathcal{L}$) <+> *UNIV*)
(*return-spmf ideal-s-core′*) (*return-spmf real-s-core′*)
proof −

have *inj-xor*: ⟦*msg* ∈ *carrier* $\mathcal{L}$; *x* ∈ *carrier* $\mathcal{L}$; *y* ∈ *carrier* $\mathcal{L}$; *x* ⊕ *msg* = *y* ⊕ *msg*⟧ ⟹ *x* = *y* **for** *msg x y*
by (*metis* (*no-types*, *opaque-lifting*) *local.xor-ac*(*2*) *local.xor-left-inverse*)

note [*simp*] = *enc-callee-def dec-callee-def look-callee-def nempty-carrier finite-carrier exec-gpv-bind spmf.map-comp map-bind-spmf bind-map-spmf bind-spmf-const o-def Let-def*

show *?thesis*
apply (*rule trace-core-eq-simI-upto*[**where** *S*=*S*])
subgoal *Init-OK*
by (*simp add*: *ideal-s-core′-def real-s-core′-def S.simps*)
subgoal *POut-OK* **for** *s-i s-r query*
apply (*cases query*)
subgoal for *e-key*
apply (*cases e-key*)
subgoal for *e-shell* **by** (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*]

split: *key.party.splits*)
subgoal *e-kernel* **by** (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
done
subgoal for *e-auth*
apply (*cases e-auth*)
subgoal for *e-shell*
by (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*]
split:*auth.party.splits*)
done
done
subgoal *PState-OK* **for** *s-i s-r query*
apply (*cases query*)
subgoal for *e-key*
apply (*cases e-key*)
subgoal for *e-shell* **by** (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*]
intro!: *trace-eq-simcl.base S.intros*[*simplified*] *split*: *key.party.splits*)
subgoal *e-kernel* **by** (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*]
sec-party-of-key-party-def intro!: *trace-eq-simcl.base S.intros*[*simplified*] *split*: *key.party.splits*)

done
subgoal for *e-auth*
apply (*cases e-auth*)
subgoal for *e-shell* **by** (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*]
intro!: *trace-eq-simcl.base S.intros*[*simplified*] *split*:*auth.party.splits*)
done
done
subgoal *AOut-OK* **for** *s-i s-r query*
apply (*cases query*)
subgoal for *q-key* **by** (*erule S.cases*) *simp-all*
subgoal for *q-auth*
apply (*cases q-auth*)
subgoal for *q-auth-drop* **by** (*erule S.cases*) (*simp-all add*: *id-oracle-def*)
subgoal for *q-auth-lfe*
apply (*cases q-auth-lfe*)
subgoal for *q-auth-look*
proof (*erule S.cases*, *goal-cases*)
case (*3 s-act-s s-act-k s-act-a s-act-ka msg*) — Corresponds to *s-1-1*
then show *?case*
apply(*simp add*: *exec-gpv-extend-state-oracle exec-gpv-map-gpv-id*
exec-gpv-plus-oracle-right exec-gpv-plus-oracle-left)
apply (*subst one-time-pad*[*symmetric*, *of msg*])
apply (*simp-all add*: *xor-comm*)
apply (*rule bind-spmf-cong*[*OF HOL.refl*])
by (*simp add*: *xor-comm*)
qed *simp-all*
subgoal for *q-auth-fedit* **by** (*erule S.cases*) (*auto simp add*: *id-oracle-def*
split:*auth.iadv-fedit.split*)
done
done

done
subgoal *AState-OK* **for** *s-i s-r query*
apply (*cases query*)
subgoal for *q-key* **by** (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S.intros*[*simplified*])
subgoal for *q-auth*
apply (*cases q-auth*)
subgoal for *q-auth-drop* **by** (*erule S.cases*) (*auto simp add*: *id-oracle-def*)
subgoal for *q-auth-lfe*
apply (*cases q-auth-lfe*)
subgoal for *q-auth-look*
proof (*erule S.cases*, *goal-cases*)
case (*3 s-act-s s-act-k s-act-a s-act-ka msg*) — Corresponds to *s-1-1*
then show *?case*
apply(*simp add*: *exec-gpv-extend-state-oracle exec-gpv-map-gpv-id exec-gpv-plus-oracle-right exec-gpv-plus-oracle-left*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *clarsimp simp add*: *set-spmf-of-set inj-on-def intro*: *inj-xor*)
apply (*rule inj-xor*, *simp-all*)
apply(*subst* (*1 2 3*) *inv-into-f-f*)
by (*auto simp add*: *S.simps inj-on-def intro*: *inj-xor*)
qed (*auto intro*!: *trace-eq-simcl.base S.intros*[*simplified*])
subgoal for *q-auth-fedit* **by** (*erule S.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *id-oracle-def intro*!: *trace-eq-simcl.base S.intros*[*simplified*])
done
done
done
subgoal *UOut-OK* **for** *s-i s-r query*
apply (*cases query*)
subgoal for *q-alice*
proof (*erule S.cases*, *goal-cases*)
case (*2 s-act-s s-act-k s-act-a s-act-ka*) — Corresponds to *s-0-1*
then show *?case*
apply (*cases auth.Alice* ∈ *s-act-a*; *cases key.Alice* ∈ *s-act-k*)
apply (*simp-all add*: *stateless-callee-def split-def split*!: *auth.iusr-alice.split*)
done
qed (*simp-all add*: *stateless-callee-def split*: *auth.iusr-alice.split*)
subgoal for *q-bob*
proof (*erule S.cases*, *goal-cases*)
case (*4 s-act-s s-act-k s-act-a s-act-ka msg*) — Corresponds to *s-2-1*
then show *?case*
apply (*cases sec.Bob* ∈ *s-act-s*)
subgoal
apply (*clarsimp simp add*: *stateless-callee-def*)
apply (*simp add*: *spmf-rel-eq*[*symmetric*])
apply (*rule rel-spmf-bindI2*)
by *simp-all*
subgoal by (*cases sec.Bob* ∈ *s-act-a*) (*clarsimp simp add*: *state-*

less-callee-def)+
done
qed (*simp-all add*: *stateless-callee-def*)
done
subgoal *UState-OK* **for** *s-i s-r query*
apply (*cases query*)
subgoal for *q-alice*
proof (*erule S.cases*, *goal-cases*)
case (*2 s-act s-act-k s-act-a s-act-ka*) — Corresponds to *s-0-1*
then show *?case*
apply (*cases auth.Alice* ∈ *s-act-a*; *cases key.Alice* ∈ *s-act-k*)
subgoal
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *state-less-callee-def split-def split*!: *auth.iusr-alice.split if-splits*)
apply(*rule trace-eq-simcl.base*)
apply (*rule S.intros*(*3*)[*simplified*])
by *simp-all*
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *state-less-callee-def split-def split*: *auth.iusr-alice.split*)+
qed (*auto simp add*: *stateless-callee-def split*: *auth.iusr-alice.split-asm*)
subgoal for *q-bob*
proof (*erule S.cases*, *goal-cases*)
case (*4 s-act-s s-act-k s-act-a s-act-ka msg*) — Corresponds to *s-2-1*
then show *?case*
apply (*cases sec.Bob* ∈ *s-act-s*)
subgoal
apply (*clarsimp simp add*: *stateless-callee-def map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst map-spmf-of-set-inj-on*)
apply (*simp-all add*: *inj-on-def*)
apply (*subst map-spmf-of-set-inj-on*[*symmetric*])
apply (*simp add*: *inj-on-def*)
apply *clarsimp*
apply(*rule trace-eq-simcl.base*)
apply (*rule S.intros*(*5*)[*simplified*])
apply (*simp-all split*: *sec.party.splits*)
by *auto*
subgoal by (*clarsimp simp add*: *stateless-callee-def split*: *if-splits*)
done
next
case (*7 s-act-s s-act-k s-act-a s-act-ka msg key*) — Corresponds to *s-2′-1*
then show *?case*
apply (*cases sec.Bob* ∈ *s-act-s*)
subgoal
apply (*clarsimp simp add*: *stateless-callee-def map-spmf-conv-bind-spmf*[*symmetric*])
apply (*rule S.intros*(*8*)[*simplified*])
apply *simp-all*
by *auto*
subgoal by (*clarsimp simp add*: *stateless-callee-def split*: *if-splits*)
done

qed (*auto simp add*: *stateless-callee-def split*: *auth.iusr-alice.split-asm*)
done
done
qed

11.7.2 Proving the trace equivalence of fused cores and rests

private definition $\mathcal{I}$*-adv-core* :: (*key.iadv* + *'msg auth.iadv*, *key.oadv* + *'msg auth.oadv*) $\mathcal{I}$
where $\mathcal{I}$*-adv-core* ≡ $\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*sec.Inp-Fedit* ' (*carrier* $\mathcal{L}$)) *UNIV*))

private definition $\mathcal{I}$*-usr-core* :: (*'msg sec.iusr*, *'msg sec.ousr*) $\mathcal{I}$
where $\mathcal{I}$*-usr-core* ≡ $\mathcal{I}$*-uniform* (*sec.Inp-Send* ' (*carrier* $\mathcal{L}$)) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* *UNIV* (*sec.Out-Recv* ' *carrier* $\mathcal{L}$)

private definition *invar-ideal′* :: ((- × *'msg astate* × -) × -) × *estate* × *'msg sec.state* ⇒ *bool*
where *invar-ideal′* = *pred-prod* (*pred-prod* (*pred-prod* (λ-. *True*) (*pred-prod* (*pred-option* (λ*x*. *x* ∈ *carrier* $\mathcal{L}$)) (λ-. *True*))) (λ-. *True*)) (*pred-prod* (λ-. *True*) (*pred-prod* (*sec.pred-s-kernel* (λ*x*. *x* ∈ *carrier* $\mathcal{L}$)) (λ-. *True*)))

private definition *invar-real′* :: - × (*'msg key.s-kernel* × -) × *'msg sec.s-kernel* × - ⇒ *bool*
where *invar-real′* = *pred-prod* (λ-. *True*) (*pred-prod* (*pred-prod* (*key.pred-s-kernel* (λ*x*. *x* ∈ *carrier* $\mathcal{L}$)) (λ-. *True*)) (*pred-prod* (*sec.pred-s-kernel* (λ*x*. *x* ∈ *carrier* $\mathcal{L}$)) (λ-. *True*)))

lemma *invar-ideal-s-core′* [*simp*]: *invar-ideal′ ideal-s-core′*
by(*simp add*: *invar-ideal′-def ideal-s-core′-def*)

lemma *invar-real-s-core′* [*simp*]: *invar-real′ real-s-core′*
by(*simp add*: *invar-real′-def real-s-core′-def*)

lemma *WT-ideal-core′* [*WT-intro*]: *WT-core* $\mathcal{I}$*-adv-core* $\mathcal{I}$*-usr-core* *invar-ideal′ ideal-core′*
apply(*rule WT-core.intros*)
apply
(*auto split!*: *sum.splits option.splits if-split-asm simp add*: $\mathcal{I}$*-adv-core-def* $\mathcal{I}$*-usr-core-def exec-gpv-map-gpv-id exec-gpv-extend-state-oracle exec-gpv-plus-oracle-left exec-gpv-plus-oracle-right invar-ideal′-def sec.in-set-spmf-iface-drop sec.in-set-spmf-iface-look sec.in-set-spmf-iface-fedit sec.in-set-spmf-iface-alice sec.in-set-spmf-iface-bob id-oracle-def look-callee-def exec-gpv-bind set-spmf-of-set sec.poke-alt-def foldl-spmf-pair-right*)
done

lemma *WT-ideal-rest′* [*WT-intro*]:
assumes *WT-rest* $\mathcal{I}$*-adv-restk* $\mathcal{I}$*-usr-restk* *I-key-rest key-rest*
and *WT-rest* $\mathcal{I}$*-adv-resta* $\mathcal{I}$*-usr-resta* *I-auth-rest auth-rest*
shows *WT-rest* ($\mathcal{I}$*-adv-restk* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-resta*) ($\mathcal{I}$*-usr-restk* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr-resta*) (λ(-,

s-rest). pred-prod I-key-rest I-auth-rest s-rest) ideal-rest′
by(rule WT-rest.intros)(fastforce simp add: fused-resource.fuse.simps parallel-eoracle-def dest: WT-restD-rfunc-adv[OF assms(1)] WT-restD-rfunc-adv[OF assms(2)] WT-restD-rfunc-usr[OF assms(1)] WT-restD-rfunc-usr[OF assms(2)] simp add: assms[THEN WT-restD-rinit])+

lemma WT-real-core′ [WT-intro]: WT-core $\mathcal{I}$-adv-core $\mathcal{I}$-usr-core invar-real′ real-core′
apply(rule WT-core.intros)
apply(auto simp add: $\mathcal{I}$-adv-core-def $\mathcal{I}$-usr-core-def enc-callee-def dec-callee-def stateless-callee-def Let-def exec-gpv-extend-state-oracle exec-gpv-map-gpv′ exec-gpv-plus-oracle-left exec-gpv-plus-oracle-right
invar-real′-def in-set-spmf-parallel-handler key.in-set-spmf-poke sec.poke-alt-def auth.in-set-spmf-iface-look auth.in-set-spmf-iface-fedit
sec.in-set-spmf-iface-alice sec.in-set-spmf-iface-bob
split!: key.ousr-alice.splits key.ousr-bob.splits auth.ousr-alice.splits auth.ousr-bob.splits sum.splits if-split-asm)
done

private lemma trace-eq-sec:
fixes $\mathcal{I}$-adv-restk $\mathcal{I}$-adv-resta $\mathcal{I}$-usr-restk $\mathcal{I}$-usr-resta
defines outs-adv ≡ (UNIV <+> UNIV <+> UNIV <+> sec.Inp-Fedit ‘ carrier $\mathcal{L}$) <+> outs-$\mathcal{I}$ ($\mathcal{I}$-adv-restk $\oplus_{\mathcal{I}}$ $\mathcal{I}$-adv-resta)
and outs-usr ≡ (sec.Inp-Send ‘ carrier $\mathcal{L}$ <+> UNIV) <+> outs-$\mathcal{I}$ ($\mathcal{I}$-usr-restk $\oplus_{\mathcal{I}}$ $\mathcal{I}$-usr-resta)
assumes WT-key [WT-intro]: WT-rest $\mathcal{I}$-adv-restk $\mathcal{I}$-usr-restk I-key-rest key-rest

and WT-auth [WT-intro]: WT-rest $\mathcal{I}$-adv-resta $\mathcal{I}$-usr-resta I-auth-rest auth-rest
shows (outs-adv <+> outs-usr) $\vdash_C$ fused-resource.fuse ideal-core′ ideal-rest′ ((ideal-s-core′, ideal-s-rest′)) ≈
fused-resource.fuse real-core′ real-rest′ ((real-s-core′, real-s-rest′))
proof −
define e$\mathcal{I}$-adv-rest :: (-, - × (key.event + auth.event) list) $\mathcal{I}$
where e$\mathcal{I}$-adv-rest ≡ map-$\mathcal{I}$ id (case-sum (map-prod Inl (map Inl)) (map-prod Inr (map Inr))) (e$\mathcal{I}$ $\mathcal{I}$-adv-restk $\oplus_{\mathcal{I}}$ e$\mathcal{I}$ $\mathcal{I}$-adv-resta)
define e$\mathcal{I}$-usr-rest :: (-, - × (key.event + auth.event) list) $\mathcal{I}$
where e$\mathcal{I}$-usr-rest ≡ map-$\mathcal{I}$ id (case-sum (map-prod Inl (map Inl)) (map-prod Inr (map Inr))) (e$\mathcal{I}$ $\mathcal{I}$-usr-restk $\oplus_{\mathcal{I}}$ e$\mathcal{I}$ $\mathcal{I}$-usr-resta)

note I-defs = $\mathcal{I}$-adv-core-def $\mathcal{I}$-usr-core-def
note eI-defs = e$\mathcal{I}$-adv-rest-def e$\mathcal{I}$-usr-rest-def

have fact1[unfolded outs-plus-$\mathcal{I}$]:
trace-rest-eq ideal-rest′ ideal-rest′ (outs-$\mathcal{I}$ ($\mathcal{I}$-adv-restk $\oplus_{\mathcal{I}}$ $\mathcal{I}$-adv-resta)) (outs-$\mathcal{I}$ ($\mathcal{I}$-usr-restk $\oplus_{\mathcal{I}}$ $\mathcal{I}$-usr-resta)) s s **for** s
apply(rule rel-rest′-into-trace-rest-eq[**where** S=(=) **and** M=(=), unfolded eq-onp-def], simp-all)
apply(fold relator-eq)
apply(rule rel-rest′-mono[THEN predicate2D, rotated −1, OF HOL.refl[of ideal-rest′, folded relator-eq]])

by *auto*

have *fact2* [*unfolded eI-defs*]: *callee-invariant-on* (*callee-of-rest ideal-rest'*) (λ(-, *s-rest*). *pred-prod I-key-rest I-auth-rest s-rest*) (*e*$\mathcal{I}$*-adv-rest* $\oplus_\mathcal{I}$ *e*$\mathcal{I}$*-usr-rest*)
apply *unfold-locales*
subgoal for *s x y s'*
apply(*cases* (*snd s*, *x*) *rule*: *parallel-oracle.cases*)
apply(*auto 4 3 simp add*: *parallel-eoracle-def eI-defs split*!: *sum.splits dest*: *WT-restD*(*1,2*)[*OF WT-key*] *WT-restD*(*1,2*)[*OF WT-auth*])
done
subgoal for *s*
apply(*fastforce intro*!: *WT-calleeI simp add*: *parallel-eoracle-def eI-defs image-image dest*: *WT-restD*(*1,2*)[*OF WT-key*] *WT-restD*(*1,2*)[*OF WT-auth*] *intro*: *rev-image-eqI*)
done
done

have *fact3*[*unfolded I-defs*]: *callee-invariant-on* (*callee-of-core ideal-core'*) *invar-ideal'* ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-core*))
by(*rule WT-intro*)+

have *fact4*[*unfolded I-defs*]: *callee-invariant-on* (*callee-of-core real-core'*) *invar-real'* ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-core* $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-core*))
by(*rule WT-intro*)+

note *nempty-carrier*[*simp*]
show *?thesis* **using** *WT-key*[*THEN WT-restD-rinit*] *WT-auth*[*THEN WT-restD-rinit*]
apply (*simp add*: *real-rest'-def real-s-rest'-def assms*(*1*, *2*))
thm *fuse-trace-eq*[**where** $\mathcal{IE}$=$\mathcal{I}$*-full* **and** $\mathcal{ICA}$=$\mathcal{I}$*-adv-core* **and** $\mathcal{ICU}$=$\mathcal{I}$*-usr-core* **and** $\mathcal{IRA}$=*e*$\mathcal{I}$*-adv-rest* **and** $\mathcal{IRU}$=*e*$\mathcal{I}$*-usr-rest*,*unfolded eI-defs* $\mathcal{I}$*-adv-core-def* $\mathcal{I}$*-usr-core-def*,*simplified*]
apply (*rule fuse-trace-eq*[**where** $\mathcal{IE}$=$\mathcal{I}$*-full* **and** $\mathcal{ICA}$=$\mathcal{I}$*-adv-core* **and** $\mathcal{ICU}$=$\mathcal{I}$*-usr-core* **and** $\mathcal{IRA}$=*e*$\mathcal{I}$*-adv-rest* **and** $\mathcal{IRU}$=*e*$\mathcal{I}$*-usr-rest*
and *?IR1.0* = λ(-, *s-rest*). *pred-prod I-key-rest I-auth-rest s-rest*
and *?IR2.0* = λ(-, *s-rest*). *pred-prod I-key-rest I-auth-rest s-rest*
and *?IC1.0* = *invar-ideal'* **and** *?IC2.0*=*invar-real'*,
unfolded eI-defs $\mathcal{I}$*-adv-core-def* $\mathcal{I}$*-usr-core-def*, *simplified*])
by (*simp-all add*: *trace-eq-core fact1 fact2 fact3 fact4 ideal-s-rest'-def*)
qed

11.7.3 Simplifying the final resource by moving the interfaces from core to rest

lemma *connect*[*unfolded* $\mathcal{I}$*-adv-core-def* $\mathcal{I}$*-usr-core-def*]:
fixes $\mathcal{I}$*-adv-restk* $\mathcal{I}$*-adv-resta* $\mathcal{I}$*-usr-restk* $\mathcal{I}$*-usr-resta*
defines $\mathcal{I}$ ≡ ($\mathcal{I}$*-adv-core* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-restk* $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-resta*)) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-usr-core* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-usr-restk* $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-resta*))
assumes [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-restk* $\mathcal{I}$*-usr-restk* *I-key-rest key-rest*
and [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-resta* $\mathcal{I}$*-usr-resta* *I-auth-rest auth-rest*
and *exception-*$\mathcal{I}$ $\mathcal{I}$ ⊢*g D* √

shows *connect D* (*obsf-resource ideal-resource*) = *connect D* (*obsf-resource real-resource*)
proof −
note *I-defs* = *I-adv-core-def I-usr-core-def*

have *fact1*: *I* ⊢*res RES* (*fused-resource.fuse ideal-core′ ideal-rest′*) *s* √
if *pred-prod I-key-rest I-auth-rest* (*snd* (*snd s*)) *invar-ideal′* (*fst s*)
for *s*
unfolding *assms*(*1*)
apply(*rule callee-invariant-on.WT-resource-of-oracle*[**where** *I*=*pred-prod invar-ideal′* (λ(-, *s-rest*). *pred-prod I-key-rest I-auth-rest s-rest*)])
subgoal by(*rule fused-resource.callee-invariant-on-fuse*)(*rule WT-intro*)+
subgoal using *that* **by**(*cases s*)(*simp*)
done

have *fact2*: *I* ⊢*res RES* (*fused-resource.fuse real-core′ real-rest′*) *s* √
if *pred-prod I-key-rest I-auth-rest* (*snd* (*snd s*)) *invar-real′* (*fst s*)
for *s*
unfolding *real-rest′-def assms*(*1*)
apply(*rule callee-invariant-on.WT-resource-of-oracle*[**where** *I*=*pred-prod invar-real′* (λ(-, *s-rest*). *pred-prod I-key-rest I-auth-rest s-rest*)])
subgoal by(*rule fused-resource.callee-invariant-on-fuse*)(*rule WT-intro*)+
subgoal using *that* **by**(*cases s*)(*simp*)
done

show *?thesis*
unfolding *attach-ideal attach-real*
apply (*rule connect-cong-trace*[**where** *I*=*exception-I I*])
apply (*rule trace-eq-obsf-resourceI*, *subst trace-eq′-resource-of-oracle*)
apply (*rule trace-eq-sec*[*OF assms*(*2*) *assms*(*3*)])
subgoal by (*rule assms*(*4*))
subgoal using *WT-gpv-outs-gpv*[*OF assms*(*4*)] **by**(*simp add*: *I-defs assms*(*1*) *nempty-carrier*)
subgoal using *assms*(*2*,*3*)[*THEN WT-restD-rinit*] **by** (*intro WT-obsf-resource*)(*rule fact1*; *simp add*: *ideal-s-rest′-def*)
subgoal using *assms*(*2*,*3*)[*THEN WT-restD-rinit*] **by** (*intro WT-obsf-resource*)(*rule fact2*; *simp add*: *real-s-rest′-def ideal-s-rest′-def*)
done
qed

end

end

end

11.8 Concrete security

context *one-time-pad* **begin**

lemma *WT-enc-callee* [*WT-intro*]:
$\mathcal{I}$*-uniform* (*sec.Inp-Send* ' *carrier* $\mathcal{L}$) *UNIV*, $\mathcal{I}$*-uniform UNIV* (*key.Out-Alice* ' *carrier* $\mathcal{L}$) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*sec.Inp-Send* ' *carrier* $\mathcal{L}$) *UNIV* $\vdash_C$ *CNV enc-callee* () $\surd$
by (*rule WT-converter-of-callee*) (*auto 4 3 simp add*: *enc-callee-def stateless-callee-def image-def split*!: *key.ousr-alice.split*)

lemma *WT-dec-callee* [*WT-intro*]:
$\mathcal{I}$*-uniform UNIV* (*sec.Out-Recv* ' *carrier* $\mathcal{L}$), $\mathcal{I}$*-uniform UNIV* (*key.Out-Bob* ' *carrier* $\mathcal{L}$) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*sec.Out-Recv* ' *carrier* $\mathcal{L}$) $\vdash_C$ *CNV dec-callee* () $\surd$
by(*rule WT-converter-of-callee*)(*auto simp add*: *dec-callee-def stateless-callee-def split*!: *sec.ousr-bob.splits*)

lemma *pfinite-enc-callee* [*pfinite-intro*]:
pfinite-converter ($\mathcal{I}$*-uniform* (*sec.Inp-Send* ' *carrier* $\mathcal{L}$) *UNIV*) ($\mathcal{I}$*-uniform UNIV* (*key.Out-Alice* ' *carrier* $\mathcal{L}$) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*sec.Inp-Send* ' *carrier* $\mathcal{L}$) *UNIV*) (*CNV enc-callee* ())
apply(*rule raw-converter-invariant.pfinite-converter-of-callee*[**where** *I*=λ-. *True*])
subgoal by *unfold-locales*(*auto simp add*: *enc-callee-def stateless-callee-def*)
subgoal by(*auto simp add*: *enc-callee-def stateless-callee-def*)
subgoal by *simp*
done

lemma *pfinite-dec-callee* [*pfinite-intro*]:
pfinite-converter ($\mathcal{I}$*-uniform UNIV* (*sec.Out-Recv* ' *carrier* $\mathcal{L}$)) ($\mathcal{I}$*-uniform UNIV* (*key.Out-Bob* ' *carrier* $\mathcal{L}$) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*sec.Out-Recv* ' *carrier* $\mathcal{L}$)) (*CNV dec-callee* ())
apply(*rule raw-converter-invariant.pfinite-converter-of-callee*[**where** *I*=λ-. *True*])
subgoal by *unfold-locales*(*auto simp add*: *dec-callee-def stateless-callee-def*)
subgoal by(*auto simp add*: *dec-callee-def stateless-callee-def*)
subgoal by *simp*
done

context
fixes
key-rest :: (*'key-s-rest*, *key.event*, *'key-iadv-rest*, *'key-iusr-rest*, *'key-oadv-rest*, *'key-ousr-rest*) *rest-wstate* **and**
auth-rest :: (*'auth-s-rest*, *auth.event*, *'auth-iadv-rest*, *'auth-iusr-rest*, *'auth-oadv-rest*, *'auth-ousr-rest*) *rest-wstate* **and**
$\mathcal{I}$*-adv-restk* **and** $\mathcal{I}$*-adv-resta* **and** $\mathcal{I}$*-usr-restk* **and** $\mathcal{I}$*-usr-resta* **and** *I-key-rest* **and** *I-auth-rest*
assumes
WT-key-rest [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-restk* $\mathcal{I}$*-usr-restk* *I-key-rest key-rest* **and**
WT-auth-rest [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-resta* $\mathcal{I}$*-usr-resta* *I-auth-rest auth-rest*

begin

theorem *secure*:
defines $\mathcal{I}$*-real* ≡ (($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*sec.Inp-Fedit* ' (*carrier* $\mathcal{L}$)) *UNIV*))) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-adv-restk* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-resta*))
and $\mathcal{I}$*-common-core* ≡ $\mathcal{I}$*-uniform* (*sec.Inp-Send* ' (*carrier* $\mathcal{L}$)) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*sec.Out-Recv* ' *carrier* $\mathcal{L}$)
and $\mathcal{I}$*-common-rest* ≡ $\mathcal{I}$*-usr-restk* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr-resta*
and $\mathcal{I}$*-ideal* ≡ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*sec.Inp-Fedit* ' (*carrier* $\mathcal{L}$)) *UNIV*)) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-adv-restk* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-resta*)
shows *constructive-security-obsf* (*real-resource TYPE*(-) *TYPE*(-) *key-rest auth-rest*) (*sec.resource* (*ideal-rest key-rest auth-rest*)) (*sim* $|_=$ 1_C) $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* ($\mathcal{I}$*-common-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common-rest*) $\mathcal{A}$ *0*
proof
let *?*$\mathcal{I}$*-common* = $\mathcal{I}$*-common-core* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common-rest*
show $\mathcal{I}$*-real* $\oplus_{\mathcal{I}}$ *?*$\mathcal{I}$*-common* ⊢*res real-resource TYPE*(-) *TYPE*(-) *key-rest auth-rest* √
unfolding $\mathcal{I}$*-real-def* $\mathcal{I}$*-common-core-def* $\mathcal{I}$*-common-rest-def* *real-resource-def attach-c1f22-c1f22-def wiring-c1r22-c1r22-def fused-wiring-def*
by(*rule WT-intro* | *simp*)+

show [*WT-intro*]: $\mathcal{I}$*-ideal* $\oplus_{\mathcal{I}}$ *?*$\mathcal{I}$*-common* ⊢*res sec.resource* (*ideal-rest key-rest auth-rest*) √
unfolding $\mathcal{I}$*-common-core-def* $\mathcal{I}$*-common-rest-def* $\mathcal{I}$*-ideal-def* *ideal-rest-def*
by(*rule WT-intro*)+ *simp*

show [*WT-intro*]: $\mathcal{I}$*-real*, $\mathcal{I}$*-ideal* $\vdash_C$ *sim* $|_=$ 1_C √
unfolding $\mathcal{I}$*-real-def* $\mathcal{I}$*-ideal-def*
apply(*rule WT-intro*)+
subgoal
unfolding *sim-def Let-def look-callee-def*
apply (*fold conv-callee-parallel-id-right*[**where** *s'*=()])
apply (*fold conv-callee-parallel-id-left*[**where** *s*=()])
apply (*subst ldummy-converter-of-callee*)
apply (*rule WT-converter-of-callee*)
by (*auto simp add*: *id-oracle-def map-gpv-conv-bind*[*symmetric*] *map-lift-spmf split*: *auth.oadv-look.split option.split*)
by (*rule WT-intro*)

show *pfinite-converter* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* (*sim* $|_=$ 1_C)
unfolding $\mathcal{I}$*-real-def* $\mathcal{I}$*-ideal-def*
apply(*rule pfinite-intro*)+
subgoal
unfolding *sim-def Let-def look-callee-def*
apply (*fold conv-callee-parallel-id-right*[**where** *s'*=()])
apply (*fold conv-callee-parallel-id-left*[**where** *s*=()])
apply (*subst ldummy-converter-of-callee*)
apply(*rule raw-converter-invariant.pfinite-converter-of-callee*[**where** *I*=λ-. *True*])

subgoal
by *unfold-locales* (*auto split*!: *sum.split sec.oadv-look.split option.split simp add*: *left-gpv-map id-oracle-def intro*!: *WT-intro WT-gpv-right-gpv WT-gpv-left-gpv*)
by (*auto split*!: *sum.splits sec.oadv-look.splits option.splits*)
by (*rule pfinite-intro*)

assume *WT* [*WT-intro*]: *exception-ℐ* (*ℐ-real* $\oplus_{\mathcal{I}}$ *?ℐ-common*) ⊢*g* 𝒜 √
show *advantage* 𝒜 (*obsf-resource* ((*sim* $|_=$ *1*$_C$) $|_=$ (*1*$_C$ $|_=$ *1*$_C$) ⊳ *sec.resource* (*ideal-rest key-rest auth-rest*)))
(*obsf-resource* (*real-resource TYPE*(-) *TYPE*(-) *key-rest auth-rest*)) ≤ *0*
using *connect*[*OF WT-key-rest*, *OF WT-auth-rest*, *OF WT*[*unfolded assms*(*1*, *2*, *3*)]]
unfolding *advantage-def* **by** (*simp add*: *ideal-resource-def*)
qed *simp*

end

end

11.9 Asymptotic security

locale *one-time-pad′* =
fixes ℒ :: *security* ⇒ (*′msg*, *′more*) *boolean-algebra-scheme*
assumes *one-time-pad* [*locale-witness*]: ⋀*η*. *one-time-pad* (ℒ *η*)
begin

sublocale *one-time-pad* ℒ *η* **for** *η* **..**

definition *real-resource′* **where** *real-resource′ rest1 rest2 η* = *real-resource TYPE*(-) *TYPE*(-) *η* (*rest1 η*) (*rest2 η*)
definition *ideal-resource′* **where** *ideal-resource′ rest1 rest2 η* = *sec.resource η* (*ideal-rest* (*rest1 η*) (*rest2 η*))
definition *sim′* **where** *sim′ η* = (*sim* $|_=$ *1*$_C$)

context
fixes
key-rest :: *nat* ⇒ (*′key-s-rest*, *key.event*, *′key-iadv-rest*, *′key-iusr-rest*, *′key-oadv-rest*, *′key-ousr-rest*) *rest-wstate* **and**
auth-rest :: *nat* ⇒ (*′auth-s-rest*, *auth.event*, *′auth-iadv-rest*, *′auth-iusr-rest*, *′auth-oadv-rest*, *′auth-ousr-rest*) *rest-wstate* **and**
ℐ-adv-restk **and** *ℐ-adv-resta* **and** *ℐ-usr-restk* **and** *ℐ-usr-resta* **and** *I-key-rest* **and** *I-auth-rest*
assumes
WT-key-res: ⋀*η*. *WT-rest* (*ℐ-adv-restk η*) (*ℐ-usr-restk η*) (*I-key-rest η*) (*key-rest η*) **and**
WT-auth-rest: ⋀*η*. *WT-rest* (*ℐ-adv-resta η*) (*ℐ-usr-resta η*) (*I-auth-rest η*) (*auth-rest η*)
begin

theorem *secure'*:
defines $\mathcal{I}$*-real* ≡ λη. (($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*sec.Inp-Fedit* ‘ (*carrier* ($\mathcal{L}$ η))) *UNIV*))) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-adv-restk* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-resta* η))
and $\mathcal{I}$*-common* ≡ λη. (($\mathcal{I}$*-uniform* (*sec.Inp-Send* ‘ (*carrier* ($\mathcal{L}$ η))) *UNIV* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform UNIV* (*sec.Out-Recv* ‘ *carrier* ($\mathcal{L}$ η))) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-usr-restk* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-usr-resta* η))
and $\mathcal{I}$*-ideal* ≡ λη. ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*sec.Inp-Fedit* ‘ (*carrier* ($\mathcal{L}$ η))) *UNIV*)) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-adv-restk* η $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-adv-resta* η)
shows *constructive-security-obsf'* (*real-resource' key-rest auth-rest*) (*ideal-resource' key-rest auth-rest*) *sim'* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* $\mathcal{A}$
proof(*rule constructive-security-obsf'I*)
show *constructive-security-obsf* (*real-resource' key-rest auth-rest* η) (*ideal-resource' key-rest auth-rest* η)
(*sim'* η) ($\mathcal{I}$*-real* η) ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-common* η) ($\mathcal{A}$ η) *0* **for** η
unfolding *real-resource'-def ideal-resource'-def sim'-def* $\mathcal{I}$*-real-def* $\mathcal{I}$*-common-def* $\mathcal{I}$*-ideal-def*
by(*rule secure*)(*rule WT-key-res WT-auth-rest*)+
qed *simp*

end

end

end
theory *Diffie-Hellman-CC*
imports
Game-Based-Crypto.Diffie-Hellman
../Asymptotic-Security
../Construction-Utility
../Specifications/Key
../Specifications/Channel
begin

hide-const (**open**) *Resumption.Pause Monomorphic-Monad.Pause Monomorphic-Monad.Done*

no-notation *Sublist.parallel* (**infixl** ‹‖› *50*)
no-notation *plus-oracle* (**infix** ‹$\oplus_O$› *500*)

12 Diffie-Hellman construction

locale *diffie-hellman* =
auth: *ideal-channel id* :: *'grp* ⇒ *'grp False* +
key: *ideal-key carrier* $\mathcal{G}$ +
cyclic-group $\mathcal{G}$
for
$\mathcal{G}$ :: *'grp cyclic-group* (**structure**)
begin

12.1 Defining user callees

datatype *'grp' cstate* = *CState-Void* | *CState-Half nat* | *CState-Full nat* × *'grp'*

datatype *icnv-alice* = *Inp-Activation-Alice*
datatype *icnv-bob* = *Iact-Activation-Bob*

datatype *ocnv-alice* = *Out-Activation-Alice*
datatype *ocnv-bob* = *Out-Activation-Bob*

```
fun alice-callee :: 'grp cstate ⇒ key.iusr-alice + icnv-alice
 ⇒ (('grp key.ousr-alice + ocnv-alice) × 'grp cstate, 'grp auth.iusr-alice + auth.iusr-bob,
auth.ousr-alice + 'grp auth.ousr-bob) gpv
  where
    alice-callee CState-Void (Inr -) = do {
      x ← lift-spmf (sample-uniform (order 𝒢));
      let msg = g [^] x;
      Pause
        (Inl (auth.Inp-Send msg))
        (λrsp. case rsp of
          Inl - ⇒ Done (Inr Out-Activation-Alice, CState-Half x)
        | Inr - ⇒ Fail) }
  | alice-callee (CState-Half x) (Inl -) =
      Pause
        (Inr auth.Inp-Recv)
        (λrsp. case rsp of
          Inl - ⇒ Fail
        | Inr msg ⇒ case msg of
            auth.Out-Recv gy ⇒
              let key = gy [^] x in
              Done (Inl (key.Out-Alice key), CState-Full (x, key)))
  | alice-callee (CState-Full (x, key)) (Inl -) = Done (Inl (key.Out-Alice key),
CState-Full (x, key))
  | alice-callee - - = Fail

fun bob-callee :: 'grp cstate  ⇒ key.iusr-bob + icnv-bob
 ⇒ (('grp key.ousr-bob + ocnv-bob) × 'grp cstate, auth.iusr-bob + 'grp auth.iusr-alice,
'grp auth.ousr-bob + auth.ousr-alice) gpv
  where
    bob-callee CState-Void (Inr -) = do {
      y ← lift-spmf (sample-uniform (order 𝒢));
      let msg = g [^] y;
      Pause
        (Inr (auth.Inp-Send msg))
        (λrsp. case rsp of
          Inl - ⇒ Fail
        | Inr - ⇒ Done (Inr Out-Activation-Bob, CState-Half y)) }
  | bob-callee (CState-Half y) (Inl -) =
      Pause
        (Inl auth.Inp-Recv)
```

(λ*rsp. case rsp of*
Inl msg $\Rightarrow$ *case msg of*
auth.Out-Recv gx $\Rightarrow$
let k = *gx* [^] *y in*
Done (*Inl* (*key.Out-Bob k*), *CState-Full* (*y*, *k*))
| *Inr* - $\Rightarrow$ *Fail*)
| *bob-callee* (*CState-Full* (*y*, *key*)) (*Inl* -) = *Done* (*Inl* (*key.Out-Bob key*), *CState-Full* (*y*, *key*))
| *bob-callee* - - = *Fail*

12.2 Defining adversary callee

type-synonym *'grp' astate* = (*'grp'* × *'grp'*) *option*

type-synonym *'grp' isim* = *'grp' auth.iadv* + *'grp' auth.iadv*
datatype *osim* = *Out-Simulator*

fun *sim-callee-base* :: ((*'grp* × *'grp*) $\Rightarrow$ *'grp*) $\Rightarrow$ (*'grp astate*, *'grp auth.iadv*, *'grp auth.oadv*) *oracle'*
where
sim-callee-base - - (*Inl* -) = *return-pmf None*
| *sim-callee-base pick gpair-opt* (*Inr* (*Inl* -)) = *do* {
sample ← *do* {
x ← *sample-uniform* (*order* $\mathcal{G}$);
y ← *sample-uniform* (*order* $\mathcal{G}$);
return-spmf (**g** [^] *x*, **g** [^] *y*) };
let sample' = *case-option sample id gpair-opt*;
return-spmf (*Inr* (*Inl* (*auth.Out-Look* (*pick sample'*))), *Some sample'*) }
| *sim-callee-base* - *gpair-opt* (*Inr* (*Inr* -)) = *return-spmf* (*Inr* (*Inr auth.Out-Fedit*), *gpair-opt*)

fun *sim-callee* :: *'grp astate* $\Rightarrow$ *'grp auth.iadv* + *'grp auth.iadv*
$\Rightarrow$ ((*'grp auth.oadv* + *'grp auth.oadv*) × *'grp astate*, *key.iadv* + *'grp isim*, *key.oadv* + *osim*) *gpv*
where
sim-callee s-gpair query =
(*let handle* = (λ*gpair-pick wrap-out q-split. do* {
- ← *Pause* (*Inr query*) *Done*;
(*out*, *s-gpair'*) ← *lift-spmf* (*sim-callee-base gpair-pick s-gpair q-split*);
Done (*wrap-out out*, *s-gpair'*) }) *in*
case-sum (*handle fst Inl*) (*handle snd Inr*) *query*)

12.3 Defining event-translator

datatype *estate-base* = *EState-Void* | *EState-Store* | *EState-Collect*
type-synonym *estate* = *bool* × (*estate-base* × *auth.s-shell*) × *estate-base* × *auth.s-shell*

definition *einit* :: *estate*
where

einit ≡ (*False*, (*EState-Void*, {}), *EState-Void*, {})

definition *eleak* :: (*estate*, *key.event*, *'grp isim*, *osim*) *eoracle*
where
eleak ≡ (λ(*s-flg*, (*s-event1*, *s-shell1*), *s-event2*, *s-shell2*) *query*.
let handle-arg1 = (λ*s q*. *case* (*s*, *q*) *of* (*EState-Store*, *Some* (*Inr* (*Inr* -))) ⇒ (*True*, *EState-Collect*) | (*s'*, -) ⇒ (*False*, *s'*)) *in*
let handle-arg2 = (λ*s q D*. *case* (*s*, *q*) *of* (*EState-Store*, *Inr* -) ⇒ *D* | - ⇒ *return-pmf None*) *in*
let (*is-ch1*, *s-event1'*) = *handle-arg1 s-event1* (*case-sum Some* (λ-. *None*) *query*) *in*
let (*is-ch2*, *s-event2'*) = *handle-arg1 s-event2* (*case-sum* (λ-. *None*) *Some query*) *in*
let check-pst1 = *is-ch1* ∧ *s-event2'* ≠ *EState-Void* ∧ *auth.Bob* ∈ *s-shell1* ∧ *auth.Alice* ∈ *s-shell2 in*
let check-pst2 = *is-ch2* ∧ *s-event1'* ≠ *EState-Void* ∧ *auth.Alice* ∈ *s-shell1* ∧ *auth.Bob* ∈ *s-shell2 in*
let e-pstfix1 = *if check-pst1 then* [*key.Event-Shell key.Bob*] *else* [] *in*
let e-pstfix2 = *if check-pst2 then* [*key.Event-Shell key.Alice*] *else* [] *in*
let e-prefix = *if* ¬*s-flg then* [*key.Event-Kernel*] *else* [] *in*
let (*s-flg'*, *event*) = *if is-ch2* ∨ *is-ch1 then* (*True*, *e-prefix* @ *e-pstfix1* @ *e-pstfix2*) *else* (*s-flg*, []) *in*
let result-base = *return-spmf* ((*Out-Simulator*, *event*), *s-flg'*, (*s-event1'*, *s-shell1*), *s-event2'*, *s-shell2*) *in*
case-sum (*handle-arg2 s-event1*) (*handle-arg2 s-event2*) *query result-base*)

fun *etran-base* :: (*key.party* × *key.party* ⇒ *key.party* × *key.party*)
⇒ (*estate*, *auth.event*, *key.event list*) *oracle'*
where
etran-base mod-event (*s-flg*, (*s-event1*, *s-shell1*), *s-event2*, *s-shell2*) (*auth.Event-Shell party*) =
(*let party-dual* = *auth.case-party* (*auth.Bob*) (*auth.Alice*) *party in*
let epair = *auth.case-party prod.swap id party* (*key.Bob*, *key.Alice*) *in*
let (*s-event-eq*, *s-event-neq*) = *auth.case-party prod.swap id party* (*s-event1*, *s-event2*) *in*
let check = *party-dual* ∈ *s-shell2* ∧ *s-event-eq* = *EState-Collect* ∧ *s-event-neq* ≠ *EState-Void in*
let event = *if check then* [*key.Event-Shell* ((*fst o mod-event*) *epair*)] *else* [] *in*
let s-shell1' = *insert party s-shell1 in*
if party ∈ *s-shell1 then*
return-pmf None
else
return-spmf (*event*, *s-flg*, (*s-event1*, *s-shell1'*), *s-event2*, *s-shell2*))

fun *etran* :: (*estate*, (*icnv-alice* + *icnv-bob*) + *auth.event* + *auth.event*, *key.event list*) *oracle'*
where
etran (*s-flg*, (*EState-Void*, *s-shell1*), *s-event2*, *s-shell2*) (*Inl* (*Inl* -)) =
(*let check* = (*s-event2* = *EState-Collect* ∧ *auth.Alice* ∈ *s-shell1* ∧ *auth.Bob* ∈

s-shell2) *in*
let event = *if check then* [*key.Event-Shell key.Alice*] *else* [] *in*
let state = (*s-flg*, (*EState-Store*, *s-shell1*), *s-event2*, *s-shell2*) *in*
if auth.Alice ∈ *s-shell1 then return-spmf* (*event*, *state*) *else return-pmf None*)
| *etran* (*s-flg*, (*s-event1*, *s-shell1*), *EState-Void*, *s-shell2*) (*Inl* (*Inr* -)) =
(*let check* = (*s-event1* = *EState-Collect* ∧ *auth.Bob* ∈ *s-shell1* ∧ *auth.Alice* ∈ *s-shell2*) *in*
let event = *if check then* [*key.Event-Shell key.Bob*] *else* [] *in*
let state = (*s-flg*, (*s-event1*, *s-shell1*), *EState-Store*, *s-shell2*) *in*
if auth.Alice ∈ *s-shell2 then return-spmf* (*event*, *state*) *else return-pmf None*)
| *etran state* (*Inr query*) =
(*let handle* = (*λmod-s mod-e q. do* {
(*evt*, *state′*) ← *etran-base mod-e* (*mod-s state*) *q*;
return-spmf (*evt*, *mod-s state′*) }) *in*
case-sum (*handle id id*) (*handle* (*apsnd prod.swap*) *prod.swap*) *query*)
| *etran* - - = *return-pmf None*

12.4 Defining Ideal and Real constructions

context
fixes
auth1-rest :: (*′auth1-s-rest*, *auth.event*, *′auth1-iadv-rest*, *′auth1-iusr-rest*, *′auth1-oadv-rest*, *′auth1-ousr-rest*) *rest-wstate* **and**
auth2-rest :: (*′auth2-s-rest*, *auth.event*, *′auth2-iadv-rest*, *′auth2-iusr-rest*, *′auth2-oadv-rest*, *′auth2-ousr-rest*) *rest-wstate*
begin

primcorec *ideal-core-alt*
where
cpoke ideal-core-alt = *cpoke* (*translate-core etran key.core*)
| *cfunc-adv ideal-core-alt* = †(*cfunc-adv key.core*) $\oplus_O$ (*λ*(*se*, *sc*) *q. do* {
((*out*, *es*), *se′*) ← *eleak se q*;
sc′ ← *foldl-spmf* (*cpoke key.core*) (*return-spmf sc*) *es*;
return-spmf (*out*, *se′*, *sc′*) })
| *cfunc-usr ideal-core-alt* = *cfunc-usr* (*translate-core etran key.core*)

primcorec *ideal-rest-alt*
where
rinit ideal-rest-alt = *rinit* (*parallel-rest auth1-rest auth2-rest*)
| *rfunc-adv ideal-rest-alt* = (*λs q. map-spmf* (*apfst* (*apsnd* (*map Inr*))) (*rfunc-adv* (*parallel-rest auth1-rest auth2-rest*) *s q*))
| *rfunc-usr ideal-rest-alt* = (
let handle = *map-sum* (*λ- :: icnv-alice. Out-Activation-Alice*) (*λ- :: icnv-bob. Out-Activation-Bob*) *in*
plus-eoracle (*λs q. return-spmf* ((*handle q*, [*q*]), *s*)) (*rfunc-usr* (*parallel-rest auth1-rest auth2-rest*)))

primcorec *ideal-rest*
where

rinit ideal-rest = (*einit*, *rinit ideal-rest-alt*)
| *rfunc-adv ideal-rest* = (λ*s q*. *case q of*
Inl ql ⇒ *map-spmf* (*apfst* (*map-prod Inl id*)) (*eleak*† *s ql*)
| *Inr qr* ⇒ *map-spmf* (*apfst* (*map-prod Inr id*)) (*translate-eoracle etran* †(*rfunc-adv ideal-rest-alt*) *s qr*))
| *rfunc-usr ideal-rest* = *translate-eoracle etran* †(*rfunc-usr ideal-rest-alt*)

definition *ideal-resource*
where
ideal-resource ≡
(*let sim* = *CNV sim-callee None in*
attach ((*sim* $|_=$ 1_C) ⊙ $lassocr_C$ $|_=$ 1_C $|_=$ 1_C) (*key.resource ideal-rest*))

definition *real-resource*
where
real-resource ≡
(*let dh-wiring* = *parallel-wiring* ⊙ (*CNV alice-callee CState-Void* $|_=$ *CNV bob-callee CState-Void*) ⊙ *parallel-wiring* ⊙ (1_C $|_=$ $swap_C$) *in*
attach (((1_C $|_=$ 1_C) $|_=$ $rassocl_C$ ⊙ (*dh-wiring* $|_=$ 1_C)) ⊙ *fused-wiring*) ((*auth.resource auth1-rest*) ∥ (*auth.resource auth2-rest*)))

12.5 Wiring and simplifying the Ideal construction

abbreviation *basic-rest-sinit*
where
basic-rest-sinit ≡ (((), ()), *rinit auth1-rest*, *rinit auth2-rest*)

primcorec *basic-rest* :: ((*unit* × *unit*) × -, -, -, -, -, -, -) *rest-scheme*
where
rinit basic-rest = (*rinit auth1-rest*, *rinit auth2-rest*)
| *rfunc-adv basic-rest* = †(*parallel-eoracle* (*rfunc-adv auth1-rest*) (*rfunc-adv auth2-rest*))
| *rfunc-usr basic-rest* = †(*parallel-eoracle* (*rfunc-usr auth1-rest*) (*rfunc-usr auth2-rest*))

definition *ideal-s-core′* :: (*′grp astate* × -) × - × *′grp key.state*
where
ideal-s-core′ ≡ ((*None*, ()), *einit*, *key.PState-Store*, {})

definition *ideal-s-rest′*
where
ideal-s-rest′ ≡ *basic-rest-sinit*

primcorec *ideal-core′*
where
cpoke ideal-core′ = (λ(*s-cnv*, *s-event*, *s-core*) *event*. *do* {
(*events*, *s-event′*) ← (*etran s-event event*);
s-core′ ← *foldl-spmf key.poke* (*return-spmf s-core*) *events*;
return-spmf (*s-cnv*, *s-event′*, *s-core′*) })
| *cfunc-adv ideal-core′* = (λ((*s-sim*, -), *s-event-core*) *q*.
map-spmf

(λ((*out*, *s-sim*′), *s-event-core*′). (*out*, (*s-sim*′, ()), *s-event-core*′))
(*exec-gpv*
(†*key.iface-adv* $\oplus_O$ (λ(*se*, *sc*) *isim*. *do* {
((*out*, *es*), *se*′) ← *eleak se isim*;
sc′ ← *foldl-spmf* (*cpoke key.core*) (*return-spmf sc*) *es*;
return-spmf (*out*, *se*′, *sc*′) }))
(*sim-callee s-sim q*) *s-event-core*))
| *cfunc-usr ideal-core*′ = (λ(*s-cnv*, *s-core*) *q*.
map-spmf (λ(*out*, *s-core*′). (*out*, *s-cnv*, *s-core*′)) (†*key.iface-usr s-core q*))

primcorec *ideal-rest*′
where
rinit ideal-rest′ = *rinit basic-rest*
| *rfunc-adv ideal-rest*′ = (λ*s q*. *map-spmf* (*apfst* (*apsnd* (*map Inr*))) (*rfunc-adv basic-rest s q*))
| *rfunc-usr ideal-rest*′ = (
let handle = *map-sum* (λ- :: *icnv-alice*. *Out-Activation-Alice*) (λ- :: *icnv-bob*. *Out-Activation-Bob*) *in*
plus-eoracle (λ*s q*. *return-spmf* ((*handle q*, [*q*]), *s*)) (*rfunc-usr basic-rest*))

12.5.1 The ideal attachment lemma

context
begin

lemma *ideal-resource-shift-interface*: *key.resource ideal-rest* = *RES*
(*apply-wiring* (*rassocl*$_w$ $|_w$ (*id*, *id*)) (*fused-resource.fuse ideal-core-alt ideal-rest-alt*))

((*einit*, *key.PState-Store*, {}), *rinit ideal-rest-alt*)
proof −
have *state-iso* (*rprodl* ∘ *apfst prod.swap*, *apfst prod.swap* ∘ *lprodr*)
by(*simp add*: *state-iso-def rprodl-def lprodr-def apfst-def*; *unfold-locales*; *simp add*: *split-def*)

note *f1*= *resource-of-oracle-state-iso*[*OF this*]

have *f2*: *key.fuse ideal-rest* = *apply-state-iso* (*rprodl* ∘ *apfst prod.swap*, *apfst prod.swap* ∘ *lprodr*)
(*apply-wiring* (*rassocl*$_w$ $|_w$ (*id*, *id*)) (*fused-resource.fuse ideal-core-alt ideal-rest-alt*))
by (*rule move-simulator-interface*[*unfolded apply-wiring-state-iso-assoc*, **where** *etran*=*etran* **and** *ifunc*=*eleak* **and** *einit*=*einit* **and**
core=*key.core* **and** *rest*=*ideal-rest* **and** *core*′=*ideal-core-alt* **and** *rest*′=*ideal-rest-alt*]) *simp-all*

show *?thesis*
unfolding *key.resource-def*
by (*subst f2*, *subst f1*) *simp*
qed

private lemma *ideal-resource-alt-def*: *ideal-resource* =
(*let sim* = *CNV sim-callee None in*
let s-init = ((*einit*, *key.PState-Store*, {}), *rinit ideal-rest-alt*) *in*
attach ((*sim* $|_=$ *1*$_C$) $|_=$ *1*$_C$ $|_=$ *1*$_C$) (*RES* (*fused-resource.fuse ideal-core-alt ideal-rest-alt*) *s-init*))
proof –
note *ideal-resource-shift-interface*
moreover have *sim* = *CNV sim-callee None* $\Longrightarrow$
(*sim* $|_=$ *1*$_C$) $\odot$ *lassocr*$_C$ $|_=$ *1*$_C$ $|_=$ *1*$_C$ $\rhd$ *RES* (*apply-wiring* (*rassocl*$_w$ $|_w$ (*id*, *id*)) (*fused-resource.fuse ideal-core-alt ideal-rest-alt*)) *s* =
(*sim* $|_=$ *1*$_C$) $|_=$ *1*$_C$ $|_=$ *1*$_C$ $\rhd$ *RES* (*fused-resource.fuse ideal-core-alt ideal-rest-alt*) *s* (**is** *?L* $\Longrightarrow$ *?R*) **for** *sim s*
proof –
have *fact1*: $\mathcal{I}$*-full*, $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full* $\vdash_C$ *CNV sim-callee s* $\surd$ **for** *s*
apply(*subst WT-converter-of-callee*)
apply (*rule WT-gpv-*$\mathcal{I}$*-mono*)
apply (*rule WT-gpv-full*)
apply (*rule* $\mathcal{I}$*-full-le-plus-*$\mathcal{I}$)
apply(*rule order-refl*)
apply(*rule order-refl*)
by (*simp-all add*:)

have *fact2*: ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*)) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\vdash c$
apply-wiring (*rassocl*$_w$ $|_w$ (*id*, *id*)) (*fused-resource.fuse ideal-core-alt ideal-rest-alt*) *s* $\surd$ **for** *s*
apply (*rule WT-calleeI*)
subgoal for *call*
apply (*cases s*, *cases call*)
apply (*rename-tac* [!] *x*)
apply (*case-tac* [!] *x*)
apply (*rename-tac* [*2*] *y*)
apply (*case-tac* [*2*] *y*)
by (*auto simp add*: *apply-wiring-def rassocl*$_w$*-def parallel2-wiring-def fused-resource.fuse.simps*)
done

note [*simp*] = *spmf.map-comp map-bind-spmf bind-map-spmf bind-spmf-const o-def*

assume *?L*
then show *?R*
apply *simp*
apply (*subst* (*1 2*) *conv-callee-parallel-id-right*[*symmetric*, **where** *s'*=()])
apply(*subst eq-resource-on-UNIV-iff* [*symmetric*])
apply (*subst eq-resource-on-trans*)
apply (*rule eq-*$\mathcal{I}$*-attach-on'*)
defer
apply (*rule parallel-converter2-eq-*$\mathcal{I}$*-cong*)

apply (*rule comp-converter-of-callee-wiring*)
apply (*rule wiring-lassocr*)
apply (*subst conv-callee-parallel-id-right*)
apply(*rule WT-intro*)+
apply (*rule fact1*)
apply(*rule WT-intro*)+
apply (*rule eq-$\mathcal{I}$-converter-reflI*)
apply(*rule WT-intro*)+
defer
apply (*subst* (*1 2 3 4*) *converter-of-callee-id-oracle*[*symmetric*, **where** *s*=()])
apply (*subst conv-callee-parallel*[*symmetric*])+
apply (*subst* (*1 2*) *attach-CNV-RES*)
subgoal
apply (*rule eq-resource-on-resource-of-oracleI*[**where** *S*=(=)])
defer
apply *simp*
apply (*rule rel-funI*)+
apply (*simp add*: *prod.rel-eq eq-on-def*)
subgoal for *s′ s q′ q*
apply (*cases s*; *cases q*)
apply (*rename-tac* [!] *x*)
apply (*case-tac* [!] *x*)
apply (*rename-tac* [!] *y*)
apply (*case-tac* [*4*] *y*)
apply (*auto simp add*: *apply-wiring-def parallel2-wiring-def attach-wiring-right-def*
rassocl$_w$*-def lassocr*$_w$*-def map-fun-def map-prod-def split-def*)
subgoal for *s-flg* - - - - - - - - - *q*
apply (*case-tac* (*s-flg*, *q*) *rule*: *sim-callee.cases*)
apply (*simp-all add*: *split-def split*!: *sum.split if-splits cong*: *if-cong*)
by (*rule rel-spmf-bindI′*[**where** *A*=(=)], *simp*, *clarsimp split*!: *sum.split if-splits*
simp add: *split-def map-gpv-conv-bind*[*symmetric*] *map-lift-spmf map′-lift-spmf*)+
by (*simp add*: *spmf-rel-eq map-fun-def id-oracle-def split-def*;
rule bind-spmf-cong[*OF refl*], *clarsimp split*!: *sum.split if-splits*
simp add: *split-def map-gpv-conv-bind*[*symmetric*] *map-lift-spmf map′-lift-spmf*)+
done
apply *simp*
apply (*rule WT-resource-of-oracle*[*OF fact2*])
by *simp*
qed

ultimately show *?thesis*
unfolding *ideal-resource-def* **by** *simp*
qed

lemma *attach-ideal*: *ideal-resource* = *RES* (*fused-resource.fuse ideal-core′ ideal-rest′*)

(*ideal-s-core′*, *ideal-s-rest′*)
proof −

have *fact1*: *ideal-rest′* = *attach-rest* 1_I 1_I *id ideal-rest-alt* (**is** *?L* = *?R*)
proof −
note [*simp*] = *spmf.map-comp map-bind-spmf bind-map-spmf bind-spmf-const o-def*

have *rinit ?L* = *rinit ?R*
by *simp*

moreover have *rfunc-adv ?L* = *rfunc-adv ?R*
unfolding *attach-rest-id-oracle-adv*
by (*simp add*: *extend-state-oracle-def split-def map-spmf-conv-bind-spmf*)

moreover have *rfunc-usr ?L* = *rfunc-usr ?R*
unfolding *attach-rest-id-oracle-usr*
apply (*rule ext*)+
subgoal for *s q* **by** (*cases q*) (*simp-all add*: *split-def extend-state-oracle-def plus-eoracle-def*)
done

ultimately show *?thesis*
by (*coinduction*) *simp*
qed

have *fact2*: *ideal-core′* = *attach-core sim-callee* 1_I *ideal-core-alt* (**is** *?L* = *?R*)
proof −

have *cpoke ?L* = *cpoke ?R*
by (*simp add*: *split-def map-spmf-conv-bind-spmf*)

moreover have *cfunc-adv ?L* = *cfunc-adv ?R*
unfolding *attach-core-def*
by (*simp add*: *split-def*)

moreover have *cfunc-usr ?L* = *cfunc-usr ?R*
unfolding *attach-core-id-oracle-usr*
by *simp*

ultimately show *?thesis*
by (*coinduction*) *simp*
qed

show *?thesis*
unfolding *ideal-resource-alt-def Let-def*
apply(*subst* (*1 2 3*) *converter-of-callee-id-oracle*[*symmetric*, *of* ()])
apply(*subst attach-parallel-fuse′*)
by (*simp add*: *fact1 fact2 ideal-s-core′-def ideal-s-rest′-def*)

qed

end

12.6 Wiring and simplifying the Real construction

definition *real-s-core′* :: (- × ′*grp cstate* × ′*grp cstate*) × ′*grp auth.state* × ′*grp auth.state*
where
real-s-core′ ≡ (((), *CState-Void*, *CState-Void*), (*auth.State-Void*, {}), (*auth.State-Void*, {}))

definition *real-s-rest′*
where
real-s-rest′ ≡ *basic-rest-sinit*

primcorec *real-core′* :: ((*unit* × -) × -, -, -, -, -, -) *core*
where
cpoke real-core′ = (λ(*s-advusr*, *s-core*) *event*.
map-spmf (*Pair s-advusr*) (*parallel-handler auth.poke auth.poke s-core event*))
| *cfunc-adv real-core′* = †(*auth.iface-adv* $\ddagger_O$ *auth.iface-adv*)
| *cfunc-usr real-core′* = (λ((*s-adv*, *s-usr*), *s-core*) *iusr*.
let handle-req = *lsumr* ∘ *map-sum id* (*rsuml* ∘ *map-sum swap-sum id* ∘ *lsumr*) ∘ *rsuml in*
let handle-ret = *lsumr* ∘ (*map-sum id* (*rsuml* ∘ (*map-sum swap-sum id* ∘ *lsumr*)) ∘ *rsuml*) ∘ *map-sum id swap-sum in*
let handle-inp = *map-sum id swap-sum* ∘ (*lsumr* ∘ *map-sum id* (*rsuml* ∘ *map-sum swap-sum id* ∘ *lsumr*) ∘ *rsuml*) *in*
let handle-out = *apfst* (*lsumr* ∘ (*map-sum id* (*rsuml* ∘ (*map-sum swap-sum id* ∘ *lsumr*)) ∘ *rsuml*)) *in*
map-spmf
(λ((*ousr*, *s-usr′*), *s-core′*). (*ousr*, (*s-adv*, *s-usr′*), *s-core′*))
(*exec-gpv*
(*auth.iface-usr* $\ddagger_O$ *auth.iface-usr*)
(*map-gpv′*
handle-out handle-inp handle-ret
((*alice-callee* $\ddagger_I$ *bob-callee*) *s-usr* (*handle-req iusr*)))
s-core))

definition *real-rest′* :: ((*unit* × *unit*) × -, -, -, -, -, -, -) *rest-scheme*
where
real-rest′ ≡ *basic-rest*

12.6.1 The real attachment lemma

lemma *attach-real*: *real-resource* = 1_C $|_=$ $rassocl_C$ ⊳ *RES* (*fused-resource.fuse real-core′ real-rest′*) (*real-s-core′*, *real-s-rest′*)
proof –

have *att-core*: *real-core′* = *attach-core* 1_I

($attach\text{-}wiring\ parallel\text{-}wiring_w$
($attach\text{-}wiring\text{-}right\ (parallel\text{-}wiring_w \circ_w (id,\ id)\ |_w\ swap_w)\ (alice\text{-}callee$
$\ddagger_I\ bob\text{-}callee)))$
($parallel\text{-}core\ auth.core\ auth.core$) (**is** $?L = ?R$)
proof −

have *cpoke ?L = cpoke ?R*
by *simp*

moreover have *cfunc-adv ?L = cfunc-adv ?R*
unfolding *attach-core-id-oracle-adv*
by (*simp add*: *extend-state-oracle-def*)

moreover have *cfunc-usr ?L = cfunc-usr ?R*
unfolding $parallel\text{-}wiring_w\text{-}def\ swap\text{-}lassocr_w\text{-}def\ swap_w\text{-}def\ lassocr_w\text{-}def$
$rassocl_w\text{-}def$
apply (*simp add*: *parallel2-wiring-simps comp-wiring-simps*)
apply (*simp add*: *attach-wiring-simps attach-wiring-right-simps*)
by (*simp add*: *map-gpv-conv-map-gpv′ map-gpv′-comp apfst-def*)

ultimately show *?thesis*
by (*coinduction*) *blast*
qed

have *att-rest*: $real\text{-}rest' = attach\text{-}rest\ 1_I\ 1_I\ id\ (parallel\text{-}rest\ auth1\text{-}rest\ auth2\text{-}rest)$
(**is** $?L = ?R$)
proof −
have *rinit ?L = rinit ?R*
unfolding *real-rest′-def*
by *simp*

moreover have *rfunc-adv ?L = rfunc-adv ?R*
unfolding *real-rest′-def attach-rest-id-oracle-adv*
by (*simp add*: *extend-state-oracle-def*)

moreover have *rfunc-usr ?L = rfunc-usr ?R*
unfolding *real-rest′-def attach-rest-id-oracle-usr*
by (*simp add*: *extend-state-oracle-def*)

ultimately show *?thesis*
by (*coinduction*) *blast*
qed

have *fact1*:
$(\mathcal{I}\text{-}full \oplus_{\mathcal{I}} \mathcal{I}\text{-}full) \oplus_{\mathcal{I}} (\mathcal{I}\text{-}full \oplus_{\mathcal{I}} \mathcal{I}\text{-}full),\ (\mathcal{I}\text{-}full \oplus_{\mathcal{I}} \mathcal{I}\text{-}full) \oplus_{\mathcal{I}} (\mathcal{I}\text{-}full \oplus_{\mathcal{I}} \mathcal{I}\text{-}full)$
$\vdash_C$
$CNV\ (alice\text{-}callee \ddagger_I bob\text{-}callee)\ (CState\text{-}Void,\ CState\text{-}Void)\ \surd$
apply(*subst conv-callee-parallel*)
apply(*rule WT-intro*)

apply (*rule WT-converter-of-callee*[**where** $\mathcal{I}$=$\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full* **and** $\mathcal{I}'$=$\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*])
apply (*rule WT-gpv-*$\mathcal{I}$*-mono*)
apply (*rule WT-gpv-full*)
apply (*rule* $\mathcal{I}$*-full-le-plus-*$\mathcal{I}$)
apply(*rule order-refl*)
apply(*rule order-refl*)
subgoal for *s q*
apply (*cases s*; *cases q*)
apply (*auto simp add*: *Let-def split*!: *cstate.splits if-splits auth.ousr-bob.splits*)
by (*metis auth.ousr-bob.exhaust range-eqI*)
apply (*rule WT-converter-of-callee*[**where** $\mathcal{I}$=$\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full* **and** $\mathcal{I}'$=$\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*])
apply (*rule WT-gpv-*$\mathcal{I}$*-mono*)
apply (*rule WT-gpv-full*)
apply (*rule* $\mathcal{I}$*-full-le-plus-*$\mathcal{I}$)
apply(*rule order-refl*)
apply(*rule order-refl*)
subgoal for *s q*
apply (*cases s*; *cases q*)
apply (*auto simp add*: *Let-def split*!: *cstate.splits if-splits auth.ousr-bob.splits*)
by (*metis auth.ousr-bob.exhaust range-eqI*)
done

have *fact2*:
($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*),($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\vdash_C$
CNV (*alice-callee* $\ddagger_I$ *bob-callee*) (*CState-Void*, *CState-Void*) $\odot$ *parallel-wiring* $\odot$ (1_C $|_=$ $swap_C$) $\surd$
apply(*rule WT-intro*)
apply (*rule fact1*)
apply(*rule WT-intro*)+
done

have *fact3*:
($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*),($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\vdash_C$
CNV (*alice-callee* $\ddagger_I$ *bob-callee*) (*CState-Void*, *CState-Void*) $\odot$ *parallel-wiring* $\odot$ (1_C $|_=$ $swap_C$) $\sim$
CNV (*attach-wiring-right* ($parallel\text{-}wiring_w$ $\circ_w$ (*id*, *id*) $|_w$ $swap_w$) (*alice-callee* $\ddagger_I$ *bob-callee*)) (*CState-Void*, *CState-Void*)
apply (*rule comp-converter-of-callee-wiring*)
apply(*rule wiring-intro*)+
apply(*rule fact1*)
done

have *fact4*:
($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*),($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*)

$\vdash_C$

parallel-wiring $\odot$ *CNV* (*alice-callee* $\ddagger_I$ *bob-callee*) (*CState-Void*, *CState-Void*) $\odot$ *parallel-wiring* $\odot$ (1_C $|_=$ $swap_C$) $\sim$

CNV (*attach-wiring* $parallel\text{-}wiring_w$ (*attach-wiring-right* ($parallel\text{-}wiring_w$ $\circ_w$ (*id*, *id*) $|_w$ $swap_w$) (*alice-callee* $\ddagger_I$ *bob-callee*))) (*CState-Void*, *CState-Void*)

apply (*rule eq-$\mathcal{I}$-converter-trans*)
apply (*rule eq-$\mathcal{I}$-comp-cong*)
apply(*rule eq-$\mathcal{I}$-converter-reflI*)
apply(*rule WT-intro*)
apply (*rule fact3*)
apply (*rule comp-wiring-converter-of-callee*)
apply (*rule wiring-intro*)
apply (*subst eq-$\mathcal{I}$-converterD-WT*[*OF fact3*, *simplified fact2*, *symmetric*])
by *blast*

show *?thesis*
unfolding *real-resource-def auth.resource-def*
apply (*subst resource-of-parallel-oracle*[*symmetric*])
apply (*subst attach-compose*)
apply(*subst attach-wiring-resource-of-oracle*)
apply (*rule wiring-intro*)
apply (*rule WT-resource-of-oracle*[**where** $\mathcal{I}$=(($\mathcal{I}$-*full* $\oplus_\mathcal{I}$ $\mathcal{I}$-*full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$-*full* $\oplus_\mathcal{I}$ $\mathcal{I}$-*full*)) $\oplus_\mathcal{I}$ (($\mathcal{I}$-*full* $\oplus_\mathcal{I}$ $\mathcal{I}$-*full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$-*full* $\oplus_\mathcal{I}$ $\mathcal{I}$-*full*))])
subgoal for - *s*
apply (*rule WT-calleeI*)
apply (*cases s*)
apply(*case-tac call*)
apply(*rename-tac* [!] *x*)
apply(*case-tac* [!] *x*)
apply(*rename-tac* [!] *y*)
apply(*case-tac* [!] *y*)
apply(*auto simp add*: *fused-resource.fuse.simps*)
done
apply *simp*
subgoal
apply (*subst parallel-oracle-fuse*)
apply(*subst resource-of-oracle-state-iso*)
apply *simp*
apply(*simp add*: *parallel-state-iso-def*)
apply(*subst parallel-converter2-comp2-out*)
apply(*subst conv-callee-parallel*[*symmetric*])
apply(*subst eq-resource-on-UNIV-iff*[*symmetric*])
apply(*rule eq-resource-on-trans*)
apply(*rule eq-$\mathcal{I}$-attach-on'*)
prefer *2*
apply (*rule eq-$\mathcal{I}$-comp-cong*)
apply(*rule eq-$\mathcal{I}$-converter-reflI*)
apply(*rule WT-intro*)+
apply(*rule parallel-converter2-eq-$\mathcal{I}$-cong*)

apply(*rule eq-$\mathcal{I}$-converter-reflI*)
apply(*rule WT-intro*)+
apply(*rule parallel-converter2-eq-$\mathcal{I}$-cong*)
prefer *2*
apply(*rule eq-$\mathcal{I}$-converter-reflI*)
apply(*rule WT-intro*)+
apply(*rule fact4*)
prefer *3*
apply(*subst attach-compose*)
apply(*fold converter-of-callee-id-oracle*[**where** *s*=()])
apply(*subst attach-parallel-fuse*′[**where** *f-init*=*id*])
apply(*unfold converter-of-callee-id-oracle*)
apply(*subst eq-resource-on-UNIV-iff*)
subgoal by (*simp add*: *att-core*[*symmetric*] *att-rest*[*symmetric*] *real-s-core*′*-def real-s-rest*′*-def*)
apply (*rule WT-resource-of-oracle*[**where** $\mathcal{I}$=($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) $\oplus_{\mathcal{I}}$ ((($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*)) $\oplus_{\mathcal{I}}$ ($\mathcal{I}$-*full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$-*full*))])
subgoal for *s*
apply (*rule WT-calleeI*)
apply (*cases s*)
apply(*case-tac call*)
apply(*rename-tac* [!] *x*)
apply(*case-tac* [!] *x*)
apply(*rename-tac* [!] *y*)
apply(*case-tac* [!] *y*)
apply(*rename-tac* [*5*−*6*] *z*)
apply (*case-tac* [*5*−*6*] *z*)
apply (*auto simp add*: *fused-resource.fuse.simps parallel-eoracle-def*)
done
apply *simp*
done
done
qed

12.7 A lazy construction and its DH reduction

12.7.1 Defining a lazy construction with an inlined sampler

type-synonym *′grp′ st-state* = (*′grp′* × *′grp′* × *′grp′*) *option*
type-synonym *′grp′ bc-state* = (*′grp′ st-state* × *′grp′ cstate* × *′grp′ cstate*) × *′grp′ auth.state* × *′grp′ auth.state*

context
fixes *sample-triple* :: (*′grp* × *′grp* × *′grp*) *spmf*
begin

abbreviation *basic-core-sinit* :: *′grp bc-state*
where
basic-core-sinit ≡ ((*None*, *CState-Void*, *CState-Void*), (*auth.State-Void*, {}), *auth.State-Void*, {})

fun *basic-core-helper-base* :: (*'grp bc-state*, *unit*, *unit*) *oracle'*
where
basic-core-helper-base ((*s-key*, *CState-Void*, *s-cnv2*), (*auth.State-Void*, *parties1*), *s-auth2*) - =
(*if auth.Alice* ∈ *parties1*
then return-spmf ((), (*s-key*, *CState-Half 0*, *s-cnv2*), (*auth.State-Store* **1**, *parties1*), *s-auth2*)
else return-pmf None)
| *basic-core-helper-base* - - = *return-pmf None*

definition *basic-core-helper* :: (*'grp bc-state*, *icnv-alice* + *icnv-bob*) *handler*
where
basic-core-helper ≡ (λ*state query*.
let handle = λ((*sk*, (*sc1*, *sc2*)), *sa1*, *sa2*). ((*sk*, (*sc2*, *sc1*)), *sa2*, *sa1*) *in*
let func = λ*h-s f s*. *map-spmf* (*h-s o snd*) (*f* (*h-s s*) ()) *in*
let func-alc = *func id basic-core-helper-base in*
let func-bob = *func handle basic-core-helper-base in*
case-sum (λ-. *func-alc state*) (λ-. *func-bob state*) *query*)

fun *basic-core-oracle-adv* :: *unit* + *unit* ⇒ (*'grp st-state* × *'grp auth.state*, *'grp auth.iadv*, *'grp auth.oadv*) *oracle'*
where
basic-core-oracle-adv sel (*None*, *auth.State-Store* -, *parties*) (*Inr* (*Inl* -)) = *do* {
(*gxy*, *gx*, *gy*) ← *sample-triple*;
let out = *case-sum* (λ-. *gx*) (λ-. *gy*) *sel*;
return-spmf (*Inr* (*Inl* (*auth.Out-Look out*)), *Some* (*gxy*, *gx*, *gy*), *auth.State-Store* **1**, *parties*)
}
| *basic-core-oracle-adv sel* (*Some dhs*, *auth.State-Store* -, *parties*) (*Inr* (*Inl* -)) =
(*case dhs of* (*gxy*, *gx*, *gy*) ⇒
let out = *case-sum* (λ-. *gx*) (λ-. *gy*) *sel in*
return-spmf (*Inr* (*Inl* (*auth.Out-Look out*)), *Some dhs*, *auth.State-Store* **1**, *parties*))
| *basic-core-oracle-adv* - (*s-key*, *auth.State-Store* -, *parties*) (*Inr* (*Inr* -)) =
return-spmf (*Inr* (*Inr auth.Out-Fedit*), *s-key*, *auth.State-Collect* **1**, *parties*)
| *basic-core-oracle-adv* - - - = *return-pmf None*

fun *basic-core-oracle-usr-base* :: (*'grp bc-state*, *unit*, *'grp*) *oracle'*
where
basic-core-oracle-usr-base ((*s-key*, *CState-Half* -, *s-cnv2*), *s-auth1*, *auth.State-Collect* -, *parties2*) - =
(*let h-state* = λ*k*. ((*Some k*, *CState-Full* (*0*, **1**), *s-cnv2*), *s-auth1*, *auth.State-Collected*, *parties2*) *in*
if auth.Bob ∈ *parties2 then*
(*case s-key of*
None ⇒ *do* {
(*gxy*, *gx*, *gy*) ← *sample-triple*;

return-spmf (*gxy*, *h-state* (*gxy*, *gx*, *gy*)) }
| *Some* (*gxy*, *gx*, *gy*) ⇒ *return-spmf* (*gxy*, *h-state* (*gxy*, *gx*, *gy*)))
else return-pmf None)
| *basic-core-oracle-usr-base* ((*Some dhs*, *CState-Full* -, *s-cnv2*), *s-auth1*, *auth.State-Collected*, *parties2*) - =
(*case dhs of* (*gxy*, *gx*, *gy*) ⇒
return-spmf (*gxy*, (*Some dhs*, *CState-Full* (*0*, **1**), *s-cnv2*), *s-auth1*, *auth.State-Collected*, *parties2*))
| *basic-core-oracle-usr-base* - - = *return-pmf None*

definition *basic-core-oracle-usr* :: (-, *key.iusr-alice* + *key.iusr-bob*, -) *oracle′*
where
basic-core-oracle-usr ≡ (λ*state query*.
let handle = λ((*sk*, (*sc1*, *sc2*)), *sa1*, *sa2*). ((*sk*, (*sc2*, *sc1*)), *sa2*, *sa1*) *in*
let func = λ*h-o h-s f s*. *map-spmf* (*map-prod h-o h-s*) (*f* (*h-s s*) ()) *in*
let func-alc = *func* (*Inl o key.Out-Alice*) *id basic-core-oracle-usr-base in*
let func-bob = *func* (*Inr o key.Out-Bob*) *handle basic-core-oracle-usr-base in*
case-sum (λ-. *func-alc state*) (λ-. *func-bob state*) *query*)

primcorec *basic-core*
where
cpoke basic-core = (λ(*s-other*, *s-core*) *event*.
map-spmf (*Pair s-other*) (*parallel-handler auth.poke auth.poke s-core event*))
| *cfunc-adv basic-core* = (λ((*s-key*, *s-cnv*), *s-auth1*, *s-auth2*) *iadv*.
let handle = (λ*sel s-init h-out h-state query*.
map-spmf
(λ(*out*, (*s-key′*, *s-auth′*)). (*h-out out*, (*s-key′*, *s-cnv*), *h-state s-auth′ s-auth1 s-auth2*))
(*basic-core-oracle-adv sel* (*s-key*, *s-init*) *query*)) *in*
case-sum (*handle* (*Inl* ()) *s-auth1 Inl* (λ*x y z*. (*x*, *z*))) (*handle* (*Inr* ()) *s-auth2 Inr* (λ*x y z*. (*y*, *x*))) *iadv*)
| *cfunc-usr basic-core* =
(*let handle* = *map-sum* (λ-. *Out-Activation-Alice*) (λ-. *Out-Activation-Bob*) *in*
basic-core-oracle-usr $\oplus_O$ (λ*s q*. *map-spmf* (*Pair* (*handle q*)) (*basic-core-helper s q*)))

primcorec *lazy-core*
where
cpoke lazy-core = (λ*s*. *case-sum* (λ*q*. *basic-core-helper s q*) (*cpoke basic-core s*))
| *cfunc-adv lazy-core* = *cfunc-adv basic-core*
| *cfunc-usr lazy-core* = *basic-core-oracle-usr*

definition *lazy-rest*
where
lazy-rest ≡ *ideal-rest′*

end

12.7.2 Defining a lazy construction with an external sampler

context
begin

private type-synonym *('grp', 'iadv-rest', 'iusr-rest') dh-inp* =
(('grp' auth.iadv + 'grp' auth.iadv) + 'iadv-rest') + (key.iusr-alice + key.iusr-bob) + (icnv-alice + icnv-bob) + 'iusr-rest'

private type-synonym *('grp', 'oadv-rest', 'ousr-rest') dh-out* =
(('grp' auth.oadv + 'grp' auth.oadv) + 'oadv-rest') + ('grp' key.ousr-alice + 'grp' key.ousr-bob) + (ocnv-alice + ocnv-bob) + 'ousr-rest'

```
fun interceptor-base-look :: unit + unit ⇒ 'grp st-state × 'grp auth.state
  ⇒ ('grp auth.oadv-look × 'grp st-state, unit, 'grp × 'grp × 'grp) gpv
  where
    interceptor-base-look sel (None, auth.State-Store -, parties) = do {
      (gxy, gx, gy) ← Pause () Done;
      let out = case-sum (λ-. gx) (λ-. gy) sel;
      Done (auth.Out-Look out, Some (gxy, gx, gy)) }
  | interceptor-base-look sel (Some dhs, auth.State-Store -, parties) = (
      case dhs of (gxy, gx, gy) ⇒
      let out = case-sum (λ-. gx) (λ-. gy) sel in
      Done (auth.Out-Look out, Some (gxy, gx, gy)))
  | interceptor-base-look - - = Fail
```

```
fun interceptor-base-recv :: 'grp bc-state ⇒ ('grp × 'grp bc-state, unit, 'grp × 'grp
× 'grp) gpv
  where
    interceptor-base-recv ((s-key, CState-Half -, s-cnv2), s-auth1, auth.State-Collect
-, parties2) = (
    let h-state = λk. ((Some k, CState-Full (0, 1), s-cnv2), s-auth1, auth.State-Collected,
parties2) in
      if auth.Bob ∈ parties2 then
        case s-key of
          None ⇒ do {
            (gxy, gx, gy) ← Pause () Done;
            Done (gxy, h-state (gxy, gx, gy)) }
        | Some (gxy, gx, gy) ⇒ Done (gxy, h-state (gxy, gx, gy))
      else
        Fail)
 | interceptor-base-recv ((Some dhs, CState-Full -, s-cnv2), s-auth1, auth.State-Collected,
parties2) = (
      case dhs of (gxy, gx, gy) ⇒
     Done (gxy, (Some dhs, CState-Full (0, 1), s-cnv2), s-auth1, auth.State-Collected,
parties2))
  | interceptor-base-recv - = Fail
```

```
fun interceptor :: - ⇒ (-, -, -) dh-inp ⇒ (('grp, -, -) dh-out × -, unit, 'grp ×
'grp × 'grp) gpv
```

where
interceptor (*sc*, *sr*) (*Inl* (*Inl* (*q*))) = (
let select-s = (*case sc of* ((*sk*, -), *sa1*, *sa2*) ⇒ *case-sum* (λ-. (*sk*, *sa1*)) (λ-. (*sk*, *sa2*))) *in*
let handle-s = (λ*x*. *case sc of* ((*sk*, (*sc1*, *sc2*)), *sa1*, *sa2*) ⇒ ((*x*, (*sc1*, *sc2*)), *sa1*, *sa2*)) *in*
let func-look = (λ*sel h-o*. *do* {
(*o-look*, *dhs*) ← *interceptor-base-look* (*sel* ()) (*select-s* (*sel* ())) ;
Done (*Inl* (*Inl* (*h-o* (*Inr* (*Inl o-look*)))), *handle-s dhs*, *sr*) }) *in*
let func-dfe = *do* {
(*out*, *sc'*) ← *lift-spmf* (*cfunc-adv* (*lazy-core undefined*) *sc q*);
Done (*Inl* (*Inl out*), *sc'*, *sr*) } *in*
case q of
(*Inl* (*Inr* (*Inl* -))) ⇒ *func-look Inl Inl*
| (*Inr* (*Inr* (*Inl* -))) ⇒ *func-look Inr Inr*
| - ⇒ *func-dfe*)
| *interceptor* (*sc*, *sr*) (*Inl* (*Inr* (*q*))) = *do* {
((*out*, *es*), *sr'*) ← *lift-spmf* (*rfunc-adv lazy-rest sr q*);
sc' ← *lift-spmf* (*foldl-spmf* (λ*a e*. *cpoke* (*lazy-core undefined*) *a e*) (*return-spmf sc*) *es*);
Done (*Inl* (*Inr out*), (*sc'*, *sr'*)) }
| *interceptor* (*sc*, *sr*) (*Inr* (*Inl* (*q*))) = (
let handle = λ((*sk*, (*sc1*, *sc2*)), *sa1*, *sa2*). ((*sk*, (*sc2*, *sc1*)), *sa2*, *sa1*) *in*
let func-recv = (λ*h-o h-s*. *do* {
(*o-recv*, *sc'*) ← *interceptor-base-recv* (*h-s sc*);
Done (*Inr* (*Inl* (*h-o o-recv*)), *h-s sc'*, *sr*) }) *in*
case-sum (λ-. *func-recv* (*Inl o key.Out-Alice*) *id*) (λ-. *func-recv* (*Inr o key.Out-Bob*) *handle*) *q*)
| *interceptor* (*sc*, *sr*) (*Inr* (*Inr* (*q*))) = *do* {
((*out*, *es*), *sr'*) ← *lift-spmf* (*rfunc-usr lazy-rest sr q*);
sc' ← *lift-spmf* (*foldl-spmf* (λ*a e*. *cpoke* (*lazy-core undefined*) *a e*) (*return-spmf sc*) *es*);
Done (*Inr* (*Inr out*), (*sc'*, *sr'*)) }

end

12.7.3 Reduction to Diffie-Hellman game

definition *DH0-sample* :: ('*grp* × '*grp* × '*grp*) *spmf*
where
DH0-sample = *do* {
x ← *sample-uniform* (*order* $\mathcal{G}$);
y ← *sample-uniform* (*order* $\mathcal{G}$);
return-spmf ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*) }

definition *DH1-sample* :: ('*grp* × '*grp* × '*grp*) *spmf*
where
DH1-sample = *do* {
x ← *sample-uniform* (*order* $\mathcal{G}$);

y ← *sample-uniform* (*order* 𝒢);
z ← *sample-uniform* (*order* 𝒢);
return-spmf (**g** [^] *z*, **g** [^] *x*, **g** [^] *y*) }

lemma *lossless-DH0-sample* [*simp*]: *lossless-spmf DH0-sample*
by (*auto simp add*: *DH0-sample-def sample-uniform-def intro*: *order-gt-0*)

lemma *lossless-DH1-sample* [*simp*]: *lossless-spmf DH1-sample*
by (*auto simp add*: *DH1-sample-def sample-uniform-def intro*: *order-gt-0*)

definition *DH-adversary-curry* :: (*'grp* × *'grp* × *'grp* ⇒ *bool spmf*) ⇒ *'grp* ⇒ *'grp* ⇒ *'grp* ⇒ *bool spmf*
where
DH-adversary-curry ≡ (λ*A x y z*. *bind-spmf* (*return-spmf* (*z*, *x*, *y*)) *A*)

definition *DH-adversary*
where
DH-adversary D ≡ *DH-adversary-curry* (λ*xyz*.
run-gpv (*obsf-oracle* (*obsf-oracle* (λ(*tpl*, *s*) *q*. *map-spmf* (*apsnd* (*Pair tpl*) ∘ *fst*) (*exec-gpv* (λ- -. *return-spmf* (*tpl*, ())) (*interceptor s q*) ())))))
(*obsf-distinguisher D*) (*OK* (*OK* (*xyz*, *basic-core-sinit*, *basic-rest-sinit*))))

context
begin

private abbreviation *S-ic-asm s-cnv1 s-cnv2 s-krn1 s-krn2* ≡
let s-cnvs = {*CState-Void*} ∪ {*CState-Half 0*} ∪ {*CState-Full* (*0*, **1**)} *in*
let s-krns = {*auth.State-Void*} ∪ {*auth.State-Store* **1**} ∪ {*auth.State-Collect* **1**} ∪ {*auth.State-Collected*} *in*
s-cnv1 ∈ *s-cnvs* ∧ *s-cnv2* ∈ *s-cnvs* ∧ *s-krn1* ∈ *s-krns* ∧ *s-krn2* ∈ *s-krns*

private inductive *S-ic* :: (*'grp* × *'grp* × *'grp*) *spmf* ⇒ (*'grp bc-state* × (*unit* × *unit*) × *'auth1-s-rest* × *'auth2-s-rest*) *spmf* ⇒
((*'grp* × *'grp* × *'grp*) × *'grp bc-state* × (*unit* × *unit*) × *'auth1-s-rest* × *'auth2-s-rest*) *spmf* ⇒ *bool*
for 𝒮 :: (*'grp* × *'grp* × *'grp*) *spmf* **where**
S-ic 𝒮 (*return-spmf* (((*None*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ((), ()), *s-rest1*, *s-rest2*))
(*map-spmf* (λ*x*. (*x*, (((*None*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ((), ()), *s-rest1*, *s-rest2*))) 𝒮)
if *S-ic-asm s-cnv1 s-cnv2 s-krn1 s-krn2*
| *S-ic* 𝒮 (*return-spmf* (((*Some x*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ((), ()), *s-rest1*, *s-rest2*))
(*return-spmf* (*x*, (((*Some x*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ((), ()), *s-rest1*, *s-rest2*)))
if *S-ic-asm s-cnv1 s-cnv2 s-krn1 s-krn2*

private lemma *trace-eq-intercept*:
defines *outs-adv* ≡ ((*UNIV* <+> *UNIV* <+> *UNIV*) <+> *UNIV* <+> *UNIV*

<+> *UNIV*) <+> *UNIV* <+> *UNIV*
and *outs-usr* ≡ (*UNIV* <+> *UNIV*) <+> (*UNIV* <+> *UNIV*) <+> *UNIV*
<+> *UNIV*
assumes *lossless-spmf sample-triple*
shows *trace-callee-eq* (*fused-resource.fuse* (*lazy-core sample-triple*) *lazy-rest*)
(λ(*tpl*, *s*) *q*. *map-spmf* (*apsnd* (*Pair tpl*) ∘ *fst*) (*exec-gpv* (λ- -. *return-spmf* (*tpl*,
())) (*interceptor s q*) ()))
(*outs-adv* <+> *outs-usr*)
(*return-spmf* (*basic-core-sinit*, *basic-rest-sinit*)) (*pair-spmf sample-triple* (*return-spmf*
(*basic-core-sinit*, *basic-rest-sinit*)))
(**is** *trace-callee-eq ?L ?R ?OI ?sl ?sr*)
proof −
have *auth-poke-alt*[*simplified split-def Let-def*]:
auth.poke = (λ(*sl*, *sr*) *q*. *let p* = *auth.case-event id q in*
map-spmf (*Pair sl*) (*if p* ∈ *sr then return-pmf None else return-spmf* (*insert*
p sr)))
by (*rule ext*)+ (*simp add*: *split-def Let-def split*!: *auth.event.splits*)

note *S-ic-cases* = *S-ic.cases*[**where** 𝒮=*sample-triple*, *simplified*]
note *S-ic-intros* = *S-ic.intros*[**where** 𝒮=*sample-triple*, *simplified*]

note [*simp*] = *assms*(*3*)[*unfolded lossless-spmf-def*] *mk-lossless-lossless*[*OF assms*(*3*)]

fused-resource.fuse.simps lazy-rest-def basic-core-oracle-usr-def basic-core-helper-def
exec-gpv-bind spmf.map-comp map-bind-spmf bind-map-spmf bind-spmf-const
o-def Let-def split-def
extend-state-oracle-def plus-eoracle-def parallel-eoracle-def map-fun-def

have *trace-callee-eq ?L ?R ?OI ?sl ?sr*
unfolding *assms*(*1*,*2*) **apply** (*rule trace′-eqI-sim-upto*[**where** *S*=*S-ic sam-*
ple-triple])
subgoal *Init-OK*
by (*simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *pair-spmf-alt-def S-ic-intros*)
subgoal *Out-OK* **for** *sl sr q*
apply (*cases q*)
subgoal for *q-adv*
apply (*cases q-adv*)
subgoal for *q-adv-core*
apply (*cases q-adv-core*)
subgoal for *q-acore1*
apply (*cases q-acore1*)
subgoal for *q-drop* **by** (*erule S-ic-cases*) *simp-all*
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inl* (), (*None*, *s-krn1*, *s-act1*)) *rule*: *intercep-*
tor-base-look.cases) *auto*

subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inl* (), (*Some x*, *s-krn1*, *s-act1*)) *rule*: *interceptor-base-look.cases*) *auto*
done
subgoal for *q-fedit*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inl* (), (*None*, *s-krn1*, *s-act1*), *Inr* (*Inr q-fedit*)) *rule*: *basic-core-oracle-adv.cases*) *simp-all*
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inl* (), (*Some x*, *s-krn1*, *s-act1*), *Inr* (*Inr q-fedit*)) *rule*: *basic-core-oracle-adv.cases*) *simp-all*
done
done
done
subgoal for *q-acore2*
apply (*cases q-acore2*)
subgoal for *q-drop* **by** (*erule S-ic-cases*) *simp-all*
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inr* (), (*None*, *s-krn2*, *s-act2*)) *rule*: *interceptor-base-look.cases*) *auto*
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inr* (), (*Some x*, *s-krn2*, *s-act2*)) *rule*: *interceptor-base-look.cases*) *auto*
done
subgoal for *q-fedit*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inr* (), (*None*, *s-krn2*, *s-act2*), *Inr* (*Inr q-fedit*)) *rule*: *basic-core-oracle-adv.cases*) *simp-all*
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inr* (), (*Some x*, *s-krn2*, *s-act2*), *Inr* (*Inr q-fedit*)) *rule*: *basic-core-oracle-adv.cases*) *simp-all*
done
done
done
done
subgoal for *q-adv-rest*
apply (*cases q-adv-rest*)
subgoal for *q-arest1* **by** (*erule S-ic-cases*) *simp-all*
subgoal for *q-arest2* **by** (*erule S-ic-cases*) *simp-all*
done
done
subgoal for *q-usr*
apply (*cases q-usr*)

```
subgoal for q-usr-core
  apply (cases q-usr-core)
  subgoal for q-alice
    apply (erule S-ic-cases)
    subgoal for s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2
      by (simp, case-tac (((None, s-cnv1, s-cnv2), (s-krn1, s-act1), s-krn2,
s-act2), ()) rule: basic-core-oracle-usr-base.cases) (auto split: option.splits)
    subgoal for s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2
      by (simp, case-tac (((Some x, s-cnv1, s-cnv2), (s-krn1, s-act1), s-krn2,
s-act2), ()) rule: basic-core-oracle-usr-base.cases) (auto split: option.splits)
    done
  subgoal for q-bob
    apply (erule S-ic-cases)
    subgoal for s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2
      by (simp, case-tac (((None, s-cnv2, s-cnv1), (s-krn2, s-act2), s-krn1,
s-act1), ()) rule: basic-core-oracle-usr-base.cases) (auto split: option.splits)
    subgoal for s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2
      by (simp, case-tac (((Some x, s-cnv2, s-cnv1), (s-krn2, s-act2), s-krn1,
s-act1), ()) rule: basic-core-oracle-usr-base.cases) (auto split: option.splits)
    done
  done
subgoal for q-usr-rest
  apply (cases q-usr-rest)
  subgoal for q-act
    apply (cases q-act)
    subgoal for a-alice by (erule S-ic-cases) simp-all
    subgoal for a-bob by (erule S-ic-cases) simp-all
    done
  subgoal for q-urest
    apply (cases q-urest)
    subgoal for q-urest1 by (erule S-ic-cases) simp-all
    subgoal for q-urest2 by (erule S-ic-cases) simp-all
    done
  done
done
done
subgoal State-OK for sl sr q
apply (cases q)
subgoal for q-adv
  apply (cases q-adv)
  subgoal for q-adv-core
    apply (cases q-adv-core)
    subgoal for q-acore1
      apply (cases q-acore1)
      subgoal for q-drop by (erule S-ic-cases) simp-all
      subgoal for q-lfe
        apply (cases q-lfe)
        subgoal for q-look
          apply (erule S-ic-cases)
```

subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (*Inl* (), (*None*, *s-krn1*, *s-act1*)) *rule*: *interceptor-base-look.cases*)
apply (*simp-all add*: *map-spmf-conv-bind-spmf*[*symmetric*])
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *cond-spmf-fst-def vimage-def intro*!: *trace-eq-simcl-map S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (*Inl* (), (*Some x*, *s-krn1*, *s-act1*)) *rule*: *interceptor-base-look.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-ic-intros*)
done
subgoal for *q-fedit*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inl* (), (*None*, *s-krn1*, *s-act1*), *Inr* (*Inr q-fedit*)) *rule*: *basic-core-oracle-adv.cases*)
(*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inl* (), (*Some x*, *s-krn1*, *s-act1*), *Inr* (*Inr q-fedit*)) *rule*: *basic-core-oracle-adv.cases*)
(*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
done
done
done
subgoal for *q-acore2*
apply (*cases q-acore2*)
subgoal for *q-drop* **by** (*erule S-ic-cases*) *simp-all*
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (*Inr* (), (*None*, *s-krn2*, *s-act2*)) *rule*: *interceptor-base-look.cases*)
apply (*simp-all add*: *map-spmf-conv-bind-spmf*[*symmetric*])
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *cond-spmf-fst-def vimage-def intro*!: *trace-eq-simcl-map S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (*Inr* (), (*Some x*, *s-krn2*, *s-act2*)) *rule*: *interceptor-base-look.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-ic-intros*)
done
subgoal for *q-fedit*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*

by (*simp*, *case-tac* (*Inr* (), (*None*, *s-krn2*, *s-act2*), *Inr* (*Inr* *q-fedit*)) *rule*: *basic-core-oracle-adv.cases*)
(*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
by (*simp*, *case-tac* (*Inr* (), (*Some x*, *s-krn2*, *s-act2*), *Inr* (*Inr* *q-fedit*)) *rule*: *basic-core-oracle-adv.cases*)
(*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
done
done
done
done
subgoal for *q-adv-rest*
apply (*cases q-adv-rest*)
subgoal for *q-urest1*
apply (*erule S-ic-cases*)
subgoal
apply *clarsimp*
apply (*subst bind-commute-spmf*)
apply (*subst* (*2*) *bind-commute-spmf*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, (-, *x*), -). *x*
and *p*=λ(*a*, (*b*, -), *d*). (*a*, *b*, *d*) **and** *f*=λ(*a*, *b*, *d*) *c*. (*a*, (*b*, *c*), *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal
apply *clarsimp*
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, (-, *x*), -). *x*
and *p*=λ(*a*, (*b*, -), *d*). (*a*, *b*, *d*) **and** *f*=λ(*a*, *b*, *d*) *c*. (*a*, (*b*, *c*), *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-ic-intros*)
done
subgoal for *q-urest2*
apply (*erule S-ic-cases*)
subgoal
apply *clarsimp*
apply (*subst bind-commute-spmf*)

apply (*subst* (*2*) *bind-commute-spmf*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, -, -, *x*). *x*
and *p*=λ(*a*, *b*, *c*, -). (*a*, *b*, *c*) **and** *f*=λ(*a*, *b*, *c*) *d*. (*a*, *b*, *c*, *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal
apply *clarsimp*
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, -, -, *x*). *x*
and *p*=λ(*a*, *b*, *c*, -). (*a*, *b*, *c*) **and** *f*=λ(*a*, *b*, *c*) *d*. (*a*, *b*, *c*, *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-ic-intros*)
done
done
done
subgoal for *q-usr*
apply (*cases q-usr*)
subgoal for *q-usr-core*
apply (*cases q-usr-core*)
subgoal for *q-alice*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*None*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ()) *rule*: *basic-core-oracle-usr-base.cases*)
proof (*goal-cases*)
case (*1 s-key - s-cnv2 s-auth1 - parties2 -*)
then show *?case* **by** (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *cond-spmf-fst-def vimage-def intro*!: *trace-eq-simcl-map S-ic-intros*)
qed (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*Some x*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ()) *rule*: *basic-core-oracle-usr-base.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
done
subgoal for *q-bob*
apply (*erule S-ic-cases*)

subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*None*, *s-cnv2*, *s-cnv1*), (*s-krn2*, *s-act2*), *s-krn1*, *s-act1*), ()) *rule*: *basic-core-oracle-usr-base.cases*)
proof (*goal-cases*)
case (*1 s-key - s-cnv2 s-auth1 - parties2 -*)
then show *?case* **by** (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *cond-spmf-fst-def vimage-def intro*!: *trace-eq-simcl-map S-ic-intros*)
qed (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*Some x*, *s-cnv2*, *s-cnv1*), (*s-krn2*, *s-act2*), *s-krn1*, *s-act1*), ()) *rule*: *basic-core-oracle-usr-base.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
done
done
subgoal for *q-usr-rest*
apply (*cases q-usr-rest*)
subgoal for *q-act*
apply (*cases q-act*)
subgoal for *a-alice*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*None*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ()) *rule*: *basic-core-helper-base.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*Some x*, *s-cnv1*, *s-cnv2*), (*s-krn1*, *s-act1*), *s-krn2*, *s-act2*), ()) *rule*: *basic-core-helper-base.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
done
subgoal for *a-bob*
apply (*erule S-ic-cases*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*None*, *s-cnv2*, *s-cnv1*), (*s-krn2*, *s-act2*), *s-krn1*, *s-act1*), ()) *rule*: *basic-core-helper-base.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal for *s-cnv1 s-cnv2 s-krn1 s-krn2 x s-act1 s-act2 s-rest1 s-rest2*
apply (*simp*, *case-tac* (((*Some x*, *s-cnv2*, *s-cnv1*), (*s-krn2*, *s-act2*), *s-krn1*, *s-act1*), ()) *rule*: *basic-core-helper-base.cases*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
done
done
subgoal for *q-urest*
apply (*cases q-urest*)

subgoal for *q-urest1*
apply (*erule S-ic-cases*)
subgoal
apply *clarsimp*
apply (*subst bind-commute-spmf*)
apply (*subst* (*2*) *bind-commute-spmf*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, (-, *x*), -). *x*
and *p*=λ(*a*, (*b*, -), *d*). (*a*, *b*, *d*) **and** *f*=λ(*a*, *b*, *d*) *c*. (*a*, (*b*, *c*), *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal
apply *clarsimp*
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, (-, *x*), -). *x*
and *p*=λ(*a*, (*b*, -), *d*). (*a*, *b*, *d*) **and** *f*=λ(*a*, *b*, *d*) *c*. (*a*, (*b*, *c*), *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-ic-intros*)
done
subgoal for *q-urest2*
apply (*erule S-ic-cases*)
subgoal
apply *clarsimp*
apply (*subst bind-commute-spmf*)
apply (*subst* (*2*) *bind-commute-spmf*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, -, -, *x*). *x*
and *p*=λ(*a*, *b*, *c*, -). (*a*, *b*, *c*) **and** *f*=λ(*a*, *b*, *c*) *d*. (*a*, *b*, *c*, *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base S-ic-intros*)
subgoal
apply *clarsimp*
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)
apply (*subst* (*1 2*) *cond-spmf-fst-bind*)

apply (*clarsimp intro*!: *trace-eq-simcl-bind simp add*: *auth-poke-alt set-scale-spmf split*: *if-split-asm*)
apply (*subst* (*asm*) (*1 2 3 4*) *foldl-spmf-helper2*[**where** *acc*=*return-spmf* - **and** *q*=λ(-, -, -, *x*). *x*
and *p*=λ(*a*, *b*, *c*, -). (*a*, *b*, *c*) **and** *f*=λ(*a*, *b*, *c*) *d*. (*a*, *b*, *c*, *d*), *simplified*])
by (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-ic-intros*)
done
done
done
done
done
done

then show *?thesis* **by** *simp*
qed

private abbreviation *dummy x* ≡ *TRY map-spmf OK x ELSE return-spmf Fault*

lemma *reduction*: *advantage D* (*obsf-resource* (*RES* (*fused-resource.fuse* (*lazy-core DH1-sample*) *lazy-rest*) (*basic-core-sinit*, *basic-rest-sinit*)))
(*obsf-resource* (*RES* (*fused-resource.fuse* (*lazy-core DH0-sample*) *lazy-rest*) (*basic-core-sinit*, *basic-rest-sinit*))) = *ddh.advantage* $\mathcal{G}$ (*DH-adversary D*)
proof −
have *fact1*: *bind-spmf* (*DH0-sample*) *A* = *do* {
x ← *sample-uniform* (*order* $\mathcal{G}$);
y ← *sample-uniform* (*order* $\mathcal{G}$);
(*DH-adversary-curry A*) (**g** [^] *x*) (**g** [^] *y*) (**g** [^] (*x* ∗ *y*))
} **for** *A* **by** (*simp add*: *DH0-sample-def DH-adversary-curry-def nat-pow-pow*)

have *fact2*: *bind-spmf DH1-sample A* = *do* {
x ← *sample-uniform* (*order* $\mathcal{G}$);
y ← *sample-uniform* (*order* $\mathcal{G}$);
z ← *sample-uniform* (*order* $\mathcal{G}$);
(*DH-adversary-curry A*) (**g** [^] *x*) (**g** [^] *y*) (**g** [^] (*z*))
} **for** *A* **by** (*simp add*: *DH1-sample-def DH-adversary-curry-def*)

{
fix *sample-triple* :: (*'grp* × *'grp* × *'grp*) *spmf*
assume ∗: *lossless-spmf sample-triple*
define *s-init* **where** *s-init* ≡ (*basic-core-sinit*, *basic-rest-sinit*)
have [*unfolded s-init-def*, *simp*]: *dummy* (*dummy* (*return-spmf s-init*)) = *return-spmf* (*OK* (*OK s-init*)) **by** *auto*
have [*unfolded s-init-def*, *simp*]: *dummy* (*dummy* (*pair-spmf sample-triple* (*return-spmf s-init*))) =
map-spmf (*OK* ∘ *OK*) (*pair-spmf sample-triple* (*return-spmf s-init*))
using ∗ **by** (*simp add*: *o-def map-spmf-conv-bind-spmf pair-spmf-alt-def*)

have *connect D* (*RES* (*obsf-oracle* (*obsf-oracle* (*fused-resource.fuse* (*lazy-core sample-triple*) *lazy-rest*))) (*OK* (*OK* (*basic-core-sinit*, *basic-rest-sinit*)))) =
bind-spmf (*map-spmf* (*OK o OK*) (*pair-spmf sample-triple* (*return-spmf* (*basic-core-sinit*, *basic-rest-sinit*))))
(*run-gpv* (*obsf-oracle* (*obsf-oracle* (λ(*tpl*, *s*) *q*. *map-spmf* ((*apsnd* (*Pair tpl*)) *o fst*) (*exec-gpv* (λ- -. *return-spmf* (*tpl*, ())) (*interceptor s q*) ()))))) *D*) **for** *D*
apply (*simp add*: *connect-def exec-gpv-resource-of-oracle spmf.map-comp*)
apply (*subst return-bind-spmf*[**where** *x*=*OK* (*OK* (*basic-core-sinit*, *basic-rest-sinit*)), *symmetric*])
apply (*rule trace-callee-eq-run-gpv*[**where** *?I1.0*=(λ-. *True*) **and** *?I2.0*=(λ-. *True*) **and** *ℐ*=*ℐ-full* **and** *A*=*UNIV*])
subgoal by (*rule trace-eq-intercept*[*OF* ∗, *THEN trace-callee-eq-obsf-oracleI*, *THEN trace-callee-eq-obsf-oracleI*, *simplified*])
by (*simp-all add*: ∗ *pair-spmf-alt-def*)
} **note** *detach-sampler* = *this*

show *?thesis*
unfolding *advantage-def*
apply (*subst* (*1 2*) *spmf-obsf-distinguisher-obsf-resource-True*[*symmetric*])
apply (*subst* (*1 2*) *obsf-resource-of-oracle*)+
by (*simp add*: *detach-sampler pair-spmf-alt-def bind-map-spmf fact1 fact2*)
(*simp add*: *ddh.advantage-def ddh.ddh-0-def ddh.ddh-1-def DH-adversary-def*)
qed

end

12.8 Proving the trace-equivalence of simplified Ideal and Lazy constructions

context
begin

private abbreviation *isample-nat* ≡ *sample-uniform* (*order 𝒢*)
private abbreviation *isample-key* ≡ *spmf-of-set* (*carrier 𝒢*)
private abbreviation *isample-pair-nn* ≡ *pair-spmf isample-nat isample-nat*
private abbreviation *isample-pair-nk* ≡ *pair-spmf isample-nat isample-key*

private inductive *S-il* :: ((*'grp astate* × *unit*) × *estate* × *'grp key.state*) *spmf* ⇒ *'grp bc-state spmf* ⇒ *bool*
where
— (Auth1 =a)@(Auth2 =0)
sil-0-0: *S-il* (*return-spmf* ((*None*, ()), (*False*, (*EState-Void*, *s-act1*), *EState-Void*, *s-act2*), *key.PState-Store*, {}))
(*return-spmf* ((*None*, *CState-Void*, *CState-Void*), (*auth.State-Void*, *s-act1*), *auth.State-Void*, *s-act2*))
— ../(Auth1 =a)@(Auth2 =0) # wl
| *sil-1-0*: *S-il* (*return-spmf* ((*None*, ()), (*False*, (*EState-Store*, *s-act1*), *EState-Void*, *s-act2*), *key.PState-Store*, {}))

(*return-spmf* ((*None*, *CState-Half 0*, *CState-Void*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Void*, *s-act2*))
if *auth.Alice* ∈ *s-act1*
| *sil-2-0*: *S-il* (*map-spmf* (λ*k*. ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Void*, *s-act2*), *key.State-Store* *k*, {})) *isample-key*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Void*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Void*, *s-act2*))
if *auth.Alice* ∈ *s-act1*
— ../(Auth1 =a)@(Auth2 =0) # look
| *sil-1′-0*: *S-il* (*map-spmf* (λ*y*. ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*False*, (*EState-Store*, *s-act1*), *EState-Void*, *s-act2*), *key.PState-Store*, {})) *isample-nat*)
(*map-spmf* (λ*yz*. ((*Some* (**g** [^] *snd yz*, **g** [^] (*x* :: *nat*), **g** [^] *fst yz*), *CState-Half 0*, *CState-Void*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Void*, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act1*
| *sil-2′-0*: *S-il* (*map-spmf* (λ*yk*. ((*Some* (**g** [^] *x*, **g** [^] *fst yk*), ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Void*, *s-act2*), *key.State-Store* (*snd yk*), {})) *isample-pair-nk*)
(*map-spmf* (λ*yz*. ((*Some* (**g** [^] *snd yz*, **g** [^] (*x* :: *nat*), **g** [^] *fst yz*), *CState-Half 0*, *CState-Void*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Void*, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act1*
— (Auth1 =a)@(Auth2 =1)
| *sil-0-1*: *S-il* (*return-spmf* ((*None*, ()), (*False*, (*EState-Void*, *s-act1*), *EState-Store*, *s-act2*), *key.PState-Store*, {}))
(*return-spmf* ((*None*, *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Store* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act2*
— ../(Auth1 =a)@(Auth2 =1) # wl
| *sil-1-1*: *S-il* (*return-spmf* ((*None*, ()), (*False*, (*EState-Store*, *s-act1*), *EState-Store*, *s-act2*), *key.PState-Store*, {}))
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2*
| *sil-2-1*: *S-il* (*map-spmf* (λ*k*. ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Store*, *s-act2*), *key.State-Store* *k*, *s-actk*)) *isample-key*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *key.Alice* ∉ *s-actk* **and** *auth.Bob* ∈ *s-act1* ⟷ *key.Bob* ∈ *s-actk*
| *sil-3-1*: *S-il* (*return-spmf* ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Store*, *s-act2*), *key.State-Store* *k*, *s-actk*))
(*map-spmf* (λ*xy*. ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] *fst xy*, **g** [^] *snd xy*), *CState-Half 0*, *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Store* **1**, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *key.Alice* ∉ *s-actk* **and** *auth.Bob* ∈ *s-act1* **and** *key.Bob* ∈ *s-actk* **and** k = **g** [^] *z*
— ../(Auth1 =a)@(Auth2 =1) # look
| *sil-1c-1c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*False*, (*EState-Store*,

s-act1), *EState-Store*, *s-act2*), *key.PState-Store*, {}))
(*map-spmf* (λ*z*. ((*Some* (**g** [^] *z*, **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Half 0*, *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*)) *isample-nat*)
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2*
| *sil-2c-1c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Store*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] *z*, **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *key.Alice* ∉ *s-actk* **and** *auth.Bob* ∈ *s-act1* ⟷ *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z* **and** *z* ∈ *set-spmf isample-nat*
| *sil-3c-1c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Store*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Half 0*, *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Store* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *key.Alice* ∉ *s-actk* **and** *auth.Bob* ∈ *s-act1* **and** *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z*
— (Auth1 =a)@(Auth2 =2)
| *sil-0-2*: *S-il* (*map-spmf* (λ*k*. ((*None*, ()), (*True*, (*EState-Void*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, {})) *isample-key*)
(*return-spmf* ((*None*, *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act2*
— ../(Auth1 =a)@(Auth2 =2) # wl
| *sil-1-2*: *S-il* (*map-spmf* (λ*k*. ((*None*, ()), (*True*, (*EState-Store*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*)) *isample-key*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* ⟷ *key.Alice* ∈ *s-actk* **and** *key.Bob* ∉ *s-actk*
| *sil-2-2*: *S-il* (*map-spmf* (λ*k*. ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*)) *isample-key*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* ⟷ *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* ⟷ *key.Bob* ∈ *s-actk*
| *sil-3-2*: *S-il* (*return-spmf* ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*map-spmf* (λ*xy*. ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] *fst xy*, **g** [^] *snd xy*), *CState-Half 0*, *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Collect* **1**, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* ⟷ *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* **and** *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z*
— ../(Auth1 =a)@(Auth2 =2) # look
| *sil-1c-2c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Store*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] *z*, **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Half 0*,

CState-Half 0), (*auth.State-Store* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* ⟷ *key.Alice* ∈ *s-actk* **and** *key.Bob* ∉ *s-actk* **and** *k* = **g** [^] *z* **and** *z* ∈ *set-spmf isample-nat*
| *sil-2c-2c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] *z*, **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* ⟷ *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* ⟷ *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z* **and** *z* ∈ *set-spmf isample-nat*
| *sil-3c-2c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Half 0*, *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* ⟷ *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* **and** *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z*
— (Auth1 =a)@(Auth2 =3)
— ../(Auth1 =a)@(Auth2 =3) # wl
| *sil-1-3*: *S-il* (*return-spmf* ((*None*, ()), (*True*, (*EState-Store*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*map-spmf* (λ*xy*. ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] *fst xy*, **g** [^] *snd xy*), *CState-Full* (*0*, **1**), *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Collected*, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* **and** *key.Alice* ∈ *s-actk* **and** *key.Bob* ∉ *s-actk* **and** *k* = **g** [^] *z*
| *sil-2-3*: *S-il* (*return-spmf* ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*map-spmf* (λ*xy*. ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] *fst xy*, **g** [^] *snd xy*), *CState-Full* (*0*, **1**), *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Collected*, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* **and** *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* ⟷ *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z*
| *sil-3-3*: *S-il* (*return-spmf* ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*map-spmf* (λ*xy*. ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] *fst xy*, **g** [^] *snd xy*), *CState-Full* (*0*, **1**), *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Collected*, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* **and** *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* **and** *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z*
— ../(Auth1 =a)@(Auth2 =3) # look
| *sil-1c-3c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Store*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Full* (*0*, **1**), *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Collected*, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* **and** *key.Alice* ∈ *s-actk* **and** *key.Bob* ∉ *s-actk* **and** *k* = **g** [^] *z*
| *sil-2c-3c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Collect*,

s-act1), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Full* (*0*, **1**), *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Collected*, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* **and** *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* ⟷ *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z*
| *sil-3c-3c*: *S-il* (*return-spmf* ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*True*, (*EState-Collect*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* (**g** [^] (*z* :: *nat*), **g** [^] (*x* :: *nat*), **g** [^] (*y* :: *nat*)), *CState-Full* (*0*, **1**), *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Collected*, *s-act2*))
if *auth.Alice* ∈ *s-act1* **and** *auth.Alice* ∈ *s-act2* **and** *auth.Bob* ∈ *s-act2* **and** *key.Alice* ∈ *s-actk* **and** *auth.Bob* ∈ *s-act1* **and** *key.Bob* ∈ *s-actk* **and** *k* = **g** [^] *z*
— (Auth1 =a)@(Auth2 =1')
| *sil-0-1′*: *S-il* (*map-spmf* (λ*x*. ((*Some* (**g** [^] *x*, **g** [^] *y*), ()), (*False*, (*EState-Void*, *s-act1*), *EState-Store*, *s-act2*), *key.PState-Store*, {})) *isample-nat*)
(*map-spmf* (λ*xz*. ((*Some* (**g** [^] *snd xz*, **g** [^] *fst xz*, **g** [^] (*y* :: *nat*)), *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Store* **1**, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act2*
— (Auth1 =a)@(Auth2 =2')
| *sil-0-2′*: *S-il* (*map-spmf* (λ*xk*. ((*Some* (**g** [^] *fst xk*, **g** [^] *y*), ()), (*True*, (*EState-Void*, *s-act1*), *EState-Collect*, *s-act2*), *key.State-Store* (*snd xk*), {})) *isample-pair-nk*)
(*map-spmf* (λ*xz*. ((*Some* (**g** [^] *snd xz*, **g** [^] *fst xz*, **g** [^] (*y* :: *nat*)), *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Collect* **1**, *s-act2*)) *isample-pair-nn*)
if *auth.Alice* ∈ *s-act2*

private lemma *trac-eq-core-il*: *trace-core-eq ideal-core′* (*lazy-core DH1-sample*)
((*UNIV* <+> *UNIV*) <+> *UNIV* <+> *UNIV*) ((*UNIV* <+> *UNIV* <+> *UNIV*) <+> *UNIV* <+> *UNIV* <+> *UNIV*) (*UNIV* <+> *UNIV*)
(*return-spmf ideal-s-core′*) (*return-spmf basic-core-sinit*)
proof −
have *isample-key-conv-nat*[*simplified map-spmf-conv-bind-spmf*]:
map-spmf f isample-key = *map-spmf* (λ*x*. *f* (**g** [^] *x*)) *isample-nat* **for** *f*
unfolding *sample-uniform-def carrier-conv-generator*
by (*simp add*: *map-spmf-of-set-inj-on*[*OF inj-on-generator*, *symmetric*] *spmf.map-comp o-def*)

have [*simp*]: *weight-spmf isample-nat* = *1*
by (*simp add*: *finite-carrier order-gt-0-iff-finite*)

have [*simp*]: *weight-spmf isample-key* = *1*
by (*simp add*: *carrier-not-empty cyclic-group.finite-carrier cyclic-group-axioms*)

have [*simp*]: *mk-lossless isample-nat* = *isample-nat*
by (*simp add*: *mk-lossless-def*)

have [*simp*]: *mk-lossless isample-pair-nn* = *isample-pair-nn*
by (*simp add*: *mk-lossless-def*)

```
  have [simp]: mk-lossless isample-pair-nk = isample-pair-nk
    by (simp add: mk-lossless-def)

  note [simp] = basic-core-helper-def basic-core-oracle-usr-def eleak-def DH1-sample-def
    Let-def split-def exec-gpv-bind spmf.map-comp o-def map-bind-spmf bind-map-spmf
bind-spmf-const

  show ?thesis
    apply (rule trace-core-eq-simI-upto[where S=S-il])
    subgoal Init-OK
      by (simp add: ideal-s-core'-def einit-def sil-0-0)
    subgoal POut-OK for sl sr query
      apply (cases query)
      subgoal for e-usrs
        apply (cases e-usrs)
      subgoal for e-alice by (erule S-il.cases) (auto simp add: map-spmf-conv-bind-spmf[symmetric])
      subgoal for e-bob by (erule S-il.cases) (auto simp add: map-spmf-conv-bind-spmf[symmetric])
        done
      subgoal for e-chns
        apply (cases e-chns)
        subgoal for e-chn1
          apply (cases e-chn1)
          subgoal for e-shell
            apply (cases e-shell)
        subgoal a-alice by (erule S-il.cases) (auto simp add: map-spmf-conv-bind-spmf[symmetric])
        subgoal a-bob by (erule S-il.cases) (auto simp add: map-spmf-conv-bind-spmf[symmetric])
            done
          done
        subgoal for e-chn2
          apply (cases e-chn2)
          subgoal for e-shell
            apply (cases e-shell)
        subgoal a-alice by (erule S-il.cases) (auto simp add: map-spmf-conv-bind-spmf[symmetric])
        subgoal a-bob by (erule S-il.cases) (auto simp add: map-spmf-conv-bind-spmf[symmetric])
            done
          done
        done
      done
    subgoal PState-OK for sl sr query
      apply (cases query)
      subgoal for e-usrs
        apply (cases e-usrs)
        subgoal for e-alice
        proof (erule S-il.cases, goal-cases)
          case (26 s-act2 y s-act1) — Corresponds to sil-0-1'
          then show ?case
            apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
            apply (simp add: pair-spmf-alt-def map-spmf-conv-bind-spmf)
```

apply (*rule trace-eq-simcl-bindI*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros*)
next
case (*27 s-act2 y s-act1*) — Corresponds to *sil-0-2′*
then show *?case*
apply (*clarsimp simp add*: *pair-spmf-alt-def isample-key-conv-nat*)
apply (*simp add*: *bind-bind-conv-pair-spmf*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-il.intros trace-eq-simcl-map*)
qed (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros trace-eq-simcl.base*)
subgoal for *e-bob*
proof (*erule S-il.cases*, *goal-cases*)
case (*4 s-act1 x s-act2*) — Corresponds to *sil-1′-0*
then show *?case*
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*simp add*: *pair-spmf-alt-def map-spmf-conv-bind-spmf*)
apply (*rule trace-eq-simcl-bindI*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros*)
next
case (*5 s-act1 x s-act2*) — Corresponds to *sil-2′-0*
then show *?case*
apply (*clarsimp simp add*: *pair-spmf-alt-def isample-key-conv-nat*)
apply (*simp add*: *bind-bind-conv-pair-spmf*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *S-il.intros trace-eq-simcl-map*)
qed (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros trace-eq-simcl.base*)
done
subgoal for *e-chns*
apply (*cases e-chns*)
subgoal for *e-auth1*
apply (*cases e-auth1*)
subgoal for *e-shell*
apply (*cases e-shell*)
subgoal *a-alice* **by** (*erule S-il.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros trace-eq-simcl.base*)
subgoal *a-bob* **by** (*erule S-il.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros trace-eq-simcl.base*)
done
done
subgoal for *e-auth2*
apply (*cases e-auth2*)
subgoal for *e-shell*
apply (*cases e-shell*)
subgoal *a-alice* **by** (*erule S-il.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros trace-eq-simcl.base*)
subgoal *a-bob* **by** (*erule S-il.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros trace-eq-simcl.base*)

done
done
done
done
subgoal *AOut-OK* **for** *sl sr query*
apply (*cases query*)
subgoal for *q-auth1*
apply (*cases q-auth1*)
subgoal for *q-drop* **by** (*erule S-il.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look* **by** (*erule S-il.cases*) (*simp-all del*: *bind-spmf-const add*: *pair-spmf-alt-def*, *clarsimp+*)
subgoal for *q-fedit* **by** (*erule S-il.cases*) (*simp-all del*: *bind-spmf-const add*: *pair-spmf-alt-def*, *clarsimp+*)
done
done
subgoal for *q-auth2*
apply (*cases q-auth2*)
subgoal for *q-drop* **by** (*erule S-il.cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look* **by** (*erule S-il.cases*) (*simp-all del*: *bind-spmf-const add*: *pair-spmf-alt-def*, *clarsimp+*)
subgoal for *q-fedit* **by** (*erule S-il.cases*) (*simp-all del*: *bind-spmf-const add*: *pair-spmf-alt-def*, *clarsimp+*)
done
done
done
subgoal *AState-OK* **for** *sl sr query*
apply (*cases query*)
subgoal for *q-auth1*
apply (*cases q-auth1*)
subgoal for *q-drop* **by** (*erule S-il.cases*) *auto*
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look*
proof (*erule S-il.cases*, *goal-cases*)
case (*2 s-act1 s-act2*) — Corresponds to *sil-1-0*
then show *?case*
apply *simp*
apply (*subst* (*1 2 3*) *bind-bind-conv-pair-spmf*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst* (*1 2*) *cond-spmf-fst-pair-spmf1*[*unfolded map-prod-def split-def*])
by (*auto intro*: *trace-eq-simcl-bindI S-il.intros*)
next
case (*7 s-act1 s-act2*) — Corresponds to *sil-1-1*
then show *?case*

apply *simp*
apply (*subst* (*1 2 3*) *bind-bind-conv-pair-spmf*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst* (*1 2*) *cond-spmf-fst-pair-spmf1*[*unfolded map-prod-def split-def*])
apply(*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *simp add*: *inj-on-def*)
apply (*subst* (*1 2*) *inv-into-f-f*)
apply (*simp-all add*: *inj-on-def map-spmf-conv-bind-spmf pair-spmf-alt-def isample-key-conv-nat*)
apply (*rule trace-eq-simcl-bindI*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros*)
next
case (*14 s-act1 s-act2 s-actk*) — Corresponds to *sil-1-2*
then show *?case*
apply *clarsimp*
apply (*subst bind-commute-spmf*, *subst* (*2*) *bind-commute-spmf*)
apply (*subst* (*1 2 3 4*) *bind-bind-conv-pair-spmf*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst* (*1 2*) *cond-spmf-fst-pair-spmf1*[*unfolded map-prod-def split-def*])
apply(*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *simp add*: *inj-on-def*)
apply (*subst* (*1 2*) *inv-into-f-f*)
apply (*simp-all add*: *inj-on-def map-spmf-conv-bind-spmf pair-spmf-alt-def isample-key-conv-nat*)
apply (*subst* (*1 2*) *bind-bind-conv-pair-spmf*)
by (*auto intro*!: *trace-eq-simcl-bindI S-il.intros*)
next
case (*20 s-act1 s-act2 s-actk k z*) — Corresponds to *sil-1-3*
then show *?case*
apply *simp*
apply (*subst bind-bind-conv-pair-spmf*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst* (*1 2*) *cond-spmf-fst-pair-spmf1*[*unfolded map-prod-def split-def*])
apply(*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *simp add*: *inj-on-def*)
apply (*subst* (*1 2*) *inv-into-f-f*)
apply (*simp-all add*: *inj-on-def map-spmf-conv-bind-spmf*)
by (*auto intro*!: *trace-eq-simcl-bindI S-il.intros*)
qed (*auto simp add*: *map-spmf-conv-bind-spmf*,
auto simp add: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *S-il.intros trace-eq-simcl.base*)
subgoal for *q-fedit*
proof (*erule S-il.cases*, *goal-cases*)
case (*4 s-act1 x s-act2*) — Corresponds to *sil-1′-0*
then show *?case*
apply *simp*
apply (*subst bind-bind-conv-pair-spmf*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])

```
        by (auto intro: S-il.intros trace-eq-simcl.base)
      next
        case (10 s-act1 s-act2 x y) — Corresponds to sil-1c-1c
        then show ?case
          apply (clarsimp simp add: pair-spmf-alt-def isample-key-conv-nat)
          apply (simp add: map-spmf-conv-bind-spmf[symmetric])
          by (auto intro!: trace-eq-simcl-map S-il.intros)
      qed (auto simp add: map-spmf-conv-bind-spmf[symmetric],
          auto intro: S-il.intros trace-eq-simcl.base trace-eq-simcl-map)
      done
    done
  subgoal for q-auth2
    apply (cases q-auth2)
    subgoal for q-drop by (erule S-il.cases) auto
    subgoal for q-lfe
      apply (cases q-lfe)
      subgoal for q-look
      proof (erule S-il.cases, goal-cases)
        case (6 s-act2 s-act1) — Corresponds to sil-0-1
        then show ?case
          apply clarsimp
          apply (subst (1 2) bind-commute-spmf)
          apply (subst (1 3) bind-bind-conv-pair-spmf)
          apply (subst bind-bind-conv-pair-spmf)
          apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
            apply (subst (1 2) cond-spmf-fst-pair-spmf1[unfolded map-prod-def
split-def])
          by (auto intro: trace-eq-simcl-bindI S-il.intros)
      next
        case (7 s-act1 s-act2) — Corresponds to sil-1-1
        then show ?case
          apply clarsimp
          apply (subst (1 2) bind-commute-spmf)
          apply (subst (1 3) bind-bind-conv-pair-spmf)
          apply (subst bind-bind-conv-pair-spmf)
          apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
            apply (subst (1 2) cond-spmf-fst-pair-spmf1[unfolded map-prod-def
split-def])
          apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
          apply (subst (1 2) inv-into-f-f)
            apply (simp-all add: inj-on-def map-spmf-conv-bind-spmf)
          apply (subst pair-spmf-alt-def)
          apply (subst bind-spmf-assoc)
          apply (rule trace-eq-simcl-bindI)
              by (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro:
S-il.intros)
      next
        case (8 s-act1 s-act2 s-actk) — Corresponds to sil-2-1
        then show ?case
```

```
apply clarsimp
apply (subst (2) bind-commute-spmf, subst (1 3) bind-commute-spmf)
apply (subst (2) bind-commute-spmf)
apply (subst (2 4) bind-bind-conv-pair-spmf)
apply (clarsimp simp add: bind-bind-conv-pair-spmf)
apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
apply (subst (1 2) cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
apply (subst (1 2) inv-into-f-f)
apply (simp-all add: inj-on-def map-spmf-conv-bind-spmf)
apply (simp add: pair-spmf-alt-def isample-key-conv-nat)
apply (subst (1 2) bind-bind-conv-pair-spmf)
by (auto intro!: trace-eq-simcl-bindI S-il.intros)
next
case (9 s-act1 s-act2 s-actk k z) — Corresponds to sil-3-1
then show ?case
apply (clarsimp simp del: bind-spmf-const simp add: pair-spmf-alt-def)
apply (subst (1 2) bind-commute-spmf)
apply (subst (1 2) bind-bind-conv-pair-spmf)
apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
apply (subst (1 2) cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
apply (subst (1 2) inv-into-f-f)
apply (simp-all add: inj-on-def map-spmf-conv-bind-spmf)
by (auto intro: trace-eq-simcl-bindI S-il.intros)
qed (auto simp del: bind-spmf-const simp add: map-spmf-conv-bind-spmf,
auto simp add: map-spmf-conv-bind-spmf[symmetric] intro: S-il.intros trace-eq-simcl.base)
subgoal for q-fedit
proof (erule S-il.cases, goal-cases)
case (10 s-act1 s-act2 x y) — Corresponds to sil-1c-1c
then show ?case
apply simp
apply (clarsimp simp add: map-spmf-conv-bind-spmf pair-spmf-alt-def isample-key-conv-nat)
apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
by (auto intro!: S-il.intros trace-eq-simcl-map)
next
case (26 s-act2 y s-act1) — Corresponds to sil-0-1'
then show ?case
apply simp
apply (subst bind-bind-conv-pair-spmf)
apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
by (auto intro: S-il.intros trace-eq-simcl.base)
qed (auto simp add: map-spmf-conv-bind-spmf[symmetric],
auto intro: S-il.intros trace-eq-simcl.base trace-eq-simcl-map)
done
```

```
      done
    done
  subgoal UOut-OK for sl sr query
    apply (cases query)
    subgoal for q-alice
      apply (erule S-il.cases)
      by (auto simp add: pair-spmf-alt-def isample-key-conv-nat)
    subgoal for q-bob
      apply (erule S-il.cases)
      by (auto simp add: pair-spmf-alt-def isample-key-conv-nat)
    done
  subgoal UState-OK for sl sr query
    apply (cases query)
    subgoal for q-alice
    proof (erule S-il.cases, goal-cases)
      case (14 s-act1 s-act2 s-actk) — Corresponds to sil-1-2
      then show ?case
        apply (clarsimp)
        apply (subst (2) bind-commute-spmf, subst bind-commute-spmf)
        apply (subst bind-bind-conv-pair-spmf, subst bind-bind-conv-pair-spmf)
        apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
        apply (subst cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
        apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
        apply(subst (1 2) inv-into-f-f)
        by (auto simp add: inj-on-def intro: S-il.intros trace-eq-simcl.base)
    next
      case (15 s-act1 s-act2 s-actk) — Corresponds to sil-2-2
      then show ?case
        apply (clarsimp)
        apply (subst (2) bind-commute-spmf, subst bind-commute-spmf)
        apply (subst bind-bind-conv-pair-spmf, subst bind-bind-conv-pair-spmf)
        apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
        apply (subst cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
        apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
        apply(subst (1 2) inv-into-f-f)
        by (auto simp add: inj-on-def intro: S-il.intros trace-eq-simcl.base)
    qed (auto simp add: map-spmf-conv-bind-spmf[symmetric],
        auto intro: S-il.intros trace-eq-simcl.base trace-eq-simcl-map)
    subgoal for q-bob
    proof (erule S-il.cases, goal-cases)
      case (8 s-act1 s-act2 s-actk) — Corresponds to sil-2-1
      then show ?case
        apply clarsimp
        apply (subst (2) bind-commute-spmf, subst bind-commute-spmf)
       apply (subst (2) bind-bind-conv-pair-spmf, subst bind-bind-conv-pair-spmf)
        apply (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric])
        apply (subst cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
        apply (subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
        apply (subst (1 2) inv-into-f-f)
```

by (*auto simp add*: *inj-on-def intro*: *S-il.intros trace-eq-simcl.base*)
next
case (*15 s-act1 s-act2 s-actk*) — Corresponds to *sil-2-2*
then show *?case*
apply *clarsimp*
apply (*subst* (*2*) *bind-commute-spmf*, *subst bind-commute-spmf*)
apply (*subst* (*2*) *bind-bind-conv-pair-spmf*, *subst bind-bind-conv-pair-spmf*)
apply (*clarsimp simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply (*subst cond-spmf-fst-pair-spmf1*[*unfolded map-prod-def split-def*])
apply (*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *simp add*: *inj-on-def*)
apply (*subst* (*1 2*) *inv-into-f-f*)
by (*auto simp add*: *inj-on-def intro*: *S-il.intros trace-eq-simcl.base*)
qed(*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*],
auto intro: *S-il.intros trace-eq-simcl.base trace-eq-simcl-map*)
done
done
qed

lemma *connect-ideal*: *connect D* (*obsf-resource ideal-resource*) =
connect D (*obsf-resource* (*RES* (*fused-resource.fuse* (*lazy-core DH1-sample*) *lazy-rest*)
(*basic-core-sinit*, *basic-rest-sinit*)))
proof −
have *fact1*: *trace-rest-eq ideal-rest′ ideal-rest′ UNIV UNIV s s* **for** *s*
by (*rule rel-rest′-into-trace-rest-eq*[**where** *S*=(=) **and** *M*=(=)]) (*simp-all add*:
eq-onp-def rel-rest′-eq)

have *fact2*: $\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full* $\vdash c$ *callee-of-rest ideal-rest′ s* $\surd$ **for** *s*
by (*rule WT-calleeI*) (*cases s*, *case-tac call*, *rename-tac* [!] *x*, *case-tac* [!] *x*,
auto)

have *fact3*: $\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\vdash c$ *callee-of-core ideal-core′ s* $\surd$ **for** *s*
by (*rule WT-calleeI*) (*cases s*, *case-tac call*, *rename-tac* [!] *x*, *case-tac* [!] *x*,
auto)

have *fact4*: $\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*) $\vdash c$ *callee-of-core* (*lazy-core xyz*) *s* $\surd$ **for**
xyz s
by (*rule WT-calleeI*) (*cases s*, *case-tac call*, *rename-tac* [!] *x*, *case-tac* [!] *x*,
auto)

show *?thesis*
apply (*rule connect-cong-trace*[**where** *A*=*UNIV* **and** $\mathcal{I}$=$\mathcal{I}$*-full*])
apply (*rule trace-eq-obsf-resourceI*)
subgoal
apply (*simp add*: *attach-ideal*)
apply (*rule fuse-trace-eq*[**where** $\mathcal{I}E$=$\mathcal{I}$*-full* **and** $\mathcal{I}CA$=$\mathcal{I}$*-full* **and** $\mathcal{I}CU$=$\mathcal{I}$*-full*
and $\mathcal{I}RA$=$\mathcal{I}$*-full* **and** $\mathcal{I}RU$=$\mathcal{I}$*-full*, *simplified*])
by (*simp-all add*: *ideal-s-rest′-def lazy-rest-def trac-eq-core-il*[*simplified*] *fact1*
fact2 fact3 fact4)
by (*simp-all add*: *attach-ideal*)

qed

end

12.9 Proving the trace-equivalence of simplified Real and Lazy constructions

context
begin

private abbreviation *rsample-nat* ≡ *sample-uniform* (*order* $\mathcal{G}$)
private abbreviation *rsample-pair-nn* ≡ *pair-spmf rsample-nat rsample-nat*

private inductive *S-rl* :: ((*unit* × *'grp cstate* × *'grp cstate*) × *'grp auth.state* × *'grp auth.state*) *spmf*
⇒ ((*'grp st-state* × *'grp cstate* × *'grp cstate*) × *'grp auth.state* × *'grp auth.state*) *spmf* ⇒ *bool*
where
— (Auth1 =a)@(Auth2 =0)
srl-0-0: *S-rl* (*return-spmf* (((), *CState-Void*, *CState-Void*), (*auth.State-Void*, *s-act1*), *auth.State-Void*, *s-act2*))
(*return-spmf* ((*None*, *CState-Void*, *CState-Void*), (*auth.State-Void*, *s-act1*), (*auth.State-Void*, *s-act2*)))
— ../(Auth1 =a)@(Auth2 =0) # wl
| *srl-1-0*: *S-rl* (*map-spmf* (λ*x*. (((), *CState-Half x*, *CState-Void*), (*auth.State-Store* (**g** [^] *x*), *s-act1*), *auth.State-Void*, *s-act2*)) *rsample-nat*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Void*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Void*, *s-act2*))
| *srl-2-0*: *S-rl* (*map-spmf* (λ*x*. (((), *CState-Half x*, *CState-Void*), (*auth.State-Collect* (**g** [^] *x*), *s-act1*), *auth.State-Void*, *s-act2*)) *rsample-nat*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Void*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Void*, *s-act2*))
— ../(Auth1 =a)@(Auth2 =0) # look
| *srl-1′-0*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Void*), (*auth.State-Store* (**g** [^] *x*), *s-act1*), *auth.State-Void*, *s-act2*))
(*map-spmf* (λ*y*. ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Void*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Void*, *s-act2*)) *rsample-nat*)
| *srl-2′-0*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Void*), (*auth.State-Collect* (**g** [^] *x*), *s-act1*), *auth.State-Void*, *s-act2*))
(*map-spmf* (λ*y*. ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Void*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Void*, *s-act2*)) *rsample-nat*)
— (Auth1 =a)@(Auth2 =1)
| *srl-0-1*: *S-rl* (*map-spmf* (λ*y*. (((), *CState-Void*, *CState-Half y*), (*auth.State-Void*, *s-act1*), *auth.State-Store* (**g** [^] *y*), *s-act2*)) *rsample-nat*)
(*return-spmf* ((*None*, *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Store* **1**, *s-act2*))
— ../(Auth1 =a)@(Auth2 =1) # wl
| *srl-1-1*: *S-rl* (*map-spmf* (λ*yx*. (((), *CState-Half* (*snd yx*), *CState-Half* (*fst yx*)),

(*auth.State-Store* (**g** [^] *snd yx*), *s-act1*), *auth.State-Store* (**g** [^] *fst yx*), *s-act2*)) *rsample-pair-nn*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*))
| *srl-2-1*: *S-rl* (*map-spmf* (λ*yx*. (((), *CState-Half* (*snd yx*), *CState-Half* (*fst yx*)), (*auth.State-Collect* (**g** [^] *snd yx*), *s-act1*), *auth.State-Store* (**g** [^] *fst yx*), *s-act2*)) *rsample-pair-nn*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*))
— ../(Auth1 =a)@(Auth2 =1) # look
| *srl-1c-1c*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Half y*), (*auth.State-Store* (**g** [^] *x*), *s-act1*), *auth.State-Store* (**g** [^] *y*), *s-act2*))
(*return-spmf* ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*))
| *srl-2c-1c*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Half y*), (*auth.State-Collect* (**g** [^] *x*), *s-act1*), *auth.State-Store* (**g** [^] *y*), *s-act2*))
(*return-spmf* ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Store* **1**, *s-act2*))
| *srl-3c-1c*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Full* (*y*, *z*)), (*auth.State-Collected*, *s-act1*), *auth.State-Store* (**g** [^] *y*), *s-act2*))
(*return-spmf* ((*Some* (*z*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Store* **1**, *s-act2*))
if *z* = (**g** [^] *x*) [^] *y*
— (Auth1 =a)@(Auth2 =2)
| *srl-0-2*: *S-rl* (*map-spmf* (λ*y*. (((), *CState-Void*, *CState-Half y*), (*auth.State-Void*, *s-act1*), *auth.State-Collect* (**g** [^] *y*), *s-act2*)) *rsample-nat*)
(*return-spmf* ((*None*, *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
— ../(Auth1 =a)@(Auth2 =2) # wl
| *srl-1-2*: *S-rl* (*map-spmf* (λ*yx*. (((), *CState-Half* (*snd yx*), *CState-Half* (*fst yx*)), (*auth.State-Store* (**g** [^] *snd yx*), *s-act1*), *auth.State-Collect* (**g** [^] *fst yx*), *s-act2*)) *rsample-pair-nn*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
| *srl-2-2*: *S-rl* (*map-spmf* (λ*yx*. (((), *CState-Half* (*snd yx*), *CState-Half* (*fst yx*)), (*auth.State-Collect* (**g** [^] *snd yx*), *s-act1*), *auth.State-Collect* (**g** [^] *fst yx*), *s-act2*)) *rsample-pair-nn*)
(*return-spmf* ((*None*, *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
— ../(Auth1 =a)@(Auth2 =2) # look
| *srl-1c-2c*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Half y*), (*auth.State-Store* (**g** [^] *x*), *s-act1*), *auth.State-Collect* (**g** [^] *y*), *s-act2*))
(*return-spmf* ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
| *srl-2c-2c*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Half y*), (*auth.State-Collect* (**g** [^] *x*), *s-act1*), *auth.State-Collect* (**g** [^] *y*), *s-act2*))
(*return-spmf* ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
| *srl-3c-2c*: *S-rl* (*return-spmf* (((), *CState-Half x*, *CState-Full* (*y*, *z*)), (*auth.State-Collected*,

s-act1), *auth.State-Collect* (**g** [^] *y*), *s-act2*))
  (*return-spmf* ((*Some* (*z*, **g** [^] *x*, **g** [^] *y*), *CState-Half 0*, *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Collect* **1**, *s-act2*))
 if *z* = (**g** [^] *x*) [^] *y*
— (Auth1 =a)@(Auth2 =3)
 | *srl-1c-3c*: *S-rl* (*return-spmf* (((), *CState-Full* (*x*, *z*), *CState-Half y*), (*auth.State-Store* (**g** [^] *x*), *s-act1*), *auth.State-Collected*, *s-act2*))
  (*return-spmf* ((*Some* (*z*, **g** [^] *x*, **g** [^] *y*), *CState-Full* (*0*, **1**), *CState-Half 0*), (*auth.State-Store* **1**, *s-act1*), *auth.State-Collected*, *s-act2*))
 if *z* = (**g** [^] *y*) [^] *x*
 | *srl-2c-3c*: *S-rl* (*return-spmf* (((), *CState-Full* (*x*, *z*), *CState-Half y*), (*auth.State-Collect* (**g** [^] *x*), *s-act1*), *auth.State-Collected*, *s-act2*))
  (*return-spmf* ((*Some* (*z*, **g** [^] *x*, **g** [^] *y*), *CState-Full* (*0*, **1**), *CState-Half 0*), (*auth.State-Collect* **1**, *s-act1*), *auth.State-Collected*, *s-act2*))
 if *z* = (**g** [^] *y*) [^] *x*
 | *srl-3c-3c*: *S-rl* (*return-spmf* (((), *CState-Full* (*x*, *z*), *CState-Full* (*y*, *z*)), (*auth.State-Collected*, *s-act1*), *auth.State-Collected*, *s-act2*))
  (*return-spmf* ((*Some* (*z*, **g** [^] *x*, **g** [^] *y*), *CState-Full* (*0*, **1**), *CState-Full* (*0*, **1**)), (*auth.State-Collected*, *s-act1*), *auth.State-Collected*, *s-act2*))
 if *z* = (**g** [^] *y*) [^] *x*
— (Auth1 =0)@(Auth2 =1')
 | *srl-0-1'*: *S-rl* (*return-spmf* (((), *CState-Void*, *CState-Half y*), (*auth.State-Void*, *s-act1*), *auth.State-Store* (**g** [^] *y*), *s-act2*))
  (*map-spmf* (λ*x*. ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Store* **1**, *s-act2*)) *rsample-nat*)
— (Auth1 =0)@(Auth2 =2')
 | *srl-0-2'*: *S-rl* (*return-spmf* (((), *CState-Void*, *CState-Half y*), (*auth.State-Void*, *s-act1*), *auth.State-Collect* (**g** [^] *y*), *s-act2*))
  (*map-spmf* (λ*x*. ((*Some* ((**g** [^] *x*) [^] *y*, **g** [^] *x*, **g** [^] *y*), *CState-Void*, *CState-Half 0*), (*auth.State-Void*, *s-act1*), *auth.State-Collect* **1**, *s-act2*)) *rsample-nat*)

private lemma *trac-eq-core-rl*: *trace-core-eq real-core'* (*basic-core DH0-sample*)
 (*UNIV* <+> *UNIV*) ((*UNIV* <+> *UNIV* <+> *UNIV*) <+> *UNIV* <+> *UNIV* <+> *UNIV*) ((*UNIV* <+> *UNIV*) <+> *UNIV* <+> *UNIV*)
 (*return-spmf real-s-core'*) (*return-spmf basic-core-sinit*)
proof −
 have *power-commute*: (**g** [^] *x*) [^] (*y* :: *nat*) = (**g** [^] *y*) [^] (*x* :: *nat*) **for** *x y*
  by (*simp add*: *nat-pow-pow mult.commute*)

 have [*simp*]: *weight-spmf rsample-nat* = *1*
  by (*simp add*: *finite-carrier order-gt-0-iff-finite*)

 have [*simp*]: *mk-lossless rsample-nat* = *rsample-nat*
  by (*simp add*: *mk-lossless-def*)

 have [*simp*]: *mk-lossless rsample-pair-nn* = *rsample-pair-nn*
  by (*simp add*: *mk-lossless-def*)

note [*simp*] = *basic-core-oracle-usr-def basic-core-helper-def exec-gpv-bind spmf.map-comp map-bind-spmf bind-map-spmf bind-spmf-const o-def Let-def split-def*

show *?thesis*
apply (*rule trace-core-eq-simI-upto*[**where** *S*=*S-rl*])
subgoal *Init-OK*
by (*simp add*: *real-s-core′-def srl-0-0*)
subgoal *POut-OK* **for** *s-l s-r query*
apply (*cases query*)
subgoal for *e-auth1* **by** (*cases e-auth1*; *erule S-rl.cases*; *auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *split*!: *if-splits*)
subgoal for *e-auth2* **by** (*cases e-auth2*; *erule S-rl.cases*; *auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *split*!: *if-splits*)
done
subgoal *PState-OK* **for** *s-l s-r query*
apply (*cases query*)
subgoal for *e-auth1* **by**(*cases e-auth1*; *erule S-rl.cases*; *auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *split*!: *if-splits intro*: *S-rl.intros trace-eq-simcl.base*)
subgoal for *e-auth2* **by** (*cases e-auth2*; *erule S-rl.cases*; *auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *split*!: *if-splits intro*: *S-rl.intros trace-eq-simcl.base*)
done
subgoal *AOut-OK* **for** *sl sr q*
apply (*cases q*)
subgoal for *q-auth1*
apply (*cases q-auth1*)
subgoal for *q-drop* **by** (*erule S-rl.cases*; *simp*)
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look* **by**(*erule S-rl.cases*; *auto simp add*: *DH0-sample-def pair-spmf-alt-def*)
subgoal for *q-fedit* **by** (*cases q-fedit*; *erule S-rl.cases*; *auto simp add*: *DH0-sample-def pair-spmf-alt-def*)
done
done
subgoal for *q-auth2*
apply (*cases q-auth2*)
subgoal for *q-drop* **by** (*erule S-rl.cases*; *simp*)
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look* **by**(*erule S-rl.cases*; *auto simp add*: *DH0-sample-def pair-spmf-alt-def*)
subgoal for *q-fedit* **by** (*cases q-fedit*; *erule S-rl.cases*; *auto simp add*: *DH0-sample-def pair-spmf-alt-def*)
done
done
done
subgoal *AState-OK* **for** *sl sr q s1 s2 s1′ s2′ oa*
apply (*cases q*)

```
subgoal for q-auth1
  apply (cases q-auth1)
  subgoal for q-drop by (erule S-rl.cases; simp)
  subgoal for q-lfe
    apply (cases q-lfe)
    subgoal for q-look
    proof (erule S-rl.cases, goal-cases)
      case (2 s-act1 s-act2) — Corresponds to srl-1-0
      then show ?case
        apply(cases s1′)
        apply (clarsimp simp add: DH0-sample-def)
        apply(simp add: bind-bind-conv-pair-spmf)
        apply(simp add: map-spmf-conv-bind-spmf[symmetric])
        apply (subst cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
        apply(simp)
        apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
        by(subst (1 2 3 4) inv-into-f-f; simp add: inj-on-def trace-eq-simcl.base S-rl.intros)
    next
      case (4 x s-act1 s-act2) — Corresponds to srl-1′-0
      then show ?case
      by(auto simp add: DH0-sample-def map-spmf-conv-bind-spmf[symmetric] intro!: trace-eq-simcl.base S-rl.intros)
    next
      case (7 s-act1 s-act2) — Corresponds to srl-1-1
      then show ?case
        apply(clarsimp simp add: DH0-sample-def pair-spmf-alt-def)
        apply(subst bind-commute-spmf)
        apply(simp add: bind-bind-conv-pair-spmf)
        apply(simp add: map-spmf-conv-bind-spmf[symmetric])
        apply (subst (1 2) cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
        apply(simp)
        apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
        by (subst (1 2 3 4) inv-into-f-f; simp add: inj-on-def trace-eq-simcl-map S-rl.intros)
    next
      case (13 s-act1 s-act2) — Corresponds to srl-1-2
      then show ?case
        apply(clarsimp simp add: DH0-sample-def pair-spmf-alt-def)
        apply(subst bind-commute-spmf)
        apply(simp add: bind-bind-conv-pair-spmf)
        apply(simp add: map-spmf-conv-bind-spmf[symmetric])
        apply (subst (1 2) cond-spmf-fst-pair-spmf1[unfolded map-prod-def split-def])
        apply(simp)
        apply(subst (1 2) cond-spmf-fst-map-Pair1; simp add: inj-on-def)
        by(subst (1 2 3 4) inv-into-f-f; simp add: inj-on-def trace-eq-simcl-map S-rl.intros)
```

qed (*auto intro*: *S-rl.intros*)
subgoal for *q-fedit*
apply (*cases q-fedit*)
by (*erule S-rl.cases*, *goal-cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base intro*: *S-rl.intros*)
done
done
subgoal for *q-auth2*
apply (*cases q-auth2*)
subgoal for *q-drop* **by** (*erule S-rl.cases*; *simp*)
subgoal for *q-lfe*
apply (*cases q-lfe*)
subgoal for *q-look*
proof (*erule S-rl.cases*, *goal-cases*)
case (*6 s-act1 s-act2*) — Corresponds to *srl-0-1*
then show *?case*
apply(*clarsimp simp add*: *DH0-sample-def*)
apply(*subst bind-commute-spmf*)
apply(*simp add*: *bind-bind-conv-pair-spmf*)
apply(*simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply(*subst cond-spmf-fst-pair-spmf1*[*simplified map-prod-def split-def*])
apply(*simp*)
apply(*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *simp add*: *inj-on-def*)
by(*subst* (*1 2 3 4*) *inv-into-f-f*; *simp add*: *inj-on-def trace-eq-simcl.base S-rl.intros*)
next
case (*7 s-act1 s-act2*) — Corresponds to *srl-1-1*
then show *?case*
apply(*clarsimp simp add*: *DH0-sample-def*)
apply(*subst bind-commute-spmf*)
apply(*simp add*: *bind-bind-conv-pair-spmf*)
apply(*simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply(*subst* (*1 2*) *cond-spmf-fst-pair-spmf1*[*simplified map-prod-def split-def*])
apply(*simp*)
apply(*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *simp add*: *inj-on-def*)
by(*subst* (*1 2 3 4*) *inv-into-f-f*; *simp add*: *inj-on-def trace-eq-simcl-map S-rl.intros*)
next
case (*8 s-act1 s-act2*) — Corresponds to *srl-2-1*
then show *?case*
apply(*clarsimp simp add*: *DH0-sample-def*)
apply(*subst bind-commute-spmf*)
apply(*simp add*: *bind-bind-conv-pair-spmf*)
apply(*simp add*: *map-spmf-conv-bind-spmf*[*symmetric*])
apply(*subst* (*1 2*) *cond-spmf-fst-pair-spmf1*[*simplified map-prod-def split-def*])
apply(*simp*)
apply(*subst* (*1 2*) *cond-spmf-fst-map-Pair1*; *simp add*: *inj-on-def*)

by(*subst* (*1 2 3 4*) *inv-into-f-f*; *simp add*: *inj-on-def trace-eq-simcl-map S-rl.intros*)
next
case (*21 y s-act1 s-act2*) — Corresponds to *srl-0-1′*
then show *?case*
by(*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base intro*: *S-rl.intros*)
qed (*auto intro*: *S-rl.intros*)
subgoal for *q-fedit*
apply (*cases q-fedit*)
by (*erule S-rl.cases*, *goal-cases*) (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*!: *trace-eq-simcl.base intro*: *S-rl.intros*)
done
done
done
subgoal *UOut-OK* **for** *sl sr q*
apply (*cases q*)
subgoal for *q-usr*
apply (*cases q-usr*)
subgoal for *q-alice* **by** (*erule S-rl.cases*; *simp add*: *DH0-sample-def pair-spmf-alt-def power-commute*)
subgoal for *q-bob* **by** (*erule S-rl.cases*; *auto simp add*: *bind-bind-conv-pair-spmf apfst-def DH0-sample-def power-commute split*!: *if-split*)
done
subgoal for *q-act*
apply (*cases q-act*)
subgoal for *q-alice*
by (*erule S-rl.cases*; *auto simp add*: *left-gpv-bind-gpv exec-gpv-parallel-oracle-left map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf intro*!: *bind-spmf-cong*)
subgoal for *q-bob*
by (*erule S-rl.cases*; *auto simp add*: *right-gpv-bind-gpv exec-gpv-parallel-oracle-right map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf intro*!: *bind-spmf-cong*)
done
done
subgoal *UState-OK* **for** *sl sr q*
apply (*cases q*)
subgoal for *q-usr*
apply (*cases q-usr*)
subgoal for *q-alice*
proof (*erule S-rl.cases*, *goal-cases*)
case (*13 s-act1 s-act2*) — Corresponds to *srl-1-2*
then show *?case*
apply(*clarsimp simp add*: *DH0-sample-def pair-spmf-alt-def*)
apply(*subst* (*1*) *bind-commute-spmf*)
apply(*simp add*: *bind-bind-conv-pair-spmf*)
apply(*subst* (*1 2*) *cond-spmf-fst-bind*)
by (*auto simp add*: *power-commute intro*!: *trace-eq-simcl-bind S-rl.intros*)
next
case (*14 s-act1 s-act2*) — Corresponds to *srl-2-2*

then show *?case*
apply(*clarsimp simp add*: *DH0-sample-def pair-spmf-alt-def*)
apply(*subst* (*1*) *bind-commute-spmf*)
apply(*simp add*: *bind-bind-conv-pair-spmf*)
apply(*subst* (*1 2*) *cond-spmf-fst-bind*)
by (*auto simp add*: *power-commute intro*!: *trace-eq-simcl-bind S-rl.intros*)
qed (*auto intro*: *S-rl.intros*)
subgoal for *q-bob*
proof (*erule S-rl.cases*, *goal-cases*)
case (*8 s-act1 s-act2*) — Corresponds to *srl-2-1*
then show *?case*
apply(*clarsimp simp add*: *DH0-sample-def*)
apply(*subst bind-commute-spmf*)
apply(*simp add*: *bind-bind-conv-pair-spmf power-commute*)
apply(*subst* (*1 2*) *cond-spmf-fst-bind*)
by (*auto simp add*: *power-commute intro*!: *trace-eq-simcl-bind S-rl.intros*)
next
case (*14 s-act1 s-act2*) — Corresponds to *srl-2-2*
then show *?case*
apply(*clarsimp simp add*: *DH0-sample-def*)
apply(*subst bind-commute-spmf*)
apply(*simp add*: *bind-bind-conv-pair-spmf power-commute*)
apply(*subst* (*1 2*) *cond-spmf-fst-bind*)
by (*auto simp add*: *power-commute intro*!: *trace-eq-simcl-bind S-rl.intros*)
qed (*auto simp add*: *power-commute intro*: *S-rl.intros*)
done
subgoal for *q-act*
apply (*cases q-act*)
subgoal for *a-alice*
proof (*erule S-rl.cases*, *goal-cases*)
case (*1 s-act1 s-act2*) — Corresponds to *srl-0-0*
then show *?case*
apply (*simp add*: *left-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl.base S-rl.intros*)
next
case (*6 s-act1 s-act2*) — Corresponds to *srl-0-1*
then show *?case*
apply (*simp add*: *left-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
apply (*subst bind-bind-conv-pair-spmf*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl.base S-rl.intros*)
next
case (*12 s-act1 s-act2*) — Corresponds to *srl-0-2*
then show *?case*
apply (*simp add*: *left-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)

apply (*subst bind-bind-conv-pair-spmf*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl.base S-rl.intros*)
next
case (*21 y s-act1 s-act2*) — Corresponds to *srl-0-2′*
then show *?case*
apply (*simp add*: *left-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl-map S-rl.intros*)
next
case (*22 y s-act1 s-act2*) — Corresponds to *srl-0-1′*
then show *?case*
apply (*simp add*: *left-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl-map S-rl.intros*)
qed (*simp-all split*!: *if-splits*)
subgoal for *a-bob*
proof (*erule S-rl.cases*, *goal-cases*)
case (*1 s-act1 s-act2*) — Corresponds to *srl-0-0*
then show *?case*
apply(*clarsimp simp add*: *right-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl.base S-rl.intros*)
next
case (*2 s-act1 s-act2*) — Corresponds to *srl-1-0*
then show *?case*
apply(*clarsimp simp add*: *right-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
apply (*subst bind-commute-spmf*, *subst bind-bind-conv-pair-spmf*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl.base S-rl.intros*)
next
case (*3 s-act1 s-act2*) — Corresponds to *srl-2-0*
then show *?case*
apply(*clarsimp simp add*: *right-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
apply (*subst bind-commute-spmf*, *subst bind-bind-conv-pair-spmf*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl.base S-rl.intros*)
next
case (*4 x s-act1 s-act2*) — Corresponds to *srl-1′-0*
then show *?case*
apply(*clarsimp simp add*: *right-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl-map S-rl.intros*)

next
case (*5 x s-act1 s-act2*) — Corresponds to *srl-2′-0*
then show *?case*
apply(*clarsimp simp add*: *right-gpv-bind-gpv pair-spmf-alt-def map-gpv-bind-gpv gpv.map-id map-gpv′-bind-gpv map′-lift-spmf split*!: *if-splits*)
by (*auto simp add*: *map-spmf-conv-bind-spmf*[*symmetric*] *intro*: *trace-eq-simcl-map S-rl.intros*)
qed (*simp-all split*!: *if-splits*)
done
done
done
qed

lemma *trace-eq-fuse-rl*: *UNIV* $\vdash_R$ *1*$_C$ $|_=$ *rassocl*$_C$ $\rhd$ *RES* (*fused-resource.fuse real-core′ real-rest′*) (*real-s-core′*, *real-s-rest′*)
≈ *RES* (*fused-resource.fuse* (*lazy-core DH0-sample*) *lazy-rest*) (*basic-core-sinit*, *basic-rest-sinit*)
proof −
have *fact1*: *UNIV* $\vdash_R$ *1*$_C$ $|_=$ *rassocl*$_C$ $\rhd$ *RES* (*fused-resource.fuse* (*basic-core DH0-sample*) *basic-rest*) (*basic-core-sinit*, *basic-rest-sinit*) ∼
RES (*fused-resource.fuse* (*lazy-core DH0-sample*) *lazy-rest*) (*basic-core-sinit*, *basic-rest-sinit*)
proof −
have [*simp*]: $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) ⊢*res RES* (*fused-resource.fuse* (*basic-core DH0-sample*) *basic-rest*) (*basic-core-sinit*, *basic-rest-sinit*) √ **for** *s*
apply (*rule WT-resource-of-oracle*, *rule WT-calleeI*)
by (*case-tac call*, *rename-tac* [!] *x*, *case-tac* [!] *x*, *rename-tac* [!] *y*, *case-tac* [!] *y*)
(*auto simp add*: *fused-resource.fuse.simps parallel-eoracle-def*)

note [*simp*] = *exec-gpv-bind spmf.map-comp o-def map-bind-spmf bind-map-spmf bind-spmf-const*

show *?thesis*
apply(*subst attach-wiring-resource-of-oracle*)
apply(*rule wiring-parallel-converter2 wiring-id-converter*[**where** $\mathcal{I}$=$\mathcal{I}$*-full*] *wiring-rassocl*[*of* $\mathcal{I}$*-full* $\mathcal{I}$*-full* $\mathcal{I}$*-full*])+
apply *simp-all*
apply (*rule eq-resource-on-resource-of-oracleI*[**where** *S*=(=)])
apply(*simp-all add*: *eq-on-def relator-eq*)
apply(*rule ext*)+
apply(*subst fuse-ishift-core-to-rest*[**where** *core*=*basic-core DH0-sample* **and** *rest*=*basic-rest* **and** *core′*=*lazy-core DH0-sample* **and**
rest′=*lazy-rest* **and** *fn*=*basic-core-helper* **and** *h-out*=*map-sum* (λ-. *Out-Activation-Alice*) (λ-. *Out-Activation-Bob*), *simplified*])
apply (*simp-all add*: *lazy-rest-def*)
apply(*fold apply-comp-wiring*)
by (*simp add*: *comp-wiring-def parallel2-wiring-def split-def sum.map-comp lassocr*$_w$*-def rassocl*$_w$*-def id-def*[*symmetric*] *sum.map-id*)

qed

have *fact2*: *UNIV* $\vdash_R$ *1*$_C$ $|_=$ *rassocl*$_C$ $\rhd$ *RES* (*fused-resource.fuse real-core′ real-rest′*) (*real-s-core′*, *real-s-rest′*) ≈
1$_C$ $|_=$ *rassocl*$_C$ $\rhd$ *RES* (*fused-resource.fuse* (*basic-core DH0-sample*) *basic-rest*) (*basic-core-sinit*, *basic-rest-sinit*)
(**is** - $\vdash_R$ - $\rhd$ *RES ?L ?s-l* ≈ - $\rhd$ *RES ?R ?s-r*) **proof** −
have [*simp*]: *trace-rest-eq basic-rest basic-rest UNIV UNIV s s* **for** *s*
by (*rule rel-rest′-into-trace-rest-eq*[**where** *S*=(=) **and** *M*=(=)]) (*simp-all add*: *eq-onp-def rel-rest′-eq*)
have [*simp*]: $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full* ⊢*c callee-of-rest basic-rest s* √ **for** *s*
unfolding *callee-of-core-def* **by** (*rule WT-calleeI*) (*cases s*, *case-tac call*, *rename-tac* [!] *x*, *case-tac* [!] *x*, *auto*)
have [*simp*]: $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) ⊢*c callee-of-core* (*basic-core DH0-sample*) *s* √ **for** *s*
unfolding *callee-of-core-def* **by** (*rule WT-calleeI*) (*cases s*, *case-tac call*, *rename-tac* [!] *x*, *case-tac* [!] *x*, *auto*)
have [*simp*]: $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) ⊢*c callee-of-core real-core′ s* √ **for** *s*
unfolding *callee-of-core-def* **by** (*rule WT-calleeI*) (*cases s*, *case-tac call*, *rename-tac* [!] *x*, *case-tac* [!] *x*, *auto*)

have *loc*[*simplified*]: ((*UNIV* <+> *UNIV*) <+> *UNIV* <+> *UNIV*) $\vdash_C$ *?L*(*?s-l*) ≈ *?R*(*?s-r*)
by (*rule fuse-trace-eq*[**where** $\mathcal{IE}$=$\mathcal{I}$*-full* **and** $\mathcal{ICA}$=$\mathcal{I}$*-full* **and** $\mathcal{ICU}$=$\mathcal{I}$*-full* **and** $\mathcal{IRA}$=$\mathcal{I}$*-full* **and** $\mathcal{IRU}$=$\mathcal{I}$*-full*, *simplified outs-plus-*$\mathcal{I}$ *outs-*$\mathcal{I}$*-full*])
(*simp-all add*: *real-rest′-def real-s-rest′-def trac-eq-core-rl*[*simplified*])

show *?thesis*
apply (*rule attach-trace-eq′*[**where** $\mathcal{I}$=$\mathcal{I}$*-full* **and** $\mathcal{I}'$=$\mathcal{I}$*-full*, *simplified outs-plus-*$\mathcal{I}$ *outs-*$\mathcal{I}$*-full*])
apply (*subst trace-eq′-resource-of-oracle*, *rule loc*[*simplified*])
by (*simp-all add*: *WT-converter-*$\mathcal{I}$*-full*)
qed

show *?thesis* **using** *fact2*[*simplified eq-resource-on-UNIV-D*[*OF fact1*]] **by** *blast*
qed

lemma *connect-real*: *connect D* (*obsf-resource real-resource*) = *connect D* (*obsf-resource* (*RES* (*fused-resource.fuse* (*lazy-core DH0-sample*) *lazy-rest*) (*basic-core-sinit*, *basic-rest-sinit*)))
proof −
have [*simp*]: $\mathcal{I}$*-full* ⊢*res real-resource* √
proof −
have [*simp*]: $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) ⊢*res RES* (*fused-resource.fuse real-core′ real-rest′*) (*real-s-core′*, *real-s-rest′*) √
apply (*rule WT-resource-of-oracle*)
apply (*rule WT-calleeI*)
subgoal for *s q*
apply (*cases s*, *cases q*, *rename-tac* [!] *x*, *case-tac* [!] *x*)

prefer *3*
subgoal for *s-cnv-core - - - - y*
apply (*cases s-cnv-core*, *rename-tac s-cnvs s-auth1 s-kern2 s-shell2*)
apply (*case-tac s-auth1*, *rename-tac s-kern1 s-shell1*)
apply (*case-tac s-cnvs*, *rename-tac su s-cnv1 s-cnv2*)
apply (*cases y*, *rename-tac* [!] *z*, *case-tac* [!] *z*, *rename-tac* [!] *query*)
apply (*auto simp add*: *fused-resource.fuse.simps split-def apfst-def*)
apply(*case-tac* (*s-cnv1*, *Inl query*) *rule*: *alice-callee.cases*; *auto split*!:
sum.splits auth.ousr-bob.splits simp add: *Let-def o-def*)
apply(*case-tac* (*s-cnv2*, *Inl query*) *rule*: *bob-callee.cases*; *auto split*!:
sum.splits auth.ousr-bob.splits simp add: *Let-def o-def*)
apply(*case-tac* (*s-cnv1*, *Inr query*) *rule*: *alice-callee.cases*; *auto split*!:
sum.splits
simp add: *Let-def o-def map-gpv-bind-gpv left-gpv-bind-gpv map-gpv′-bind-gpv*
exec-gpv-bind)
apply(*case-tac* (*s-cnv2*, *Inr query*) *rule*: *bob-callee.cases*; *auto split*!:
sum.splits
simp add: *Let-def o-def map-gpv-bind-gpv right-gpv-bind-gpv map-gpv′-bind-gpv*
exec-gpv-bind)
done
by (*auto simp add*: *fused-resource.fuse.simps*)
done

show *?thesis*
unfolding *attach-real*
apply (*rule WT-resource-attach*[**where** $\mathcal{I}'$=$\mathcal{I}$*-full* $\oplus_\mathcal{I}$ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$
$\mathcal{I}$*-full*)])
apply (*rule WT-converter-mono*[*of* $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*))
$\mathcal{I}$*-full* $\oplus_\mathcal{I}$ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*)])
apply (*rule WT-converter-parallel-converter2*)
apply (*rule WT-intro*)+
by (*simp-all add*: $\mathcal{I}$*-full-le-plus-*$\mathcal{I}$)
qed

show *?thesis*
using *trace-eq-obsf-resourceI*[*OF trace-eq-fuse-rl*, *folded attach-real*]
by (*rule connect-cong-trace*[**where** *A*=*UNIV* **and** $\mathcal{I}$=$\mathcal{I}$*-full*])
(*auto intro*!: *WT-obsf-resource*[**where** $\mathcal{I}$=$\mathcal{I}$*-full*, *simplified exception-*$\mathcal{I}$*-full*])
qed

end

end

end

12.10 Concrete security

context *diffie-hellman* **begin**

context
fixes
auth1-rest :: (*'auth1-s-rest*, *auth.event*, *'auth1-iadv-rest*, *'auth1-iusr-rest*, *'auth1-oadv-rest*, *'auth1-ousr-rest*) *rest-wstate* **and**
auth2-rest :: (*'auth2-s-rest*, *auth.event*, *'auth2-iadv-rest*, *'auth2-iusr-rest*, *'auth2-oadv-rest*, *'auth2-ousr-rest*) *rest-wstate* **and**
$\mathcal{I}$*-adv-rest1* **and** $\mathcal{I}$*-adv-rest2* **and** $\mathcal{I}$*-usr-rest1* **and** $\mathcal{I}$*-usr-rest2* **and** *I-auth1-rest* **and** *I-auth2-rest*
assumes
WT-auth1-rest [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-rest1* $\mathcal{I}$*-usr-rest1* *I-auth1-rest* *auth1-rest* **and**
WT-auth2-rest [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-rest2* $\mathcal{I}$*-usr-rest2* *I-auth2-rest* *auth2-rest*
begin

theorem *secure*:
defines $\mathcal{I}$*-real* ≡ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*auth.Inp-Fedit* ‘ (*carrier* $\mathcal{G}$)) *UNIV*)) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*auth.Inp-Fedit* ‘ (*carrier* $\mathcal{G}$)) *UNIV*))) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest2*)
and $\mathcal{I}$*-common* ≡ ($\mathcal{I}$*-uniform UNIV* (*key.Out-Alice* ‘ *carrier* $\mathcal{G}$) $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform UNIV* (*key.Out-Bob* ‘ *carrier* $\mathcal{G}$)) $\oplus_\mathcal{I}$ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-usr-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-rest2*))
and $\mathcal{I}$*-ideal* ≡ $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest2*))
shows *constructive-security-obsf*
(*real-resource TYPE*(-) *TYPE*(-) *auth1-rest auth2-rest*)
(*key.resource* (*ideal-rest auth1-rest auth2-rest*))
(*let sim = CNV sim-callee None in* ((*sim* $|_=$ 1_C) ⊙ $lassocr_C$))
$\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* $\mathcal{A}$
(*ddh.advantage* $\mathcal{G}$ (*DH-adversary TYPE*(-) *TYPE*(-) *auth1-rest auth2-rest* $\mathcal{A}$))
proof
let *?sim* = (*let sim = CNV sim-callee None in* ((*sim* $|_=$ 1_C) ⊙ $lassocr_C$))

have ∗[*WT-intro*]: ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*auth.Inp-Fedit* ‘ *carrier* $\mathcal{G}$) *UNIV*)) $\oplus_\mathcal{I}$
($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*auth.Inp-Fedit* ‘ *carrier* $\mathcal{G}$) *UNIV*)), $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full* $\vdash_C$ *CNV sim-callee s* √ **for** *s*
apply (*rule WT-converter-of-callee*, *simp-all*)
apply (*rename-tac s q r s'*, *case-tac* (*s*, *q*) *rule*: *sim-callee.cases*)
by (*auto split*: *if-splits option.splits*)

show $\mathcal{I}$*-real* $\oplus_\mathcal{I}$ $\mathcal{I}$*-common* ⊢*res real-resource TYPE*(-) *TYPE*(-) *auth1-rest auth2-rest* √
proof −
have [*WT-intro*]: $\mathcal{I}$*-uniform UNIV* (*key.Out-Alice* ‘ *carrier* $\mathcal{G}$) $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*, $\mathcal{I}$*-uniform* (*auth.Inp-Send* ‘ *carrier* $\mathcal{G}$) *UNIV* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform UNIV* (*auth.Out-Recv* ‘ *carrier* $\mathcal{G}$) $\vdash_C$ *CNV alice-callee CState-Void* √
apply (*rule WT-converter-of-callee-invar*[**where** *I*=*pred-cstate* (λ*x*. *x* ∈ *carrier* $\mathcal{G}$)])
subgoal for *s q* **by** (*cases* (*s*, *q*) *rule*: *alice-callee.cases*) (*auto simp add*:

Let-def split: *auth.ousr-bob.splits*)
subgoal for *s q* **by** (*cases* (*s*, *q*) *rule*: *alice-callee.cases*) (*auto split*: *if-split-asm auth.ousr-bob.splits simp add*: *Let-def*)
subgoal by *simp*
done

have [*WT-intro*]: $\mathcal{I}$*-uniform UNIV* (*key.Out-Bob* ‘ *carrier* $\mathcal{G}$) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*, $\mathcal{I}$*-uniform UNIV* (*auth.Out-Recv* ‘ *carrier* $\mathcal{G}$) $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-uniform* (*auth.Inp-Send* ‘ *carrier* $\mathcal{G}$) *UNIV* $\vdash_C$ *CNV bob-callee CState-Void* $\surd$
apply (*rule WT-converter-of-callee-invar*[**where** *I*=*pred-cstate* ($\lambda x.\ x \in$ *carrier* $\mathcal{G}$)])
subgoal for *s q* **by** (*cases* (*s*, *q*) *rule*: *bob-callee.cases*) (*auto simp add*: *Let-def split*: *auth.ousr-bob.splits*)
subgoal for *s q* **by** (*cases* (*s*, *q*) *rule*: *bob-callee.cases*) (*auto simp add*: *Let-def split*: *auth.ousr-bob.splits*)
subgoal by *simp*
done

show *?thesis*
unfolding $\mathcal{I}$*-real-def* $\mathcal{I}$*-common-def real-resource-def Let-def fused-wiring-def*
by (*rule WT-intro*)+
qed

show $\mathcal{I}$*-ideal* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-common* $\vdash$*res key.resource* (*ideal-rest auth1-rest auth2-rest*) $\surd$
unfolding $\mathcal{I}$*-ideal-def* $\mathcal{I}$*-common-def key.resource-def*
apply(*rule callee-invariant-on.WT-resource-of-oracle*[**where** *I*=λ((*kernel*, -), -, *s12*). *key.set-s-kernel kernel* $\subseteq$ *carrier* $\mathcal{G}$ $\wedge$ *pred-prod I-auth1-rest I-auth2-rest s12*]; (*simp add*: *WT-restD*[*OF WT-auth1-rest*] *WT-restD*[*OF WT-auth2-rest*])*?*)
apply *unfold-locales*
subgoal for *s q*
apply (*cases* (*ideal-rest auth1-rest auth2-rest*, *s*, *q*) *rule*: *key.fuse.cases*; *clarsimp split*: *if-split-asm*)
apply (*auto simp add*: *translate-eoracle-def parallel-eoracle-def plus-eoracle-def*)
apply(*auto dest*: *WT-restD-rfunc-adv*[*OF WT-auth1-rest*] *WT-restD-rfunc-adv*[*OF WT-auth2-rest*]
WT-restD-rfunc-usr[*OF WT-auth1-rest*] *WT-restD-rfunc-usr*[*OF WT-auth2-rest*] *key.foldl-poke-invar*)
apply(*auto dest*!: *key.foldl-poke-invar split*: *plus-oracle-split-asm*)
done
subgoal for *s*
apply(*rule WT-calleeI*)
subgoal for *x y s'*
apply(*auto simp add*: *translate-eoracle-def parallel-eoracle-def plus-eoracle-def*)
apply(*auto dest*: *WT-restD-rfunc-adv*[*OF WT-auth1-rest*] *WT-restD-rfunc-adv*[*OF WT-auth2-rest*]
WT-restD-rfunc-usr[*OF WT-auth1-rest*] *WT-restD-rfunc-usr*[*OF WT-auth2-rest*] *split*: *if-split-asm*)
apply(*case-tac xa*)

```
        apply auto
       done
      done
    done

  show ℐ-real, ℐ-ideal ⊢C ?sim √
    unfolding ℐ-real-def ℐ-ideal-def Let-def
    by(rule WT-intro)+

  show pfinite-converter ℐ-real ℐ-ideal ?sim
  proof -
   have [pfinite-intro]:pfinite-converter ((ℐ-full ⊕ℐ (ℐ-full ⊕ℐ ℐ-uniform (auth.Inp-Fedit
‘ carrier 𝒢) UNIV)) ⊕ℐ
     (ℐ-full ⊕ℐ (ℐ-full ⊕ℐ ℐ-uniform (auth.Inp-Fedit ‘ carrier 𝒢) UNIV))) (ℐ-full
⊕ℐ ℐ-full) (CNV sim-callee s) for s
       apply(rule raw-converter-invariant.pfinite-converter-of-callee[where I=λ-.
True], simp-all)
      subgoal
        apply (unfold-locales, simp-all)
        subgoal for s1 s2
          apply (case-tac (s1, s2) rule: sim-callee.cases)
          by (auto simp add: id-def split!: sum.splits if-splits option.splits)
        done
      subgoal for s2 s1  by (case-tac (s1, s2) rule: sim-callee.cases) auto
      done

    show ?thesis
      unfolding ℐ-real-def ℐ-ideal-def Let-def
      by (rule pfinite-intro | rule WT-intro)+
  qed

  show 0 ≤ ddh.advantage 𝒢 (diffie-hellman.DH-adversary 𝒢 auth1-rest auth2-rest
𝒜)
    by(simp add: ddh.advantage-def)

  assume WT [WT-intro]: exception-ℐ (ℐ-real ⊕ℐ ℐ-common) ⊢g 𝒜 √
 show advantage 𝒜 (obsf-resource (?sim |= 1C ▷ key.resource (ideal-rest auth1-rest
auth2-rest))) (obsf-resource (real-resource TYPE(-) TYPE(-) auth1-rest auth2-rest))
≤ ddh.advantage 𝒢 (diffie-hellman.DH-adversary 𝒢 auth1-rest auth2-rest 𝒜)
  proof -
   have id-split[unfolded Let-def]: connect 𝒜 (obsf-resource (?sim |= 1C ▷ key.resource
(ideal-rest auth1-rest auth2-rest))) =
        connect 𝒜 (obsf-resource (?sim |= (1C |= 1C) ▷ key.resource (ideal-rest
auth1-rest auth2-rest)))  (is connect - ?L = connect - ?R)
    proof -
      note [unfolded ℐ-ideal-def, WT-intro] = ‹ℐ-real, ℐ-ideal ⊢C ?sim √›
    note [unfolded ℐ-ideal-def, WT-intro] = ‹ℐ-ideal ⊕ℐ ℐ-common ⊢res key.resource
(ideal-rest auth1-rest auth2-rest) √›
```

```
    have [WT-intro]: WT-rest (ℐ-full ⊕ℐ (ℐ-adv-rest1 ⊕ℐ ℐ-adv-rest2)) (ℐ-full ⊕ℐ
(ℐ-usr-rest1 ⊕ℐ ℐ-usr-rest2)) (λ(-, s12). pred-prod I-auth1-rest I-auth2-rest s12)
      (ideal-rest auth1-rest auth2-rest)
      apply (rule WT-rest.intros; simp)
      subgoal for s q
        apply (cases s, case-tac q, rename-tac [2] x, case-tac [2] x)
          apply (auto simp add: translate-eoracle-def parallel-eoracle-def)
         using WT-restD-rfunc-adv[OF WT-auth1-rest] WT-restD-rfunc-adv[OF
WT-auth2-rest] by fastforce+
      subgoal for s q
        apply (cases s, case-tac q, rename-tac [2] x, case-tac [2] x)
      apply (auto simp add: translate-eoracle-def parallel-eoracle-def plus-eoracle-def)
         using WT-restD-rfunc-usr[OF WT-auth1-rest] WT-restD-rfunc-usr[OF
WT-auth2-rest] by fastforce+
     subgoal by(simp add: WT-restD[OF WT-auth1-rest] WT-restD[OF WT-auth2-rest])
      done

     have *: outs-ℐ (exception-ℐ (ℐ-real ⊕ℐ ℐ-common)) ⊢R ?L ∼ ?R
      apply (rule obsf-resource-eq-ℐ-cong)
      apply (rule eq-ℐ-attach-on')
        apply (rule WT-intro | simp)+
       apply(rule parallel-converter2-eq-ℐ-cong)
        apply(rule eq-ℐ-converter-reflI)
        apply (rule ‹ℐ-real, ℐ-ideal ⊢C ?sim √›[unfolded assms Let-def])
       apply (rule eq-ℐ-converter-sym)
        apply (rule parallel-converter2-id-id)
      by (auto simp add: ℐ-real-def ℐ-common-def)

     show ?thesis
      by (rule * connect-eq-resource-cong WT-intro)+
   qed

   show ?thesis
     unfolding advantage-def Let-def id-split
   unfolding Let-def connect-real connect-ideal[unfolded ideal-resource-def Let-def]
reduction[unfolded advantage-def] ..
 qed
qed

end

end
```

12.11 Asymptotic security

```
locale diffie-hellman' =
  fixes 𝒢 :: security ⇒ 'grp cyclic-group
  assumes diffie-hellman [locale-witness]: ⋀η. diffie-hellman (𝒢 η)
begin
```

sublocale *diffie-hellman* $\mathcal{G}$ η **for** η **..**

definition *real-resource'* **where** *real-resource' rest1 rest2* η = *real-resource TYPE*(-) *TYPE*(-) η (*rest1* η) (*rest2* η)
definition *ideal-resource'* **where** *ideal-resource' rest1 rest2* η = *key.resource* η (*ideal-rest* (*rest1* η) (*rest2* η))
definition *sim'* **where** *sim'* η = (*let sim* = *CNV* (*sim-callee* η) *None in* ((*sim* $|_=$ 1_C) $\odot$ $lassocr_C$))

context
fixes
auth1-rest :: *nat* $\Rightarrow$ (*'auth1-s-rest*, *auth.event*, *'auth1-iadv-rest*, *'auth1-iusr-rest*, *'auth1-oadv-rest*, *'auth1-ousr-rest*) *rest-wstate* **and**
auth2-rest :: *nat* $\Rightarrow$ (*'auth2-s-rest*, *auth.event*, *'auth2-iadv-rest*, *'auth2-iusr-rest*, *'auth2-oadv-rest*, *'auth2-ousr-rest*) *rest-wstate* **and**
$\mathcal{I}$*-adv-rest1* **and** $\mathcal{I}$*-adv-rest2* **and** $\mathcal{I}$*-usr-rest1* **and** $\mathcal{I}$*-usr-rest2* **and** *I-auth1-rest* **and** *I-auth2-rest*
assumes
WT-auth1-rest: $\bigwedge\eta$. *WT-rest* ($\mathcal{I}$*-adv-rest1* η) ($\mathcal{I}$*-usr-rest1* η) (*I-auth1-rest* η) (*auth1-rest* η) **and**
WT-auth2-rest: $\bigwedge\eta$. *WT-rest* ($\mathcal{I}$*-adv-rest2* η) ($\mathcal{I}$*-usr-rest2* η) (*I-auth2-rest* η) (*auth2-rest* η)
begin

theorem *secure*:
defines $\mathcal{I}$*-real* $\equiv$ $\lambda\eta$. (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*auth.Inp-Fedit* ' (*carrier* ($\mathcal{G}$ η))) *UNIV*)) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*auth.Inp-Fedit* ' (*carrier* ($\mathcal{G}$ η))) *UNIV*))) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-rest1* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest2* η)
and $\mathcal{I}$*-common* $\equiv$ $\lambda\eta$. ($\mathcal{I}$*-uniform UNIV* (*key.Out-Alice* ' *carrier* ($\mathcal{G}$ η)) $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform UNIV* (*key.Out-Bob* ' *carrier* ($\mathcal{G}$ η))) $\oplus_\mathcal{I}$ (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-usr-rest1* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-rest2* η))
and $\mathcal{I}$*-ideal* $\equiv$ $\lambda\eta$. $\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-rest1* η $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest2* η))
assumes *DDH*: *negligible* ($\lambda\eta$. *ddh.advantage* ($\mathcal{G}$ η) (*DH-adversary TYPE*(-) *TYPE*(-) η (*auth1-rest* η) (*auth2-rest* η) ($\mathcal{A}$ η)))
shows *constructive-security-obsf'* (*real-resource' auth1-rest auth2-rest*) (*ideal-resource' auth1-rest auth2-rest*) *sim'* $\mathcal{I}$*-real* $\mathcal{I}$*-ideal* $\mathcal{I}$*-common* $\mathcal{A}$
proof(*rule constructive-security-obsf'I*)
show *constructive-security-obsf* (*real-resource' auth1-rest auth2-rest* η) (*ideal-resource' auth1-rest auth2-rest* η) (*sim'* η) ($\mathcal{I}$*-real* η) ($\mathcal{I}$*-ideal* η) ($\mathcal{I}$*-common* η)
($\mathcal{A}$ η) (*ddh.advantage* ($\mathcal{G}$ η) (*DH-adversary TYPE*(-) *TYPE*(-) η (*auth1-rest* η) (*auth2-rest* η) ($\mathcal{A}$ η))) **for** η
unfolding *real-resource'-def ideal-resource'-def sim'-def* $\mathcal{I}$*-real-def* $\mathcal{I}$*-common-def* $\mathcal{I}$*-ideal-def*
by(*rule secure*)(*rule WT-auth1-rest WT-auth2-rest*)+
qed(*rule DDH*)

end

end

end
theory *DH-OTP* **imports**
One-Time-Pad
Diffie-Hellman-CC
begin

We need both a group structure and a boolean algebra. Unfortunately, records allow only one extension slot, so we can't have just a single structure with both operations.

context *diffie-hellman* **begin**

lemma *WT-ideal-rest* [*WT-intro*]:
assumes *WT-auth1-rest* [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-rest1* $\mathcal{I}$*-usr-rest1* *I-auth1-rest* *auth1-rest*
and *WT-auth2-rest* [*WT-intro*]: *WT-rest* $\mathcal{I}$*-adv-rest2* $\mathcal{I}$*-usr-rest2* *I-auth2-rest* *auth2-rest*
shows *WT-rest* ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest2*)) (($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-usr-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-rest2*))
(λ(-, *s*). *pred-prod I-auth1-rest I-auth2-rest s*) (*ideal-rest auth1-rest auth2-rest*)
apply(*rule WT-rest.intros*)
subgoal
by(*auto 4 4 split*: *sum.splits simp add*: *translate-eoracle-def parallel-eoracle-def dest*: *assms*[*THEN WT-restD-rfunc-adv*])
subgoal
apply(*auto 4 4 split*: *sum.splits simp add*: *translate-eoracle-def parallel-eoracle-def plus-eoracle-def dest*: *assms*[*THEN WT-restD-rfunc-usr*])
apply(*simp add*: *map-sum-def split*: *sum.splits*)
done
subgoal **by**(*simp add*: *assms*[*THEN WT-restD-rinit*])
done

end

locale *dh-otp* = *dh*: *diffie-hellman* $\mathcal{G}$ + *otp*: *one-time-pad* $\mathcal{L}$
for $\mathcal{G}$ :: *'grp cyclic-group*
and $\mathcal{L}$ :: *'grp boolean-algebra* +
assumes *carrier-*$\mathcal{G}$*-*$\mathcal{L}$: *carrier* $\mathcal{G}$ = *carrier* $\mathcal{L}$
begin

theorem *secure*:
assumes *WT-rest* $\mathcal{I}$*-adv-resta* $\mathcal{I}$*-usr-resta* *I-auth-rest auth-rest*
and *WT-rest* $\mathcal{I}$*-adv-rest1* $\mathcal{I}$*-usr-rest1* *I-auth1-rest auth1-rest*
and *WT-rest* $\mathcal{I}$*-adv-rest2* $\mathcal{I}$*-usr-rest2* *I-auth2-rest auth2-rest*
shows
constructive-security-obsf

(1_C $|_=$ *wiring-c1r22-c1r22* (*CNV otp.enc-callee* ()) (*CNV otp.dec-callee* ()) $|_=$ 1_C $\triangleright$
fused-wiring $\triangleright$ *diffie-hellman.real-resource* $\mathcal{G}$ *auth1-rest auth2-rest* $\|$ *dh.auth.resource auth-rest*)
(*otp.sec.resource* (*otp.ideal-rest* (*dh.ideal-rest auth1-rest auth2-rest*) *auth-rest*))
((1_C $\odot$
(*parallel-wiring* $\odot$ ((*let sim* = *CNV dh.sim-callee None in* (*sim* $|_=$ 1_C) $\odot$ *lassocr*$_C$) $|_=$ 1_C) $\odot$ *parallel-wiring*) $\odot$
1_C) $\odot$
(*otp.sim* $|_=$ 1_C))
(((($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*otp.sec.Inp-Fedit* ' *carrier* $\mathcal{G}$) *UNIV*)) $\oplus_\mathcal{I}$
($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*otp.sec.Inp-Fedit* ' *carrier* $\mathcal{G}$) *UNIV*))) $\oplus_\mathcal{I}$
($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*otp.sec.Inp-Fedit* ' *carrier* $\mathcal{L}$) *UNIV*))) $\oplus_\mathcal{I}$
(($\mathcal{I}$*-adv-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest2*) $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-resta*))
(($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform* (*otp.sec.Inp-Fedit* ' *carrier* $\mathcal{L}$) *UNIV*)) $\oplus_\mathcal{I}$
(($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ ($\mathcal{I}$*-adv-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-rest2*)) $\oplus_\mathcal{I}$ $\mathcal{I}$*-adv-resta*))
(($\mathcal{I}$*-uniform* (*otp.sec.Inp-Send* ' *carrier* $\mathcal{L}$) *UNIV* $\oplus_\mathcal{I}$ $\mathcal{I}$*-uniform UNIV* (*otp.sec.Out-Recv* ' *carrier* $\mathcal{L}$)) $\oplus_\mathcal{I}$
((($\mathcal{I}$*-full* $\oplus_\mathcal{I}$ $\mathcal{I}$*-full*) $\oplus_\mathcal{I}$ ($\mathcal{I}$*-usr-rest1* $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-rest2*)) $\oplus_\mathcal{I}$ $\mathcal{I}$*-usr-resta*))
$\mathcal{A}$ (*0* + (*ddh.advantage* $\mathcal{G}$
(*diffie-hellman.DH-adversary* $\mathcal{G}$ *auth1-rest auth2-rest*
(*absorb*
(*absorb* $\mathcal{A}$
(*obsf-converter* (1_C $|_=$ *wiring-c1r22-c1r22* (*CNV otp.enc-callee* ()) (*CNV otp.dec-callee* ()) $|_=$ 1_C)))
(*obsf-converter*
(*fused-wiring* $\odot$ (1_C $|_\propto$ *converter-of-resource* (1_C $|_=$ 1_C $\triangleright$ *dh.auth.resource auth-rest*)))))) +
0))
using *assms* **apply** −
apply(*rule constructive-security-obsf-composability*)
apply(*rule otp.secure*)
apply(*rule WT-intro*, *assumption*+)
unfolding *otp.real-resource-def attach-c1f22-c1f22-def*[*abs-def*] *attach-compose*
apply(*rule constructive-security-obsf-lifting-*[**where** *w-adv-real*=1_C **and** *w-adv-ideal-inv*=1_C])
apply(*rule parallel-constructive-security-obsf-fuse*)
apply(*fold carrier-*$\mathcal{G}$*-*$\mathcal{L}$)[*1*]
apply(*rule dh.secure*, *assumption*, *assumption*, *rule constructive-security-obsf-trivial*)
defer
defer
defer
apply(*rule WT-intro*)+
apply(*simp add*: *comp-converter-id-left*)
apply(*rule parallel-converter2-id-id pfinite-intro wiring-intro*)+
apply(*rule WT-intro*|*assumption*)+
apply *simp*
apply(*unfold wiring-c1r22-c1r22-def*)
apply(*rule WT-intro*)+

apply(*fold carrier-$\mathcal{G}$-$\mathcal{L}$*)[*1*]
apply(*rule WT-intro*)+

apply(*rule pfinite-intro*)
apply(*rule pfinite-intro*)
apply(*rule pfinite-intro*)
apply(*rule pfinite-intro*)
apply(*rule pfinite-intro*)
apply(*unfold carrier-$\mathcal{G}$-$\mathcal{L}$*)
apply(*rule pfinite-intro*)
apply(*rule WT-intro*)+
apply(*rule pfinite-intro*)
done

end

end

References

[1] D. A. Basin, A. Lochbihler, U. Maurer, and S. R. Sefidgar. Abstract modeling of systems communication in constructive cryptography using CryptHOL. 2021. http://www.andreas-lochbihler.de/pub/basin2021.pdf, Draft paper.

[2] D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, 33(2):494–566, 2020.

[3] A. Lochbihler. Probabilistic functions and cryptographic oracles in higher order logic. In *European Symposium on Programming (ESOP 2016), Proceedings*, volume 9632 of *LNCS*, pages 503–531. Springer, 2016.

[4] A. Lochbihler. CryptHOL. *Archive of Formal Proofs*, 2017. https://isa-afp.org/entries/CryptHOL.html, Formal proof development.

[5] A. Lochbihler and S. R. Sefidgar. Constructive cryptography in HOL. *Archive of Formal Proofs*, 2018. https://isa-afp.org/entries/Constructive_Cryptography.html, Formal proof development.

[6] A. Lochbihler, S. R. Sefidgar, D. Basin, and U. Maurer. Formalizing constructive cryptography using crypthol. In *Computer Security Foundations Symposium (CSF 2019), Proceedings*, pages 152–166. IEEE, 2019.

[7] 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*, volume 6993 of *LNCS*, pages 33–56. Springer, 2011.

[8] U. Maurer and R. Renner. Abstract cryptography. In *Innovations in Computer Science (ICS 2010), Proceedings*, pages 1–21. Tsinghua University Press, 2011.

[9] U. Maurer and R. Renner. From indifferentiability to constructive cryptography (and back). In *Theory of Cryptography Conference (TCC 2016), Proceedings, Part I*, volume 9985 of *LNCS*, pages 3–24. Springer, 2016.