

Constructive Cryptography in HOL: the Communication Modeling Aspect

Andreas Lochbihler and S. Reza Sefidgar

February 6, 2026

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

1	Material for Isabelle library	4
1.1	Probabilities	5
1.1.1	Conditional probabilities	8
2	Material for CryptHOL	14
2.1	<i>try-gpv</i>	16
2.2	term <i>gpv-stop</i>	23
2.3	term <i>exception-\mathcal{I}</i>	25
2.4	inline	27
3	Material for Constructive Crypto	32
3.1	<i>Constructive-Cryptography.Wiring</i>	37
3.2	Probabilistic finite converter	40
3.3	colossless converter	49
3.4	trace equivalence	50

4	Fused Resource	76
4.1	Event Oracles – they generate events	76
4.2	Event Handlers – they absorb (and silently handle) events . .	78
4.3	Fused Resource Construction	78
4.4	More helpful construction functions	87
5	Traces	90
6	State Isomorphism	117
6.1	Parallel State Isomorphism	118
6.2	Trisplit State Isomorphism	118
6.3	Assocl-Swap State Isomorphism	118
7	Concrete security definition	130
7.1	Composition theorems	133
8	Asymptotic security definition	142
8.1	Composition theorems	144
9	Key specification	147
9.1	Data-types for Parties, State, Events, Input, and Output . . .	147
9.1.1	Basic lemmas for automated handling of party sets (i.e. <i>s-shell</i>)	147
9.2	Defining the event handler	148
9.3	Defining the adversary interface	148
9.4	Defining the user interfaces	149
9.5	Defining the Fuse Resource	149
9.5.1	Lemma showing that the resulting resource is well-typed	149
10	Channel specification	150
10.1	Data-types for Parties, State, Events, Input, and Output . . .	150
10.1.1	Basic lemmas for automated handling of party sets (i.e. <i>s-shell</i>)	151
10.2	Defining the event handler	151
10.3	Defining the adversary interfaces	151
10.4	Defining the user interfaces	152
10.5	Defining the Fused Resource	153
10.5.1	Lemma showing that the resulting resource is well-typed	153
11	One-time-pad construction	154
11.1	Defining user callees	154
11.2	Defining adversary converter	155
11.3	Defining event-translator	155
11.3.1	Basic lemmas for automated handling of <i>sec-party-of-key-party</i>	156
11.4	Defining Ideal and Real constructions	157

11.5	Wiring and simplifying the Ideal construction	157
11.5.1	The ideal attachment lemma	158
11.6	Wiring and simplifying the Real construction	159
11.6.1	The real attachment lemma	160
11.7	Proving the trace-equivalence of simplified Ideal and Real constructions	163
11.7.1	Proving the trace-equivalence of cores	163
11.7.2	Proving the trace equivalence of fused cores and rests	168
11.7.3	Simplifying the final resource by moving the interfaces from core to rest	170
11.8	Concrete security	172
11.9	Asymptotic security	174
12	Diffie-Hellman construction	175
12.1	Defining user callees	176
12.2	Defining adversary callee	177
12.3	Defining event-translator	177
12.4	Defining Ideal and Real constructions	179
12.5	Wiring and simplifying the Ideal construction	180
12.5.1	The ideal attachment lemma	181
12.6	Wiring and simplifying the Real construction	185
12.6.1	The real attachment lemma	185
12.7	A lazy construction and its DH reduction	189
12.7.1	Defining a lazy construction with an inlined sampler	189
12.7.2	Defining a lazy construction with an external sampler	192
12.7.3	Reduction to Diffie-Hellman game	193
12.8	Proving the trace-equivalence of simplified Ideal and Lazy constructions	204
12.9	Proving the trace-equivalence of simplified Real and Lazy constructions	217
12.10	Concrete security	228
12.11	Asymptotic security	232

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*
 $(rel\text{-}set\ A\ ==>\ (A\ ==>\ rel\text{-}set\ B)\ ==>\ rel\text{-}set\ (rel\text{-}prod\ A\ B))\ Sigma$
Sigma
unfolding *Sigma-def* **by** *transfer-prover*

lemma *empty-eq-Plus [simp]*: $\{\} = A <+> B \longleftrightarrow A = \{\} \wedge B = \{\}$
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]*: $map\text{-}sum\ f\ g\ ` (A\ <+>\ B) = f\ ` A\ <+>\ g\ ` B$
by(*auto intro: rev-image-eqI*)

lemma *Plus-subset-Plus-iff [simp]*: $A\ <+>\ B \subseteq C\ <+>\ D \longleftrightarrow A \subseteq C \wedge B \subseteq D$
by *auto*

lemma *map-sum-eq-Inl-iff*: $map\text{-}sum\ f\ g\ x = Inl\ y \longleftrightarrow (\exists x'. x = Inl\ x' \wedge y = f\ x')$
by(*cases x*) *auto*

lemma *map-sum-eq-Inr-iff*: $map\text{-}sum\ f\ g\ x = Inr\ y \longleftrightarrow (\exists x'. x = Inr\ x' \wedge y = g\ x')$
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]*:

$\llbracket \text{bij } f; \text{bij } g \rrbracket \implies \text{inv-into UNIV } (\text{map-sum } f \ g) = \text{map-sum } (\text{inv-into UNIV } f)$
(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 $(\lambda x. f \ x \leq y)$

if *cont lub ord Sup* $(\leq) \ f$

by(*rule ccpo.admissibleI*)(*auto simp add: that[THEN contD] intro!: cSUP-least*)

lemma *admissible-le1-mcont [cont-intro]*:

ccpo.admissible lub ord $(\lambda x. f \ x \leq y)$ **if** *mcont lub ord Sup* $(\leq) \ f$

using that by(*simp add: admissible-le1I mcont-def*)

end

lemma *eq-alt-conversep2*: $(=) = ((\text{BNF-Def.Grp UNIV id})^{-1-1})^{-1-1}$

by(*auto simp add: Grp-def fun-eq-iff*)

lemma *nn-integral-indicator-singleton1 [simp]*:

assumes *[measurable]*: $\{y\} \in \text{sets } M$

shows $(\int^+ x. \text{indicator } \{y\} \ x * f \ x \ \partial M) = \text{emeasure } M \ \{y\} * f \ y$

by(*simp add: mult.commute*)

lemma *nn-integral-indicator-singleton1' [simp]*:

assumes $\{y\} \in \text{sets } M$

shows $(\int^+ x. \text{indicator } \{x\} \ y * f \ x \ \partial M) = \text{emeasure } M \ \{y\} * f \ y$

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*: $\text{pmf } p \ x = 1 \iff p = \text{return-pmf } x$ (**is** *?lhs = ?rhs*)

proof(*rule iffI*)

assume *?lhs*

have $\text{pmf } p \ i = 0$ **if** $x \neq i$ **for** i

proof(*rule antisym*)

have $\text{pmf } p \ i + 1 \leq \text{pmf } p \ i + \text{pmf } p \ x$ **using** $\langle ?lhs \rangle$ **by** *simp*

also have $\dots = \text{measure } (\text{measure-pmf } p) \ \{i, x\}$ **using** that

by(*subst measure-pmf.finite-measure-eq-sum-singleton*)(*simp-all add: pmf.rep-eq*)

also have $\dots \leq 1$ **by** (*rule measure-pmf.subprob-measure-le-1*)

finally show $\text{pmf } p \ i \leq 0$ **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*: $\text{measure } (\text{measure-spmf } p) A = \text{measure } (\text{measure-spmf } p) B$

if $A \cap \text{set-spmf } p = B \cap \text{set-spmf } p$

proof –

have $\text{measure } (\text{measure-spmf } p) A = \text{measure } (\text{measure-spmf } p) (A \cap \text{set-spmf } p) + \text{measure } (\text{measure-spmf } p) (A - \text{set-spmf } p)$

by(*subst measure-spmf.finite-measure-Union[symmetric]*)(*auto intro: arg-cong2[where f=measure]*)

also have $\text{measure } (\text{measure-spmf } p) (A - \text{set-spmf } p) = 0$ **by**(*simp add: measure-spmf-zero-iff*)

also have $0 = \text{measure } (\text{measure-spmf } p) (B - \text{set-spmf } p)$ **by**(*simp add: measure-spmf-zero-iff*)

also have $\text{measure } (\text{measure-spmf } p) (A \cap \text{set-spmf } p) + \dots = \text{measure } (\text{measure-spmf } p) B$

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** $\text{weight-spmf}' = \text{weight-spmf}$

lemma *weight-spmf'-parametric* [*transfer-rule*]: $\text{rel-fun } (\text{rel-spmf } A) (=) \text{weight-spmf}' \text{ weight-spmf}'$

unfolding *weight-spmf'-def* **by**(*rule weight-spmf-parametric*)

lemma *bind-spmf-to-nat-on*:

$\text{bind-spmf } (\text{map-spmf } (\text{to-nat-on } (\text{set-spmf } p)) p) (\lambda n. f (\text{from-nat-into } (\text{set-spmf } p) n)) = \text{bind-spmf } p f$

by(*simp add: bind-map-spmf cong: bind-spmf-cong*)

lemma *try-cond-spmf-fst*:

$\text{try-spmf } (\text{cond-spmf-fst } p x) q = (\text{if } x \in \text{fst } \text{set-spmf } p \text{ then } \text{cond-spmf-fst } p x \text{ else } q)$

by (*metis cond-spmf-fst-eq-return-None lossless-cond-spmf-fst try-spmf-lossless try-spmf-return-None*)

lemma *measure-try-spmf*:

$\text{measure } (\text{measure-spmf } (\text{try-spmf } p q)) A = \text{measure } (\text{measure-spmf } p) A + \text{pmf } p \text{ None} * \text{measure } (\text{measure-spmf } q) A$

proof –

have $\text{emeasure } (\text{measure-spmf } (\text{try-spmf } p q)) A = \text{emeasure } (\text{measure-spmf } p) A + \text{pmf } p \text{ None} * \text{emeasure } (\text{measure-spmf } q) A$

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*:

$\llbracket \text{rel-spmf } R \text{ } p \text{ } q; \text{rel-spmf } S \text{ } q \text{ } r \rrbracket \implies \text{rel-spmf } (R \text{ } OO \text{ } eq\text{-onp } (\lambda x. x \in \text{set-spmf } q) \text{ } OO \text{ } S) \text{ } p \text{ } r$

by(*auto intro: rel-spmf-OO-trans rel-spmf-reflI simp add: eq-onp-def*)

lemma *mcont2mcont-spmf [cont-intro]*:

mcont lub ord Sup (\leq) ($\lambda p. \text{spmfm } (f \text{ } p) \text{ } x$)

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* $\langle \text{auto simp add: rel-pmf-return-pmf1 elim!: option.rel-cases} \rangle$)

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* \implies *ord-spmf* (=) (*p* \upharpoonright *A*) (*q* \upharpoonright *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* (($\lambda p. \text{restrict-spmf } p \text{ } A$) ‘ *Y*)

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

proof(*cases* *Y* = { $\}$)

case *Y*: *False*

have *chain'*: *Complete-Partial-Order.chain* (*ord-spmf* (=)) (($\lambda p. p \upharpoonright A$) ‘ *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 (=))* ($\lambda p. p \upharpoonright A$)
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 (=))* ($\lambda p. \text{restrict-spmf } p \ A$)
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* \Rightarrow *a* | *Some* *y* \Rightarrow *b* *y*)
(*case* *x* *of* *None* \Rightarrow *a'* | *Some* *y* \Rightarrow *b'* *y*)
if *ord-spmf R* *a* *a'* \wedge *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* \Rightarrow '*b* *pmf*
and *A* :: '*b* *set*
and *F* :: '*a* \Rightarrow *real*

defines *F* $\equiv \lambda x. \text{pmf } p \ x * \text{measure (measure-pmf (f x)) } A / \text{measure (measure-pmf (bind-pmf p f)) } A$

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* $\cap (\bigcup x \in \text{set-pmf } p. \text{set-pmf (f x)}) \neq \{\}$ **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: ($\int^+ x. F \ x \ \partial \text{count-space UNIV}$) = 1*
proof –

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

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

lemma *set-cond-bind-pmf*: $\text{set-pmf } \text{cond-bind-pmf} = \{x \in \text{set-pmf } p. \text{set-pmf } (f \ x)$
 $\cap A \neq \{\}\}$
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*: $\text{cond-pmf } (\text{bind-pmf } p \ f) \ A = \text{bind-pmf } \text{cond-bind-pmf } (\lambda x.$
 $\text{cond-pmf } (f \ x) \ A)$
(is *?lhs = ?rhs*)
proof(*rule pmf-eqI*)
fix *i*
have $\text{ennreal } (\text{pmf } ?lhs \ i) = \text{ennreal } (\text{pmf } ?rhs \ i)$
proof(*cases i \in A*)
case *True*
have $\text{ennreal } (\text{pmf } ?lhs \ i) = (\int^+ x. \text{ennreal } (\text{pmf } p \ x) * \text{ennreal } (\text{pmf } (f \ x) \ i))$
 $/ \text{ennreal } (\text{measure-pmf.prob } (p \ \gg\! = \ f) \ A) \ \partial \text{count-space UNIV})$
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 $\dots = (\int^+ x. \text{ennreal } (F \ x) * \text{ennreal } (\text{pmf } (\text{cond-pmf } (f \ x) \ A) \ i))$
 $\partial \text{count-space UNIV})$
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*:
 $cond\text{-}spmf\text{-}fst\ (map\text{-}spmf\ (\lambda((x, s'), y). (f\ x, s', y))\ (pair\text{-}spmf\ p\ q))\ x =$
 $pair\text{-}spmf\ (cond\text{-}spmf\text{-}fst\ (map\text{-}spmf\ (\lambda(x, s'). (f\ x, s'))\ p)\ x)\ q\ \mathbf{is}\ ?lhs = ?rhs$
if *lossless-spmf* *q*
proof –
have $?lhs = map\text{-}spmf\ (\lambda((- , s'), y). (s', y))\ (cond\text{-}spmf\ (pair\text{-}spmf\ p\ q)\ ((\lambda((x, s'), y). (f\ x, s', y)) - ' (\{x\} \times UNIV)))$
by(*simp* *add: cond-spmf-fst-def spmf.map-comp o-def split-def*)
also have $((\lambda((x, s'), y). (f\ x, s', y)) - ' (\{x\} \times UNIV)) = ((\lambda(x, y). (f\ x, y)) - ' (\{x\} \times UNIV)) \times UNIV$
by(*auto*)
also have $map\text{-}spmf\ (\lambda((- , s'), y). (s', y))\ (cond\text{-}spmf\ (pair\text{-}spmf\ p\ q)\ \dots) =$
 $?rhs$
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*: $try\text{-}spmf\ (cond\text{-}spmf\ p\ A)\ q = (if\ set\text{-}spmf\ p \cap A \neq \{\}\ then\ cond\text{-}spmf\ p\ A\ else\ q)$
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*:
 $cond\text{-}spmf\ (try\text{-}spmf\ p\ q)\ A = (if\ lossless\text{-}spmf\ p\ then\ return\text{-}pmf\ None\ else\ cond\text{-}spmf\ q\ A)\ \mathbf{if}\ set\text{-}spmf\ p \cap A = \{\}$
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* :: $'a\ spmf \Rightarrow ('a \Rightarrow 'b\ spmf) \Rightarrow 'b\ set \Rightarrow 'a\ spmf$ **where**

$cond\text{-}bind\text{-}spmf\ p\ f\ A =$
 $(if\ \exists x \in set\text{-}spmf\ p.\ set\text{-}spmf\ (f\ x) \cap A \neq \{\}\ then$
 $cond\text{-}bind\text{-}pmf\ p\ (\lambda x.\ case\ x\ of\ None \Rightarrow return\text{-}pmf\ None\ | Some\ x \Rightarrow f\ x)$
 $(Some\ 'A)$
 $else\ return\text{-}pmf\ None)$

context *begin*

private lemma *defined*: $\llbracket y \in \text{set-spmf } (f x); y \in A; x \in \text{set-spmf } p \rrbracket$
 $\implies \text{Some } \langle A \cap (\bigcup_{x \in \text{set-pmf } p. \text{set-pmf } (\text{case } x \text{ of None} \Rightarrow \text{return-pmf None} \mid \text{Some } x \Rightarrow f x)) \neq \{\} \rangle$
by(*fastforce simp add: in-set-spmf bind-spmf-def*)

lemma *spmf-cond-bind-spmf [simp]*:
 $\text{spmf } (\text{cond-bind-spmf } p f A) x = \text{spmf } p x * \text{measure } (\text{measure-spmf } (f x)) A / \text{measure } (\text{measure-spmf } (\text{bind-spmf } p f)) A$
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]*:
 $\text{set-spmf } (\text{cond-bind-spmf } p f A) = \{x \in \text{set-spmf } p. \text{set-spmf } (f x) \cap A \neq \{\}\}$
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*: $\text{cond-spmf } (\text{bind-spmf } p f) A = \text{bind-spmf } (\text{cond-bind-spmf } p f A) (\lambda x. \text{cond-spmf } (f x) A)$
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*
 $(\text{rel-spmf } (\text{rel-prod } (=) B) \implies (=) \implies \text{rel-spmf } B) \text{ cond-spmf-fst cond-spmf-fst}$
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\} \times UNIV) \cap snd -' (\{x\} \times UNIV)]*)
(fastforce intro: rel-spmf-refl)
done

lemma *cond-spmf-fst-map-prod*:
 $\text{cond-spmf-fst } (\text{map-spmf } (\lambda(x, y). (f x, g x y)) p) (f x) = \text{map-spmf } (g x) (\text{cond-spmf-fst } p x)$
if *inj-on f (insert x (fst ' set-spmf p))*
proof –
have $\text{cond-spmf } p ((\lambda(x, y). (f x, g x y)) -' (\{f x\} \times UNIV)) = \text{cond-spmf } p (((\lambda(x, y). (f x, g x y)) -' (\{f x\} \times UNIV)) \cap \text{set-spmf } p)$
by(*rule cond-spmf-cong*) *simp*
also have $((\lambda(x, y). (f x, g x y)) -' (\{f x\} \times UNIV)) \cap \text{set-spmf } p = (\{x\} \times UNIV) \cap \text{set-spmf } p$
using that **by**(*auto 4 3 dest: inj-onD intro: rev-image-eqI*)
also have $\text{cond-spmf } p \dots = \text{cond-spmf } p (\{x\} \times UNIV)$
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 ($\lambda(x, y). (f x, g x y)$) 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 \Rightarrow ('a \Rightarrow 'b spmf) \Rightarrow 'b \Rightarrow 'a spmf* **where**

cond-bind-spmf-fst p f x = cond-bind-spmf p (map-spmf ($\lambda b. (b, ())) \circ f$) ({x} \times UNIV)

lemma *cond-bind-spmf-fst-map-spmf-fst*:

cond-bind-spmf-fst p (map-spmf fst \circ f) x = cond-bind-spmf p f ({x} \times UNIV)
(**is** *?lhs = ?rhs*)

proof –

have [*simp*]: ($\lambda x. (fst x, ())$) – ‘*{x} \times UNIV = {x} \times UNIV* **by** *auto*

have *?lhs = cond-bind-spmf p ($\lambda x. map-spmf (\lambda x. (fst x, ())) (f x)$) ({x} \times UNIV)*

by(*simp add: cond-bind-spmf-fst-def spmf.map-comp o-def*)

also have $\dots = ?rhs$ **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 \circ f) x) ($\lambda y. cond-spmf-fst (f y) x$)

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 \in set-spmf p. x \in 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*:
 $cond\text{-}spmf\text{-}fst (try\text{-}spmf\ p\ q)\ x = cond\text{-}spmf\text{-}fst\ p\ x$ **if** $x \notin fst\ 'set\text{-}spmf\ q$
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*:
 $cond\text{-}spmf\text{-}fst (try\text{-}spmf\ p\ q)\ x = (if\ lossless\text{-}spmf\ p\ then\ return\text{-}pmf\ None\ else\ cond\text{-}spmf\text{-}fst\ q\ x)$ **if** $x \notin fst\ 'set\text{-}spmf\ p$
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*:
 $cond\text{-}spmf\text{-}fst (map\text{-}spmf (apfst\ f)\ p)\ (f\ x) = cond\text{-}spmf\text{-}fst\ p\ x$ **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*:
 $cond\text{-}spmf\text{-}fst (map\text{-}spmf (\lambda(x, y). (f\ x, g\ x\ y)) (pair\text{-}spmf\ p\ q))\ a =$
 $bind\text{-}spmf (cond\text{-}spmf\text{-}fst (map\text{-}spmf (\lambda x. (f\ x, x))\ p)\ a)\ (\lambda x. map\text{-}spmf (g\ x)$
 $(mk\text{-}lossless\ q))$ (**is** *?lhs = ?rhs*)
proof –
have $(\lambda(x, y). (f\ x, g\ x\ y)) - ' (\{a\} \times UNIV) = f - ' \{a\} \times UNIV$ **by** (*auto*)
moreover **have** $(\lambda x. (f\ x, x)) - ' (\{a\} \times UNIV) = f - ' \{a\}$ **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'*:
 $cond\text{-}spmf\text{-}fst (return\text{-}spmf (x, y))\ z = (if\ x = z\ then\ return\text{-}spmf\ y\ else\ return\text{-}pmf\ None)$
by(*simp add: cond-spmf-fst-def*)

2 Material for CryptHOL

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

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

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

lemma *in-set-sample-uniform* [simp]: $x \in \text{set-spmf } (\text{sample-uniform } n) \iff x < n$

by(simp add: sample-uniform-def)

lemma (in cyclic-group) *inj-on-generator-iff* [simp]: $\llbracket x < \text{order } G; y < \text{order } G \rrbracket \implies \mathbf{g} \ [\frown] \ x = \mathbf{g} \ [\frown] \ y \iff x = y$

using inj-on-generator **by**(auto simp add: inj-on-def)

lemma *map-I-bot* [simp]: $\text{map-I } f \ g \ \perp = \perp$

unfolding bot-I-def *map-I-I-uniform* **by** simp

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

by(rule I-eqI) auto

lemma *interaction-bound-map-gpv'-le*:

defines $ib \equiv \text{interaction-bound}$

shows $\text{interaction-bound } \text{consider } (\text{map-gpv}' \ f \ g \ h \ \text{gpv}) \leq ib \ (\text{consider } \circ \ g) \ \text{gpv}$

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-bezI step.IH[unfolded ib-def])

qed

lemma *interaction-bounded-by-map-gpv'* [interaction-bound]:

assumes *interaction-bounded-by* (consider \circ 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*:

$\text{map-gpv}' \ f \ g \ h \ (\text{bind-gpv } \text{gpv } F) = \text{bind-gpv} \ (\text{map-gpv}' \ \text{id} \ g \ h \ \text{gpv}) \ (\lambda x. \text{map-gpv}' \ f \ g \ h \ (F \ x))$

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'*:

$\text{exec-gpv } \text{callee} \ (\text{map-gpv}' \ f \ g \ h \ \text{gpv}) \ s =$

$\text{map-spmf} \ (\text{map-prod } f \ \text{id}) \ (\text{exec-gpv} \ (\text{map-fun } \text{id} \ (\text{map-fun } g \ (\text{map-spmf} \ (\text{map-prod } h \ \text{id}))) \ \text{callee}) \ \text{gpv} \ s)$

using *exec-gpv-parametric'*

where $S=(=)$ **and** $\text{CALL}=\text{BNF-Def.Grp UNIV } g$ **and** $R=\text{conversep } (\text{BNF-Def.Grp UNIV } h)$ **and** $A=\text{BNF-Def.Grp UNIV } f$,

```

  unfolded rel-gpv''-Grp, simplified]
apply(subst (asm) (2) conversesep-eq[symmetric])
apply(subst (asm) prod.rel-conversesep)
apply(subst (asm) (2 4) eq-alt)
apply(subst (asm) prod.rel-Grp)
apply simp
apply(subst (asm) spmf-rel-conversesep)
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) conversesep-conversesep[symmetric])
apply(subst (asm) rel-fun-conversesep)
apply(subst (asm) rel-fun-Grp)
apply(subst (asm) rel-fun-conversesep)
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  $\implies$  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 $\neg \text{colossless-gpv } \mathcal{I} \text{ gpv} \implies \mathcal{I} \vdash_g \text{gpv}' \checkmark$
shows $\mathcal{I} \vdash_g \text{try-gpv } \text{gpv} \text{ gpv}' \checkmark$
using *assms* **by**(*coinduction arbitrary: gpv*)(*auto 4 4 dest: WT-gpvD colossless-gpvD split: if-split-asm*)

lemma (in *callee-invariant-on*) *exec-gpv-try-gpv*:

fixes *exec-gpv1*
defines *exec-gpv1* \equiv *exec-gpv*
assumes *WT*: $\mathcal{I} \vdash_g \text{gpv} \checkmark$
and *pfinite*: *pfinite-gpv* $\mathcal{I} \text{ gpv}$
and *I*: $I \ s$
and *f*: $\bigwedge s. I \ s \implies f \ (x, \ s) = z$
and *lossless*: $\bigwedge s \ x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; \ I \ s \rrbracket \implies \text{lossless-spmf} \ (\text{callee } s \ x)$
shows $\text{map-spmf } f \ (\text{exec-gpv } \text{callee} \ (\text{try-gpv } \text{gpv} \ (\text{Done } x)) \ s) =$
 $\text{try-spmf} \ (\text{map-spmf } f \ (\text{exec-gpv1 } \text{callee } \text{gpv} \ s)) \ (\text{return-spmf } z)$
(is *?lhs* = *?rhs***)**
proof –
note $[[\text{show-variants}]]$
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.prem*s **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** $R = \text{eq-onp} \ (\lambda x. \ x \in \text{set-pmf} \ (\text{the-gpv } \text{gpv}))$])
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** $R = \text{eq-onp} \ (\lambda x. \ x \in \text{set-pmf} \ (\text{callee } s \ \text{out}))$])
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  $\mathcal{I}$ -full ?gpv  $\implies$ 
TRY ?gpv  $\ggg$  ?f ELSE ?gpv' = ?gpv  $\ggg$  ( $\lambda x.$  TRY ?f x ELSE ?gpv')
  assumes lossless: gen-lossless-gpv b  $\mathcal{I}$  gpv
  and WT1:  $\mathcal{I} \vdash_g$  gpv  $\checkmark$ 
  and WT2:  $\mathcal{I} \vdash_g$  gpv'  $\checkmark$ 
  and WTf:  $\bigwedge x. x \in \text{results-gpv } \mathcal{I} \text{ gpv} \implies \mathcal{I} \vdash_g f x \checkmark$ 
  shows eq- $\mathcal{I}$ -gpv (=)  $\mathcal{I}$  (TRY bind-gpv gpv f ELSE gpv') (bind-gpv gpv ( $\lambda x.$  TRY
f x ELSE gpv'))
  using lossless WT1 WTf
proof(coinduction arbitrary: gpv)
  case (eq- $\mathcal{I}$ -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- $\mathcal{I}$ -gpv(1)] WT-gpvD[OF eq- $\mathcal{I}$ -gpv(2)]
WT-gpvD[OF WT2] WT-gpvD[OF eq- $\mathcal{I}$ -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- $\mathcal{I}$ -gpv(3)[rule-format] eq- $\mathcal{I}$ -gpv-reflI eq- $\mathcal{I}$ -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:

```

$try\text{-}gpv\ (bind\text{-}gpv\ gpv\ f)\ gpv' =$
 $bind\text{-}gpv\ (try\text{-}gpv\ (map\text{-}gpv\ Some\ id\ gpv)\ (Done\ None))\ (\lambda x.\ case\ x\ of\ None\ \Rightarrow$
 $gpv' \mid Some\ x' \Rightarrow try\text{-}gpv\ (f\ x')\ gpv')$
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:*

$bind\text{-}gpv\ (try\text{-}gpv\ (map\text{-}gpv\ Some\ id\ gpv)\ (Done\ None))\ (\lambda x.\ case\ x\ of\ None\ \Rightarrow$
 $Fail \mid Some\ y \Rightarrow f\ y) =$
 $bind\text{-}gpv\ gpv\ f$
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
spmfm-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 $\mathcal{I} \vdash_g\ gpv\ \surd$ **and** *WT2: $\mathcal{I} \vdash_g\ gpv' \surd$*
shows $eq\text{-}\mathcal{I}\text{-}gpv\ (=)\ (\mathcal{I} \oplus_{\mathcal{I}}\ \mathcal{I}')\ (try\text{-}gpv\ (left\text{-}gpv\ gpv)\ (left\text{-}gpv\ gpv'))\ (left\text{-}gpv$
 $(try\text{-}gpv\ gpv\ gpv'))$
using *assms(1)*
apply(*coinduction arbitrary: gpv*)
apply(*auto simp add: map-try-spmf spmf.map-comp o-def generat.map-comp*
spmfm-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-I-gpv-reflI WT-gpv-left-gpv*)
done

lemma *try-gpv-right-gpv:*

assumes $\mathcal{I}' \vdash_g\ gpv\ \surd$ **and** *WT2: $\mathcal{I}' \vdash_g\ gpv' \surd$*
shows $eq\text{-}\mathcal{I}\text{-}gpv\ (=)\ (\mathcal{I} \oplus_{\mathcal{I}}\ \mathcal{I}')\ (try\text{-}gpv\ (right\text{-}gpv\ gpv)\ (right\text{-}gpv\ gpv'))\ (right\text{-}gpv$
 $(try\text{-}gpv\ gpv\ gpv'))$
using *assms(1)*
apply(*coinduction arbitrary: gpv*)
apply(*auto simp add: map-try-spmf spmf.map-comp o-def generat.map-comp*
spmfm-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-I-gpv-reflI WT-gpv-right-gpv*)
done

lemma *bind-try-Done-Fail:* $bind\text{-}gpv\ (TRY\ gpv\ ELSE\ Done\ x)\ f = bind\text{-}gpv\ gpv\ f$
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
spmfm.map-comp o-def that rel-fun-def intro!: rel-pmf-bind-reflI rel-spmf-reflI generat.rel-ref-stong 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  $\equiv$  interaction-bound consider
  shows interaction-bound consider (try-gpv gpv gpv')  $\leq$  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 \mathcal{I} .*lifting* **begin**

lift-definition *stop- \mathcal{I}* :: (*'a, 'b*) $\mathcal{I} \Rightarrow$ (*'a, 'b option*) \mathcal{I} **is**
 $\lambda \text{resp } x. \text{ if } (\text{resp } x = \{\}) \text{ then } \{\} \text{ else insert None (Some ' resp } x) .$

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

lemma *responses-stop- \mathcal{I}* [*simp*]:
 responses- \mathcal{I} (stop- \mathcal{I} $\mathcal{I}) x = (\text{if } x \in \text{outs-}\mathcal{I} \mathcal{I} \text{ then insert None (Some ' responses-}\mathcal{I}$
 $\mathcal{I} x)$ else $\{\}$)
 by *transfer auto*

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

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

lifting-update \mathcal{I} .*lifting*

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

end

lemma *stop- \mathcal{I} -bot* [*simp*]: *stop- \mathcal{I} $\perp = \perp$*
 by (*simp only: bot- \mathcal{I} -def stop- \mathcal{I} -uniform*)(*simp*)

lemma *WT-gpv-stop* [*simp, WT-intro*]: *stop- \mathcal{I} $\mathcal{I} \vdash g \text{ gpv-stop gpv } \checkmark$ if $\mathcal{I} \vdash g \text{ gpv } \checkmark$*
 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: $\mathcal{I} \vdash g \text{ gpv } \checkmark$*
 and *fail: fail $\leq c$*
 shows *expectation-gpv fail (stop- \mathcal{I} $\mathcal{I}) (\lambda-. c) (\text{gpv-stop gpv}) = \text{expectation-gpv}$*
 fail $\mathcal{I} (\lambda-. c) \text{ gpv}$ (is ?lhs = ?rhs)
 proof(*rule antisym*)
 show *expectation-gpv fail (stop- \mathcal{I} $\mathcal{I}) (\lambda-. c) (\text{gpv-stop gpv}) \leq \text{expectation-gpv fail}$*

```

 $\mathcal{I}$  ( $\lambda$ -.  $c$ )  $g_{pv}$ 
  using  $WT$ 
  proof(induction arbitrary:  $g_{pv}$  rule: parallel-fixp-induct-1-1[ $OF$  complete-lattice-partial-function-definitions
complete-lattice-partial-function-definitions expectation- $g_{pv}$ .mono expectation- $g_{pv}$ .mono
expectation- $g_{pv}$ -def expectation- $g_{pv}$ -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:  $WT$ - $g_{pv}D$  intro!:
le-infI2 INF-mono)
  done
qed

```

```

define stop :: ('a option, 'b, 'c option)  $g_{pv} \Rightarrow$  - where stop = expectation- $g_{pv}$ 
fail (stop- $\mathcal{I}$   $\mathcal{I}$ ) ( $\lambda$ -.  $c$ )
show ?rhs  $\leq$  stop ( $g_{pv}$ -stop  $g_{pv}$ ) using  $WT$ 
proof(induction arbitrary:  $g_{pv}$  rule: expectation- $g_{pv}$ -fixp-induct)
  case adm show ?case by simp
  case bottom show ?case by simp
  case (step expectation- $g_{pv}'$ )
  have expectation- $g_{pv}' g_{pv}' \leq c$  if  $\mathcal{I} \vdash g g_{pv}' \checkmark$  for  $g_{pv}'$ 
    using expectation- $g_{pv}$ -const-le[of  $\mathcal{I} g_{pv}'$  fail  $c$ ] fail step.hyps(1)[of  $g_{pv}'$ ] that
    by(simp add: max-def split: if-split-asm)
  then show ?case using step unfolding stop-def
    apply(subst expectation- $g_{pv}$ .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- $\mathcal{I}$ -iff-responses- $\mathcal{I}$  dest:  $WT$ - $g_{pv}$ -ContD
intro: INF-lower2)
    subgoal by(auto intro!: INF-mono rev-bexI dest:  $WT$ - $g_{pv}D$ )
  done
qed
qed

```

```

lemma pgen-lossless- $g_{pv}$ -stop:
  fixes fail and  $g_{pv} ::$  ('a, 'b, 'c)  $g_{pv}$ 
  assumes  $WT$ :  $\mathcal{I} \vdash g g_{pv} \checkmark$ 
  and fail: fail  $\leq 1$ 
  shows pgen-lossless- $g_{pv}$  fail (stop- $\mathcal{I}$   $\mathcal{I}$ ) ( $g_{pv}$ -stop  $g_{pv}$ ) = pgen-lossless- $g_{pv}$  fail  $\mathcal{I}$ 
 $g_{pv}$ 
  by(simp add: pgen-lossless- $g_{pv}$ -def expectation- $g_{pv}$ -stop assms)

```

```

lemma pfinite- $g_{pv}$ -stop [simp]:
  pfinite- $g_{pv}$  (stop- $\mathcal{I}$   $\mathcal{I}$ ) ( $g_{pv}$ -stop  $g_{pv}$ )  $\longleftrightarrow$  pfinite- $g_{pv}$   $\mathcal{I} g_{pv}$  if  $\mathcal{I} \vdash g g_{pv} \checkmark$ 
  using that by(simp add: pgen-lossless- $g_{pv}$ -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*)
 $\implies x \in$ *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*) \implies
 $xy \in$ *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- \mathcal{I}* :: ('a, 'b) $\mathcal{I} \Rightarrow$ ('a, 'b) *exception* \mathcal{I} **where**
exception- \mathcal{I} \mathcal{I} = *map- \mathcal{I} id exception-of-option (stop- \mathcal{I} \mathcal{I})*

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

lemma *responses-exception- \mathcal{I}* [*simp*]:
responses- \mathcal{I} (exception- \mathcal{I} \mathcal{I}) x = (if $x \in$ *outs- \mathcal{I} \mathcal{I}* then *insert Fault (OK ' re-*
sponses- \mathcal{I} \mathcal{I} x) else $\{\}$)
by(*simp add: exception- \mathcal{I} -def image-image*)

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

lemma *exception- \mathcal{I} -full* [*simp*]: *exception- \mathcal{I} \mathcal{I} -full* = *\mathcal{I} -full*
unfolding *exception- \mathcal{I} -def* **by** *simp*

lemma *exception- \mathcal{I} -uniform* [*simp*]:
exception- \mathcal{I} (\mathcal{I} -uniform A B) = (if $B = \{\}$ then \perp else *\mathcal{I} -uniform A (insert Fault*
(OK ' B)))
by(*simp add: exception- \mathcal{I} -def image-image*)

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

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

2.4 inline

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

context

fixes *gpv* :: ('a, 'call, 'ret) *gpv*

assumes *gpv*: *lossless-gpv* \mathcal{I} *gpv* $\mathcal{I} \vdash g$ *gpv* \checkmark

begin

lemma *lossless-spmf-inline1*:

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

and *I*: $I \ s$

shows *lossless-spmf* (*inline1* *callee* *gpv* *s*)

proof –

have $1 = \text{expectation-gpv} \ 0 \ \mathcal{I} \ (\lambda-. \ 1) \ \text{gpv}$ **using** *gpv* **by** (*simp* *add*: *pgen-lossless-gpv-def*)

also have $\dots \leq \text{weight-spmf} \ (\text{inline1} \ \text{callee} \ \text{gpv} \ s)$ **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*: $IO \ \text{out} \ c \in \text{set-spmf} \ (\text{the-gpv} \ \text{gpv})$

with *step.prem*s **have** *out*: $\text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I}$ **by** (*auto* *dest*: *WT-gpvD*)

from *out*[*unfolded in-outs- \mathcal{I} -iff-responses- \mathcal{I}*] **obtain** *input* **where** *input*: $\text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out}$ **by** *auto*

from *out* **have** $(\prod r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out}. \ \text{expectation-gpv}' \ (c \ r)) = \int^+ x. (\prod r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out}. \ \text{expectation-gpv}' \ (c \ r)) \ \partial \text{measure-spmf} \ (\text{the-gpv} \ (\text{callee} \ s \ \text{out}))$

using *lossless* $\langle I \ s \rangle$ **by** (*simp* *add*: *lossless-spmf-def* *measure-spmf.emmeasure-eq-measure*)

also have $\dots \leq \int^+ \text{generat.} \ (\text{case} \ \text{generat} \ \text{of} \ \text{Pure} \ (r, \ s') \Rightarrow \text{weight-spmf} \ (\text{inline1} \ \text{callee} \ (c \ r) \ s') \mid - \Rightarrow 1) \ \partial \text{measure-spmf} \ (\text{the-gpv} \ (\text{callee} \ s \ \text{out}))$

apply(*intro nn-integral-mono-AE*)

apply(*clarsimp split!*: *generat.split*)

subgoal *Pure*

apply(*rule INF-lower2*)

apply(*fastforce* *dest*: *results-callee*[*OF out* $\langle I \ s \rangle$, *THEN subsetD*, *OF results-gpv.Pure*])

apply(*rule step.IH*)

apply(*fastforce* *intro*: *WT-gpvD*[*OF step.prem*s(1) *IO*] *dest*: *results-callee*[*OF out* $\langle I \ s \rangle$, *THEN subsetD*, *OF results-gpv.Pure*])

apply(*fastforce* *dest*: *results-callee*[*OF out* $\langle I \ s \rangle$, *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.prem(1) IO])
    apply(simp-all add: input)
  done
done
finally have ( $\prod r \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out. expectation-gpv}'(c \ r) \leq \dots$  . }
then show ?case using step.prem
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= $\lambda x. \text{measure}(\text{measure-spmf } x)$  -])
  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'  $\equiv$  inline1
  assumes WT:  $\mathcal{I} \vdash g \ \text{gpv} \ \checkmark$ 
    and pfinite: pfinite-gpv  $\mathcal{I} \ \text{gpv}$ 
    and f:  $\bigwedge s. I \ s \implies f(x, s) = z$ 
    and lossless:  $\bigwedge s \ x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I \ s \rrbracket \implies \text{colossless-gpv } \mathcal{I}' \ (\text{callee } s \ x)$ 
    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 ( $\lambda(\text{out}, c, \text{rpv}). (\text{out}, c, \lambda \text{input. try-gpv}(\text{rpv} \ \text{input}) \ (\text{Done } x))))$  (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.premis 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 ( $\exists$ ) bind-spmf-def)
  apply(simp add: bind-assoc-pmf)
  apply(rule rel-pmf-bindI[where  $R=eq-onp (\lambda x. x \in set-pmf (the-gpv\ gpv))$ ]))
  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  $R=eq-onp (\lambda x. x \in set-pmf (the-gpv (callee\ s\ out)))$ ]))
    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:  $\bigwedge s. I s \implies f(x, s) = z$ 
and lossless:  $\bigwedge s x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I s \rrbracket \implies \text{colossless-gpv } \mathcal{I}' \text{ (callee } s \ x)$ 
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  $\times$  's) option) TYPE(('ret  $\times$  's) option) (is rel-spmf
(eq- $\mathcal{I}$ -generat - - ?X) ?lhs ?rhs)
proof -
have ?lhs = map-spmf
  ( $\lambda x. \text{case } x \text{ of } \text{Inl } rs \Rightarrow \text{Pure } rs \mid \text{Inr } (out, oracle, rpv) \Rightarrow \text{IO } out \ (\lambda input. \text{map-gpv } f \text{ id } (\text{bind-gpv } (\text{try-gpv } (\text{map-gpv } \text{Some } id \ (oracle \ input)) \ (Done \ None))) \ (\lambda xy. \text{case } xy \text{ of } \text{None} \Rightarrow \text{Fail} \mid \text{Some } (x, y) \Rightarrow \text{inline callee } (rpv \ x) \ y)))$ )
  (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 ( $\lambda(out, c, rpv). (out, c, \lambda input. \text{TRY } rpv \ input \ \text{ELSE } \text{Done } x)))$ ) (inline1 callee gpv s) ELSE return-spmf (Inl z)
by(intro inline1-try-gpv)(auto intro: f lossless)
also have ... = map-spmf ( $\lambda y. \text{case } y \text{ of } \text{None} \Rightarrow \text{Inl } z \mid \text{Some } x' \Rightarrow \text{map-sum } f \ (\lambda(out, c, rpv). (out, c, \lambda input. \text{try-gpv } (rpv \ input) \ (Done \ x))) \ x')$ )
  (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 ( $\lambda y. \text{case } y \text{ of } \text{None} \Rightarrow \text{Pure } z \mid \text{Some } (\text{Inl } x) \Rightarrow \text{Pure } (f \ x) \mid \text{Some } (\text{Inr } (out, oracle, rpv)) \Rightarrow \text{IO } out \ (\lambda input. \text{try-gpv } (\text{map-gpv } f \ \text{id } (\text{bind-gpv } (oracle \ input) \ (\lambda(x, y). \text{inline callee } (rpv \ x) \ y))) \ (Done \ z)))$ )
  (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 generat.map-comp
spmf-rel-map intro!: rel-spmf-refl)

```

```

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  $x = \lambda x y. x = y \wedge y \in \text{results-gpv } \mathcal{I}'$  (TRY map-gpv
Some id (rpv input) ELSE Done None)])
apply(rule exI conjI refl)+
apply(rule eq-I-gpv-refl)
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 \Longrightarrow C \Longrightarrow rel-*spm*f (rel-prod R S)) \Longrightarrow *cr-prod2* s S \Longrightarrow C
 \Longrightarrow rel-*spm*f (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 ($\lambda(x, s''). (x, (s, s''))$) (*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-I-uniform-UNIV* [*simp*]: *I-uniform* A UNIV \vdash res res \surd
by(*coinduction arbitrary: res*) *auto*

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

lemma *eq- \mathcal{I} -gpv-eq-OO*:
assumes *eq- \mathcal{I} -gpv* (=) $\mathcal{I} \ \text{gpv} \ \text{gpv}' \ \text{eq-}\mathcal{I}\text{-gpv} \ A \ \mathcal{I} \ \text{gpv}' \ \text{gpv}''$
shows $\text{eq-}\mathcal{I}\text{-gpv} \ A \ \mathcal{I} \ \text{gpv} \ \text{gpv}''$
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* (=) $\mathcal{I} \ \text{gpv}'' \ \text{gpv}' \ \text{eq-}\mathcal{I}\text{-gpv} \ A \ \mathcal{I} \ \text{gpv} \ \text{gpv}'$
shows $\text{eq-}\mathcal{I}\text{-gpv} \ A \ \mathcal{I} \ \text{gpv} \ \text{gpv}''$
using *eq- \mathcal{I} -gpv-relcompp*[**where** $A' = \text{conversep} \ (=)$, *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* $A \ \mathcal{I} \ \text{gpv}1 \ \text{gpv}1'$
and *eq- \mathcal{I} -gpv* $A \ \mathcal{I} \ \text{gpv}2 \ \text{gpv}2'$
shows $\text{eq-}\mathcal{I}\text{-gpv} \ A \ \mathcal{I} \ (\text{try-gpv} \ \text{gpv}1 \ \text{gpv}2) \ (\text{try-gpv} \ \text{gpv}1' \ \text{gpv}2')$
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}*) $\text{gpv}1 \ \text{gpv}2$
shows $\text{eq-}\mathcal{I}\text{-gpv} \ A \ \mathcal{I} \ (\text{map-gpv}' \ f \ g \ h \ \text{gpv}1) \ (\text{map-gpv}' \ f' \ g \ h \ \text{gpv}2)$
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' $\mathcal{I}' \vdash_C \text{conv}1 \sim \text{conv}2$*

```

  and inj f surj g
  shows  $\mathcal{I}, \mathcal{I}' \vdash_C \text{map-converter } f g f' g' \text{ conv1} \sim \text{map-converter } f g f' g' \text{ conv2}$ 
  using assms(1)
proof(coinduction arbitrary: conv1 conv2)
  case eq- $\mathcal{I}$ -converter
  from this(2) have  $f q \in \text{outs-}\mathcal{I} (\text{map-}\mathcal{I} (\text{inv-into UNIV } f) (\text{inv-into UNIV } g) \mathcal{I})$ 
  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- $\mathcal{I}$ -gppv-map-gpv')
  apply(simp add: BNF-Def.vimage2p-def prod.rel-map)
  apply(erule eq- $\mathcal{I}$ -gppv-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-refl)

lemma *connect-map-gpv'*:
 $\text{connect } (\text{map-gpv}' f g h \text{ adv}) \text{ res} = \text{map-spmf } f (\text{connect } \text{adv } (\text{map-resource } g h \text{ res}))$
 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**
 $\text{run-resource fail-resource} = (\lambda-. \text{return-pmf None})$

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

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

primcorec *const-resource* :: ('a, 'b) resource **where**
 $\text{run-resource const-resource} = (\lambda-. \text{map-spmf } (\text{map-prod id } (\lambda-. \text{const-resource})) (\text{return-spmf } (y, ())))$

end

lemma *const-resource-sel* [simp]: $\text{run-resource } (\text{const-resource } y) = (\lambda-. \text{return-spmf } (y, \text{const-resource } y))$
 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*]:
 $\mathcal{I} \vdash \text{res } \text{const-resource } y \checkmark \iff (\forall x \in \text{outs-}\mathcal{I} \mathcal{I}. y \in \text{responses-}\mathcal{I} \mathcal{I} x) \text{ (is } ?\text{lhs} \iff ?\text{rhs})$
proof(*intro iffI ballI*)
show $y \in \text{responses-}\mathcal{I} \mathcal{I} x$ **if** $?\text{lhs}$ **and** $x \in \text{outs-}\mathcal{I} \mathcal{I}$ **for** x **using** *WT-resourceD*[*OF that*] **by** *auto*
show $?\text{lhs}$ **if** $?\text{rhs}$ **using** *that* **by**(*coinduction*)(*auto*)
qed

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

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

end

lemma *const-converter-sel* [*simp*]: *run-converter* (*const-converter* y) = ($\lambda\cdot$. *Done* ($y, \text{const-converter } y$))
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 =
(λx . *map-gpv* (*map-prod id* (λconv . *case conv of None* \Rightarrow *merge-exception-converter* | *Some conv'* \Rightarrow *conv'*)) *id* (
Pause x (λy . *Done* (*case merge-exception y of Fault* \Rightarrow (*Fault, Some* (*const-converter* *Fault*))
| *OK y'* \Rightarrow (*OK y', None*))))))

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

$run_converter\ merge_exception_converter\ x =$
 $Pause\ x\ (\lambda y. Done\ (case\ merge_exception\ y\ of\ Fault\ \Rightarrow\ (Fault,\ const_converter\ Fault)\ |\ OK\ y'\ \Rightarrow\ (OK\ y',\ merge_exception_converter)))$
 $by(simp\ add:\ o_def\ fun_eq_iff\ split:\ exception.split)$

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 $WT_const_converter\ [WT_intro,\ simp]:$
 $\mathcal{I},\ \mathcal{I}'\ \vdash_C\ const_converter\ x\ \checkmark\ \mathbf{if}\ \forall q \in outs\ \mathcal{I}\ \mathcal{I}. x \in responses\ \mathcal{I}\ \mathcal{I}\ q$
 $by(coinduction)(auto\ simp\ add:\ that)$

lemma $WT_merge_exception_converter\ [WT_intro,\ simp]:$
 $exception\ \mathcal{I}\ (\mathcal{I}1' \oplus_{\mathcal{I}}\ \mathcal{I}2'),\ exception\ \mathcal{I}\ \mathcal{I}1' \oplus_{\mathcal{I}}\ exception\ \mathcal{I}\ \mathcal{I}2' \vdash_C\ merge_exception_converter$
 \checkmark
 $by(coinduction)\ auto$

lemma $inline_left_gpv_merge_exception_converter:$
 $bind_gpv\ (inline\ run_converter\ (map_gpv'\ id\ id\ option_of_exception\ (gpv_stop\ (left_gpv\ gpv))))\ merge_exception_converter)\ (\lambda(x,\ conv'). case\ x\ of\ None\ \Rightarrow\ Fail\ |\ Some\ x'\ \Rightarrow\ Done\ (x,\ conv')) =$
 $bind_gpv\ (left_gpv\ (map_gpv'\ id\ id\ option_of_exception\ (gpv_stop\ gpv)))\ (\lambda x. case\ x\ of\ None\ \Rightarrow\ Fail\ |\ Some\ x'\ \Rightarrow\ Done\ (x,\ merge_exception_converter))$
 $\mathbf{apply}(coinduction\ arbitrary:\ gpv\ rule:\ gpv.coinduct_strong)$
 $\mathbf{apply}(simp\ add:\ bind_gpv.sel\ inline_sel\ map_bind_spmf\ bind_map_spmf\ del:\ bind_gpv.sel')$
 $\mathbf{apply}(subst\ inline1_unfold)$
 $\mathbf{apply}(clarsimp\ simp\ add:\ bind_map_spmf\ intro!\ rel_spmf_bind_refl\ simp\ add:\ generat.map_comp\ case_map_generat\ o_def\ split!\ generat.split\ intro!\ rel_funI)$
 $\mathbf{subgoal\ for}\ gpv\ out\ c\ input\ \mathbf{by}(cases\ input;\ auto\ split!\ exception.split)$
 \mathbf{done}

lemma $inline_right_gpv_merge_exception_converter:$
 $bind_gpv\ (inline\ run_converter\ (map_gpv'\ id\ id\ option_of_exception\ (gpv_stop\ (right_gpv\ gpv))))\ merge_exception_converter)\ (\lambda(x,\ conv'). case\ x\ of\ None\ \Rightarrow\ Fail\ |\ Some\ x'\ \Rightarrow\ Done\ (x,\ conv')) =$
 $bind_gpv\ (right_gpv\ (map_gpv'\ id\ id\ option_of_exception\ (gpv_stop\ gpv)))\ (\lambda x. case\ x\ of\ None\ \Rightarrow\ Fail\ |\ Some\ x'\ \Rightarrow\ Done\ (x,\ merge_exception_converter))$
 $\mathbf{apply}(coinduction\ arbitrary:\ gpv\ rule:\ gpv.coinduct_strong)$
 $\mathbf{apply}(simp\ add:\ bind_gpv.sel\ inline_sel\ map_bind_spmf\ bind_map_spmf\ del:\ bind_gpv.sel')$
 $\mathbf{apply}(subst\ inline1_unfold)$
 $\mathbf{apply}(clarsimp\ simp\ add:\ bind_map_spmf\ intro!\ rel_spmf_bind_refl\ simp\ add:$

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* ($\langle 1_w \rangle$)

where

id-wiring \equiv (*id, id*)

definition

swap-lassocr_w :: (*'a + 'b + 'c, 'd + 'e + 'f, 'b + 'a + 'c, 'e + 'd + 'f*) *wiring*

where

swap-lassocr_w \equiv *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 \equiv *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* \odot *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* \odot *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: swapC ∘ swapC = 1C
  unfolding swapC-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:  $\mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3), \mathcal{I}1 \oplus_{\mathcal{I}} (\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3) \vdash_C \text{swap-lassocr}$ 
  ∘ swap-lassocr  $\sim 1_C$ 
  (is ? $\mathcal{I}$ ,-  $\vdash_C$  ?lhs  $\sim$  -)
proof -
  have ?lhs = (rassoC ∘ (swapC |= 1C)) ∘ (lassocrC ∘ rassoC) ∘ ((swapC |=

```

$1_C) \odot \text{lassocr}_C$
by(simp add: swap-lassocr-def comp-converter-assoc)
also have $\dots = \text{rassoel}_C \odot ((\text{swap}_C \odot \text{swap}_C) \models 1_C) \odot \text{lassocr}_C$
unfolding rassoel-inverse comp-converter-id-left
by(simp add: parallel-converter2-comp1-out comp-converter-assoc)
also have $?I, ?I \vdash_C \dots \sim \text{rassoel}_C \odot 1_C \odot \text{lassocr}_C$ **unfolding** swap-inverse
by(rule eq-I-converter-refl eq-I-comp-cong WT-intro parallel-converter2-id-id)+
also have $\text{rassoel}_C \odot 1_C \odot \text{lassocr}_C = 1_C$ **by**(simp add: comp-converter-id-left
lassocr-inverse)
finally show ?thesis .
qed

lemma parallel-wiring-inverse:

$(I1 \oplus_I I2) \oplus_I (I3 \oplus_I I4), (I1 \oplus_I I2) \oplus_I (I3 \oplus_I I4) \vdash_C \text{parallel-wiring} \odot$
 $\text{parallel-wiring} \sim 1_C$
(is ?I, - \vdash_C ?lhs \sim -)

proof -

have ?lhs = $(\text{lassocr}_C \odot (1_C \models \text{swap-lassocr})) \odot (\text{rassoel}_C \odot \text{lassocr}_C) \odot ((1_C \models \text{swap-lassocr}) \odot \text{rassoel}_C)$

by(simp add: parallel-wiring-def comp-converter-assoc)

also have $\dots = (\text{lassocr}_C \odot (1_C \models \text{swap-lassocr})) \odot (1_C \models \text{swap-lassocr}) \odot \text{rassoel}_C$

by(simp add: lassocr-inverse comp-converter-id-left)

also have $\dots = \text{lassocr}_C \odot (1_C \models (\text{swap-lassocr} \odot \text{swap-lassocr})) \odot \text{rassoel}_C$

by(simp add: parallel-converter2-comp2-out comp-converter-assoc)

also have $?I, ?I \vdash_C \dots \sim \text{lassocr}_C \odot (1_C \models 1_C) \odot \text{rassoel}_C$

by(rule eq-I-converter-refl eq-I-comp-cong parallel-converter2-eq-I-cong WT-intro swap-lassocr-inverse)+

also have $?I, ?I \vdash_C \text{lassocr}_C \odot (1_C \models 1_C) \odot \text{rassoel}_C \sim \text{lassocr}_C \odot 1_C \odot \text{rassoel}_C$

by(rule eq-I-converter-refl eq-I-comp-cong parallel-converter2-id-id WT-intro)+

also have $\text{lassocr}_C \odot 1_C \odot \text{rassoel}_C = 1_C$ **by**(simp add: comp-converter-id-left
rassoel-inverse)

finally show ?thesis .

qed

— Analogous to *attach-wiring* in Wiring.thy

definition

attach-wiring-right ::

$(a, b, c, d) \text{ wiring} \Rightarrow$

$(s \Rightarrow e \Rightarrow (f \times s, a, b) \text{ gpv}) \Rightarrow (s \Rightarrow e \Rightarrow (f \times s, c, d) \text{ gpv})$

where

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

lemma

attach-wiring-right-simps:

$\text{attach-wiring-right} (f, g) = \text{map-fun id (map-fun id (map-gpv' id f g))}$

by(simp add: attach-wiring-right-def)

lemma

comp-converter-of-callee-wiring:

assumes *wiring*: $wiring \mathcal{I}2 \mathcal{I}3 \text{ conv } w$

and *WT*: $\mathcal{I}1, \mathcal{I}2 \vdash_C CNV \text{ callee } s \checkmark$

shows $\mathcal{I}1, \mathcal{I}3 \vdash_C CNV \text{ callee } s \odot \text{ conv } \sim CNV (\text{attach-wiring-right } w \text{ callee}) s$

using *wiring*

proof *cases*

case (*wiring* $f g$)

from - *wiring*(2) **have** $\mathcal{I}1, \mathcal{I}3 \vdash_C CNV \text{ callee } s \odot \text{ conv } \sim CNV \text{ callee } s \odot$

map-converter id id f g 1_C

by(*rule eq-I-comp-cong*)(*rule eq-I-converter-reflI*[*OF WT*])

also have $CNV \text{ callee } s \odot \text{ map-converter id id f g } 1_C = \text{map-converter id id f g}$
(*CNV callee s*)

by(*subst comp-converter-map-converter2*)(*simp add: comp-converter-id-right*)

also have $\dots = CNV (\text{attach-wiring-right } w \text{ callee}) s$

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

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

pfinite-converterI: *pfinite-converter* $\mathcal{I} \mathcal{I}' \text{ conv}$ **if**

$\bigwedge a. a \in \text{outs-}\mathcal{I} \mathcal{I} \Longrightarrow \text{pfinite-gpv } \mathcal{I}' (\text{run-converter conv } a)$

$\bigwedge a b \text{ conv}'. \llbracket a \in \text{outs-}\mathcal{I} \mathcal{I}; (b, \text{conv}') \in \text{results-gpv } \mathcal{I}' (\text{run-converter conv } a) \rrbracket \Longrightarrow \text{pfinite-converter } \mathcal{I} \mathcal{I}' \text{ conv}'$

lemma *pfinite-converter-coinduct*[*consumes 1, case-names pfinite-converter, case-conclusion pfinite-converter pfinite step, coinduct pred: pfinite-converter*]:

assumes $X \text{ conv}$

and *step*: $\bigwedge \text{conv } a. \llbracket X \text{ conv}; a \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \Longrightarrow \text{pfinite-gpv } \mathcal{I}' (\text{run-converter conv } a) \wedge$

$(\forall (b, \text{conv}') \in \text{results-gpv } \mathcal{I}' (\text{run-converter conv } a). X \text{ conv}' \vee \text{pfinite-converter } \mathcal{I} \mathcal{I}' \text{ conv}')$

shows *pfinite-converter* $\mathcal{I} \mathcal{I}' \text{ conv}$

```

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

lemma pfinite-converterD:
  [ pfinite-converter  $\mathcal{I}$   $\mathcal{I}'$  conv;  $a \in \text{outs-}\mathcal{I}$   $\mathcal{I}$  ]
   $\implies$  pfinite-gpv  $\mathcal{I}'$  (run-converter conv  $a$ )  $\wedge$ 
    ( $\forall (b, \text{conv}') \in \text{results-gpv } \mathcal{I}'$  (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}  $\mathcal{I}1' \subseteq \text{outs-}\mathcal{I}$   $\mathcal{I}1$   $\mathcal{I}2 \leq \mathcal{I}2'$ 
    and WT:  $\mathcal{I}1, \mathcal{I}2 \vdash_C \text{conv} \checkmark$ 
  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:  $\bigwedge x s. [ x \in \text{outs-}\mathcal{I} \mathcal{I}; I s ] \implies \text{pfinite-gpv } \mathcal{I}'$  (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 ( $\lambda \text{conv. pfinite-converter } \mathcal{I} \mathcal{I}' \text{ conv} \wedge$ 
 $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark$ )
  by(unfold-locales)(auto dest: WT-converterD pfinite-converterD)

interpretation run-pfinite-converter: raw-converter-invariant
 $\mathcal{I}$   $\mathcal{I}'$  run-converter  $\lambda \text{conv. pfinite-converter } \mathcal{I} \mathcal{I}' \text{ conv} \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark$  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 $(\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2)$ $(\mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2')$ $(\text{conv}1 \mid= \text{conv}2)$
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* \equiv *expectation-gpv* 1 \mathcal{I}'
assumes *callee*: $\bigwedge s x. \llbracket x \in \text{outs-}\mathcal{I}; I s \rrbracket \implies$ *pfinite-gpv* \mathcal{I}' (*callee* s x)
and *WT-gpv*: $\mathcal{I} \vdash_g$ *gpv* \checkmark
and I : $I s$
and *f-le-1*: $\bigwedge x. f x \leq 1$
shows *expectation-gpv* 1 \mathcal{I} f *gpv* \leq *expectation-gpv2* $(\lambda(x, s). f x)$ (*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 $(\int^+ x. (\text{case } x \text{ of } \text{Pure } a \Rightarrow f a \mid \text{IO out } c \Rightarrow \prod_{r \in \text{responses-}\mathcal{I}} \mathcal{I} \text{ out. } \text{expectation-gpv}'(c r)) \partial \text{measure-spmf}(\text{the-gpv } \text{gpv}) + 1 * \text{ennreal}(\text{pmf}(\text{the-gpv } \text{gpv}) \text{None}) =$
 $(\sum^+ x. \text{pmf}(\text{the-gpv } \text{gpv}) x * (\text{case } x \text{ of } \text{Some } (\text{Pure } a) \Rightarrow f a \mid \text{Some } (\text{IO out } c) \Rightarrow \prod_{r \in \text{responses-}\mathcal{I}} \mathcal{I} \text{ out. } \text{expectation-gpv}'(c r) \mid \text{None} \Rightarrow 1))$
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 $\dots \leq (\sum^+ x. \text{pmf}(\text{the-gpv } \text{gpv}) x * (\text{case } x \text{ of } \text{None} \Rightarrow 1 \mid \text{Some } (\text{Pure } a) \Rightarrow f a \mid \text{Some } (\text{IO out } c) \Rightarrow$
 $(\sum^+ x. \text{ennreal}(\text{pmf}(\text{the-gpv}(\text{callee } s \text{ out})) \ggg \text{case-generat}(\lambda(x, y). \text{inline1 callee}(c x) y) (\lambda \text{out } \text{rpv}'. \text{return-spmf}(\text{Inr}(\text{out}, \text{rpv}', c)))) x *$
 $(\text{case } x \text{ of } \text{None} \Rightarrow 1 \mid \text{Some } (\text{Inl}(a, s)) \Rightarrow f a$
 $\mid \text{Some } (\text{Inr}(r, \text{rpv}, \text{rpv}')) \Rightarrow \prod_{x \in \text{responses-}\mathcal{I}} \mathcal{I}' r. \text{expectation-gpv } 1 \mathcal{I}'$
 $(\lambda(x, s'). \text{expectation-gpv } 1 \mathcal{I}'(\lambda(x, s). f x) (\text{inline callee}(\text{rpv}' x) s')) (\text{rpv } x))))$
(is nn-integral - ?lhs \leq nn-integral - ?rhs)
proof(*rule nn-integral-mono*)
fix $x :: ('a, 'call, ('a, 'call, 'ret)) \text{rpv}$ *generat option*
consider $(\text{None}) x = \text{None} \mid (\text{Pure } a)$ **where** $x = \text{Some } (\text{Pure } a)$
 $\mid (\text{IO out } c)$ **where** $x = \text{Some } (\text{IO out } c)$ $\text{IO out } c \in \text{set-spmf}(\text{the-gpv } \text{gpv})$
 $\mid (\text{outside}) \text{out } c$ **where** $x = \text{Some } (\text{IO out } c)$ $\text{IO out } c \notin \text{set-spmf}(\text{the-gpv } \text{gpv})$

```

    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.premis have out: out ∈ outs- $\mathcal{I}$   $\mathcal{I}$  by(auto dest: WT-gpvD)
  then obtain response where resp: response ∈ responses- $\mathcal{I}$   $\mathcal{I}$  out unfolding
in-outs- $\mathcal{I}$ -iff-responses- $\mathcal{I}$  by blast
  with out step.premis IO have WT-resp [WT-intro]:  $\mathcal{I} \vdash g$  c response  $\surd$  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  $\lambda$ -. 1  $\mathcal{I}$  c
response] expectation-gpv-const-le[OF WT-resp, of 1 1]
    by(simp add: le-fun-def f-le-1)

  have ( $\prod r \in \text{responses-}\mathcal{I}$   $\mathcal{I}$  out. expectation-gpv' (c r)) =
    ( $\int^+$  generat. ( $\prod r \in \text{responses-}\mathcal{I}$   $\mathcal{I}$  out. expectation-gpv' (c r))  $\partial$ measure-spmf
(the-gpv (callee s out))) +
    ( $\prod r \in \text{responses-}\mathcal{I}$   $\mathcal{I}$  out. expectation-gpv' (c r)) * (1 - ennreal (weight-spmf
(the-gpv (callee s out))))
  by(simp add: measure-spmf.emmeasure-eq-measure add-mult-distrib2[symmetric]
semiring-class.distrib-left[symmetric] add-diff-inverse-ennreal weight-spmf-le-1)
  also have ... ≤ ( $\int^+$  generat. ( $\prod r \in \text{responses-}\mathcal{I}$   $\mathcal{I}$  out. expectation-gpv' (c
r))  $\partial$ 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 ... = ( $\sum^+$  z. ennreal (pmf (the-gpv (callee s out)) z) * (case z of
None  $\Rightarrow$  1 | Some generat  $\Rightarrow$  ( $\prod r \in \text{responses-}\mathcal{I}$   $\mathcal{I}$  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 ... ≤ ( $\sum^+$  z. ennreal (pmf (the-gpv (callee s out)) z) *
(case z of None  $\Rightarrow$  1 | Some (IO out' rpv')  $\Rightarrow$   $\prod x \in \text{responses-}\mathcal{I}$   $\mathcal{I}'$  out'.
expectation-gpv 1  $\mathcal{I}'$  ( $\lambda(x, s')$ . expectation-gpv 1  $\mathcal{I}'$  ( $\lambda(x, s)$ . f x) (inline callee (c x)
s')) (rpv' x)
| Some (Pure (r, s'))  $\Rightarrow$  ( $\sum^+$  x. ennreal (pmf (inline1 callee (c r) s') x)
* (case x of None  $\Rightarrow$  1 | Some (Inl (a, s))  $\Rightarrow$  f a | Some (Inr (out', rpv, rpv'))  $\Rightarrow$ 
 $\prod x \in \text{responses-}\mathcal{I}$   $\mathcal{I}'$  out'. expectation-gpv 1  $\mathcal{I}'$  ( $\lambda(x, s')$ . expectation-gpv
1  $\mathcal{I}'$  ( $\lambda(x, s)$ . f x) (inline callee (rpv' x) s')) (rpv x))))
  (is nn-integral - ?lhs2 ≤ nn-integral - ?rhs2)
proof(intro nn-integral-mono)
  fix z :: ('ret  $\times$  's, 'call', ('ret  $\times$  's, 'call', 'ret') rpv) generat option

```

```

consider (None) z = None | (Pure) x' s' where z = Some (Pure (x', s'))
Pure (x', s') ∈ set-spmf (the-gpv (callee s out))
| (IO') out' c' where z = Some (IO out' c') IO out' c' ∈ set-spmf (the-gpv
(callee s out))
| (outside) generat where z = Some generat generat ∉ set-spmf (the-gpv
(callee s out))
by (metis dest-IO.elims not-Some-eq old.prod.exhaust)
then show ?lhs2 z ≤ ?rhs2 z
proof cases
case None then show ?thesis by simp
next
case Pure
hence (x', s') ∈ results-gpv I' (callee s out) by(simp add: results-gpv.Pure)
with results-callee step.premis out have x: x' ∈ responses-I I' out and s':
I s' by auto
with IO out step.premis have WT-c [WT-intro]: I ⊢ g c x' √ by(auto dest:
WT-gpvD)
from x have (INF r∈responses-I I' out. expectation-gpv' (c r)) ≤
expectation-gpv' (c x') by(rule INF-lower)
also have ... ≤ expectation-gpv2 (λ(x, s). f x) (inline callee (c x') s')
using WT-c s' by(rule step.IH)
also have ... = ∫+ xx. (case xx of Inl (x, -) ⇒ f x
| Inr (out', callee', rpv) ⇒ INF r'∈responses-I I' out'. expectation-gpv
1 I' (λ(r, s'). expectation-gpv 1 I' (λ(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 ... = (∑+ xx. ennreal (pmf (inline1 callee (c x') s') xx) *
(case xx of None ⇒ 1 | Some (Inl (x, -) ⇒ f x
| Some (Inr (out', callee', rpv) ⇒ INF r'∈responses-I I' out'.
expectation-gpv 1 I' (λ(r, s'). expectation-gpv 1 I' (λ(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' ∈ outs-I I' using WT-callee out step.premis by(auto
dest: WT-gpvD)
have (INF r∈responses-I I' out. expectation-gpv' (c r)) ≤ min (INF (r,

```

$s') \in (\bigcup r' \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ \text{out}'. \text{results-gpv } \mathcal{I}' \ (c' \ r')) . \text{expectation-gpv}' \ (c \ r)) \ 1$
using IO' $\text{results-callee}[OF \ \text{out}, \ \text{of } s] \ \text{step.premis}$ **by**($\text{intro } INF\text{-mono}$
 min.boundedI)($\text{auto intro: results-gpv.IO intro!: INF-lower2}[OF \ \text{resp}] \ \text{exp-resp}$)
also have $\dots \leq (INF \ r' \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ \text{out}'. \text{min } (INF \ (r, \ s') \in \text{results-gpv}$
 $\mathcal{I}' \ (c' \ r')). \text{expectation-gpv}' \ (c \ r)) \ 1$
using $\text{resp out}'$ **unfolding** $\text{inf-min}[symmetric] \ \text{in-outs-}\mathcal{I}\text{-iff-responses-}\mathcal{I}$
by($\text{subst } INF\text{-inf-const2}$)($\text{auto simp add: INF-UNION}$)
also have $\dots \leq (INF \ r' \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ \text{out}'. \text{expectation-gpv } 1 \ \mathcal{I}' \ (\lambda(r',$
 $s'). \text{expectation-gpv } 1 \ \mathcal{I}' \ (\lambda(x, \ s). \ f \ x) \ (\text{inline callee } (c \ r') \ s')) \ (c' \ r'))$
(is - \leq (INF $r' \in \dots$? $r \ r'$))
proof($\text{rule } INF\text{-mono}, \ \text{rule } \text{bexI}$)
fix r'
assume $r': r' \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ \text{out}'$
have $\text{fin: pfinite-gpv } \mathcal{I}' \ (c' \ r')$ **using** $\text{callee}[OF \ \text{out}, \ \text{of } s] \ IO' \ r'$
 $WT\text{-callee}[OF \ \text{out}, \ \text{of } s] \ \text{step.premis}$ **by**($\text{auto dest: pfinite-gpv-ContD}$)
have $\text{min } (INF \ (r, \ s') \in \text{results-gpv } \mathcal{I}' \ (c' \ r')). \text{expectation-gpv}' \ (c \ r)) \ 1 \leq$
 $\text{min } (INF \ (r, \ s') \in \text{results-gpv } \mathcal{I}' \ (c' \ r')). \text{expectation-gpv2} \ (\lambda(x, \ s). \ f \ x) \ (\text{inline callee}$
 $(c \ r) \ s')) \ 1$
using $IO \ IO'$ $\text{step.premis out results-callee}[OF \ \text{out}, \ \text{of } s] \ r'$
by(intro min.mono)($\text{auto intro!: INF-mono rev-bexI step.IH dest:}$
 $WT\text{-gpv-ContD intro: results-gpv.IO}$)
also have $\dots \leq ?r \ r'$ **unfolding** $\text{expectation-gpv2-def}$ **using** fin **by**(rule
 $\text{pfinite-INF-le-expectation-gpv}$)
finally show $\text{min } (INF \ (r, \ s') \in \text{results-gpv } \mathcal{I}' \ (c' \ r')). \text{expectation-gpv}' \ (c$
 $r)) \ 1 \leq \dots$
qed
finally show $?thesis$ **using** IO' **by**($\text{simp add: mult-mono}$)
next
case outside then show $?thesis$ **by**($\text{simp add: in-set-spmf-iff-spmf}$)
qed
qed
also have $\dots = (\sum^+ z. \sum^+ x.$
 $\text{ennreal } (\text{pmf } (\text{the-gpv } (\text{callee } s \ \text{out})) \ z) \ *)$
 $\text{ennreal } (\text{pmf } (\text{case } z \ \text{of } \text{None} \Rightarrow \text{return-pmf } \text{None} \mid \text{Some } (\text{Pure } (x, \ xb)))$
 $\Rightarrow \text{inline1 callee } (c \ x) \ xb \mid \text{Some } (IO \ \text{out } \text{rpv}') \Rightarrow \text{return-spmf } (\text{Inr } (\text{out}, \ \text{rpv}', \ c)))$
 $x) \ *$
 $(\text{case } x \ \text{of } \text{None} \Rightarrow 1 \mid \text{Some } (\text{Inl } (a, \ s)) \Rightarrow f \ a \mid \text{Some } (\text{Inr } (\text{out}, \ \text{rpv},$
 $\text{rpv}')) \Rightarrow \prod_{x \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ \text{out}.}$
 $\text{expectation-gpv } 1 \ \mathcal{I}' \ (\lambda(x, \ s'). \ \text{expectation-gpv } 1 \ \mathcal{I}' \ (\lambda(x, \ s). \ f \ x) \ (\text{inline}$
 $\text{callee } (\text{rpv}' \ x) \ s')) \ (\text{rpv } x)))$
(is $\dots = (\sum^+ z. \sum^+ x. ?f \ x \ z)$
by($\text{auto intro!: nn-integral-cong split!: option.split generat.split simp add:}$
 $\text{mult.assoc nn-integral-cmult ennreal-indicator}$)
also have $(\sum^+ z. \sum^+ x. ?f \ x \ z) = (\sum^+ x. \sum^+ z. ?f \ x \ z)$
by($\text{subst nn-integral-fst-count-space}$ **where** $f = \text{case-prod } -, \ \text{simplified}$)(simp
 $\text{add: nn-integral-snd-count-space}[symmetric]$)
also have $\dots = (\sum^+ x.$
 $\text{ennreal } (\text{pmf } (\text{the-gpv } (\text{callee } s \ \text{out})) \ggg \text{case-generat } (\lambda(x, \ y). \ \text{inline1}$
 $\text{callee } (c \ x) \ y) \ (\lambda \text{out } \text{rpv}'. \ \text{return-spmf } (\text{Inr } (\text{out}, \ \text{rpv}', \ c)))) \ x) \ *$

```

(case x of None  $\Rightarrow$  1 | Some (Inl (a, s))  $\Rightarrow$  f a | Some (Inr (r, rpv,
rpv'))  $\Rightarrow$ 
   $\prod_{x \in \text{responses-}\mathcal{I} \mathcal{I}' r. \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s'). \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s). f x) (\text{inline callee } (rpv' x) s')) (rpv x))}$ 
  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 ... =  $(\sum^+ y. \sum^+ x. \text{ennreal } (pmf (the-gpv gpv) y) * \text{ennreal } (\text{case } y \text{ of None } \Rightarrow pmf (return-pmf None) x | \text{Some } (Pure xa) \Rightarrow pmf (return-spmf (Inl (xa, s))) x | \text{Some } (IO out rpv) \Rightarrow pmf (bind-spmf (the-gpv (callee s out)) (\lambda generat' \Rightarrow \text{case generat' of Pure } (x, y) \Rightarrow \text{inline1 callee } (rpv x) y | IO out rpv' \Rightarrow \text{return-spmf } (Inr (out, rpv', rpv)))) x) * (\text{case } x \text{ of None } \Rightarrow 1 | \text{Some } (Inl (a, s)) \Rightarrow f a | \text{Some } (Inr (out, rpv, rpv')) \Rightarrow \prod_{x \in \text{responses-}\mathcal{I} \mathcal{I}' out. \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s'). \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s). f x) (\text{inline callee } (rpv' x) s')) (rpv x))}$ 
(is - =  $(\sum^+ y. \sum^+ x. ?f x y)$ )
  by(auto intro!: nn-integral-cong split!: option.split generat.split simp add:
nn-integral-cmult mult.assoc ennreal-indicator)
  also have  $(\sum^+ y. \sum^+ x. ?f x y) = (\sum^+ x. \sum^+ y. ?f x y)$ 
  by(subst nn-integral-fst-count-space[where f=case-prod -, simplified])(simp add:
nn-integral-snd-count-space[symmetric])
  also have ... =  $(\sum^+ x. (pmf (inline1 callee gpv s) x) * (\text{case } x \text{ of None } \Rightarrow 1 | \text{Some } (Inl (a, s)) \Rightarrow f a | \text{Some } (Inr (out, rpv, rpv')) \Rightarrow \prod_{x \in \text{responses-}\mathcal{I} \mathcal{I}' out. \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s'). \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s). f x) (\text{inline callee } (rpv' x) s')) (rpv x))}$ 
  by(rewrite in - =  $\sqsupset$  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 ... =  $(\int^+ res. (\text{case } res \text{ of Inl } (a, s) \Rightarrow f a | \text{Inr } (out, rpv, rpv') \Rightarrow \prod_{x \in \text{responses-}\mathcal{I} \mathcal{I}' out. \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s'). \text{expectation-gpv } 1 \mathcal{I}' (\lambda(x, s). f x) (\text{inline callee } (rpv' x) s')) (rpv x))} \partial\text{measure-spmf } (inline1 \text{ callee } gpv s) + \text{ennreal } (pmf (inline1 \text{ callee } gpv s) None))$ 
  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  $(\lambda(x, s). f x)$  (inline callee gpv s) unfolding
expectation-gpv2-def
  by(rewrite in - =  $\sqsupset$  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* \checkmark

and *callee*: $\bigwedge s x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I \ s \rrbracket \implies \text{pfinite-gpv } \mathcal{I}' \ (\text{callee } s \ x)$

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* \checkmark **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}' ($\lambda-. 1$) (*inline callee gpv s*) ≤ 1 **by**(*simp add: max-def*)

have $1 = \text{expectation-gpv } 1 \ \mathcal{I} \ (\lambda-. 1) \ \text{gpv}$ **using** *fin* **by**(*simp add: pgen-lossless-gpv-def*)

also have $\dots \leq \text{expectation-gpv } 1 \ \mathcal{I}' \ (\lambda-. 1) \ (\text{inline callee gpv } s)$

by(*rule expectation-gpv-1-le-inline*[*unfolded split-def*]; *rule callee I WT WT-callee order-refl*)

finally show $1 \leq \dots$.

qed

end

lemma *pfinite-comp-converter* [*pfinite-intro*]:

pfinite-converter $\mathcal{I}1 \ \mathcal{I}3$ (*conv1* \odot *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*
 $\checkmark \ \mathcal{I}2, \mathcal{I}3 \vdash_C$ *conv2* \checkmark

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* \checkmark

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** $\forall (b, \text{conv}') \in ?\text{res}. ?P \ b \ \text{conv}' \ \vee \ -$)

proof(*clarify*)

fix *b conv''*

assume $(b, \text{conv}'') \in ?\text{res}$

then obtain *conv1'* *conv2'* **where** [*simp*]: *conv''* = *comp-converter conv1'*
conv2'

and *inline*: $((b, \text{conv1}'), \text{conv2}') \in \text{results-gpv } \mathcal{I}3$ (*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'  $\surd$  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  $\exists$  conv. conv' = map-converter f g f' g' conv  $\wedge$ 
      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-lassocrC [pfinite-intro]: pfinite-converter (( $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$ )  $\oplus_{\mathcal{I}}$   $\mathcal{I}3$ ) ( $\mathcal{I}1$ 
 $\oplus_{\mathcal{I}}$  ( $\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3$ )) lassocrC
by(coinduction)(auto simp add: lassocrC-def)

```

```

lemma pfinite-rassoclC [pfinite-intro]: pfinite-converter ( $\mathcal{I}1 \oplus_{\mathcal{I}}$  ( $\mathcal{I}2 \oplus_{\mathcal{I}} \mathcal{I}3$ )) (( $\mathcal{I}1$ 
 $\oplus_{\mathcal{I}}$   $\mathcal{I}2$ )  $\oplus_{\mathcal{I}}$   $\mathcal{I}3$ ) rassoclC
by(coinduction)(auto simp add: rassoclC-def)

```

lemma *pfiniteswap_C* [*pfiniteswap-intro*]: *pfiniteswap-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 *pfiniteswap-lassocr* [*pfiniteswap-intro*]: *pfiniteswap-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 pfiniteswap-intro WT-intro*)**+**

lemma *pfiniteswap-rassocl* [*pfiniteswap-intro*]: *pfiniteswap-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 pfiniteswap-intro WT-intro*)**+**

lemma *pfiniteswap-parallel-wiring* [*pfiniteswap-intro*]:
pfiniteswap-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 pfiniteswap-intro WT-intro*)**+**

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

lemma *pfiniteswap-converter-of-resource* [*simp, pfiniteswap-intro*]: *pfiniteswap-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**
 $\bigwedge a. a \in \text{outs-}\mathcal{I} \mathcal{I} \Longrightarrow \text{colossless-gpv } \mathcal{I}' (\text{run-converter } \text{conv } a)$
 $\bigwedge a b \text{ conv}'. \llbracket a \in \text{outs-}\mathcal{I} \mathcal{I}; (b, \text{conv}') \in \text{results-gpv } \mathcal{I}' (\text{run-converter } \text{conv } a) \rrbracket$
 $\Longrightarrow \text{colossless-converter } \mathcal{I} \mathcal{I}' \text{ conv}'$

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

lemma *colossless-converterD*:

$\llbracket \text{colossless-converter } \mathcal{I} \mathcal{I}' \text{ conv}; a \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket$
 $\implies \text{colossless-gpv } \mathcal{I}' (\text{run-converter conv } a) \wedge$
 $(\forall (b, \text{conv}') \in \text{results-gpv } \mathcal{I}' (\text{run-converter conv } a). \text{colossless-converter } \mathcal{I} \mathcal{I}'$
 $\text{conv}')$
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*
 $\mathcal{I} \mathcal{I}' \text{run-converter } (\lambda \text{conv}. \text{colossless-converter } \mathcal{I} \mathcal{I}' \text{conv} \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark)$
by(*unfold-locales*)(*auto dest: WT-converterD colossless-converterD*)

interpretation *run-colossless-converter*: *raw-converter-invariant*
 $\mathcal{I} \mathcal{I}' \text{run-converter } \lambda \text{conv}. \text{colossless-converter } \mathcal{I} \mathcal{I}' \text{conv} \wedge \mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark$ **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*: $\bigwedge \text{distinguisher}. \mathcal{I} \vdash_g \text{distinguisher} \checkmark \implies \text{connect } \text{distinguish-}$
 $\text{res} = \text{connect } \text{distinguish-} \text{res}'$
shows *outs- \mathcal{I} \mathcal{I}* $\vdash_R \text{res} \approx \text{res}'$
using *assms* **by**(*rule distinguish-trace-eq*)(*auto intro: WT-fail-resource*)

lemma *attach-trace-eq'*:
assumes *eq*: *outs- \mathcal{I} \mathcal{I}* $\vdash_R \text{res1} \approx \text{res2}$
and *WT1* [*WT-intro*]: $\mathcal{I} \vdash_{\text{res}} \text{res1} \checkmark$
and *WT2* [*WT-intro*]: $\mathcal{I} \vdash_{\text{res}} \text{res2} \checkmark$
and *WT-conv* [*WT-intro*]: $\mathcal{I}', \mathcal{I} \vdash_C \text{conv} \checkmark$
shows *outs- \mathcal{I} \mathcal{I}'* $\vdash_R \text{conv} \triangleright \text{res1} \approx \text{conv} \triangleright \text{res2}$
proof(*rule distinguish-trace-eq*)
fix $\mathcal{D} :: ('c, 'd) \text{distinguisher}$
assume [*WT-intro*]: $\mathcal{I}' \vdash_g \mathcal{D} \checkmark$
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 \triangleright res1) = connect \mathcal{D} (conv \triangleright res2)* **by**(*simp add:*
distinguish-attach)
qed

lemma *trace-callee-eq-trans* [*trans*]:
 $\llbracket \text{trace-callee-eq } \text{callee1 } \text{callee2 } A \ p \ q; \text{trace-callee-eq } \text{callee2 } \text{callee3 } A \ q \ r \rrbracket$
 $\implies \text{trace-callee-eq } \text{callee1 } \text{callee3 } A \ p \ r$
by(*simp add: trace-callee-eq-def*)

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

proposition *trace-callee-eq-coinduct [consumes 1, case-names step sim]*:
fixes $callee1 :: ('a, 'b, 's1) \text{ callee}$ **and** $callee2 :: ('a, 'b, 's2) \text{ callee}$
assumes *start: S p q*
and *step: $\bigwedge p \ q \ a. \llbracket S \ p \ q; \ a \in A \rrbracket \implies$*
 $\text{bind-spmf } p (\lambda s. \text{map-spmf } \text{fst} (\text{callee1 } s \ a)) = \text{bind-spmf } q (\lambda s. \text{map-spmf } \text{fst}$
($\text{callee2 } s \ a$)
and *sim: $\bigwedge p \ q \ a \ \text{res} \ \text{res}' \ b \ s'' \ s'. \llbracket S \ p \ q; \ a \in A; \ \text{res} \in \text{set-spmf } p; \ (b, \ s'') \in$*
 $\text{set-spmf } (\text{callee1 } \text{res} \ a); \ \text{res}' \in \text{set-spmf } q; \ (b, \ s') \in \text{set-spmf } (\text{callee2 } \text{res}' \ a) \rrbracket$
 $\implies S (\text{cond-spmf-fst} (\text{bind-spmf } p (\lambda s. \text{callee1 } s \ a)) \ b)$
 $(\text{cond-spmf-fst} (\text{bind-spmf } q (\lambda s. \text{callee2 } s \ a)) \ b)$
shows $\text{trace-callee-eq } callee1 \ callee2 \ A \ p \ q$
proof(*rule trace-callee-eqI*)
fix $xs :: ('a \times 'b) \text{ list}$ **and** z
assume $xs: \text{set } xs \subseteq A \times UNIV$ **and** $z: z \in A$

from *start* **show** $\text{trace-callee } callee1 \ p \ xs \ z = \text{trace-callee } callee2 \ q \ xs \ z$ **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 $\text{trace-callee } callee1 \ p \ (xy \# xs) \ z =$
 $\text{trace-callee } callee1 (\text{cond-spmf-fst} (\text{bind-spmf } p (\lambda s. \text{callee1 } s \ x)) \ y) \ xs \ z$
by(*simp add: bind-map-spmf split-def o-def*)
also have $\dots = \text{trace-callee } callee2 (\text{cond-spmf-fst} (\text{bind-spmf } q (\lambda s. \text{callee2 } s$
 $x)) \ y) \ xs \ z$
proof(*cases $\exists s \in \text{set-spmf } q. \exists s'. (y, s') \in \text{set-spmf } (\text{callee2 } s \ x)$*)

case *True*
then obtain $s\ s'$ **where** $ss': s \in \text{set-spmf } q\ (y, s') \in \text{set-spmf } (\text{callee2 } s\ x)$
by *blast*
from *Cons* **have** $x \in A$ **by** *simp*
from ss' *step*[*THEN arg-cong*[**where** $f = \text{set-spmf}$], *OF* $\langle S\ p\ q \rangle$ *this*] **obtain**
 $ss\ ss'$
where $ss \in \text{set-spmf } p\ (y, ss') \in \text{set-spmf } (\text{callee1 } ss\ x)$
by(*clarsimp simp add: bind-UNION*) *force*
from sim [*OF* $\langle S\ p\ q \rangle$ - *this* ss'] **show** *?thesis* **using** *Cons.prem*s **by** (*auto*
intro: Cons.IH)
next
case *False*
then have $\text{cond-spmf-fst } (\text{bind-spmf } q\ (\lambda s. \text{callee2 } s\ x))\ y = \text{return-pmf } \text{None}$
by(*auto simp add: bind-eq-return-pmf-None*)
moreover from step [*OF* $\langle S\ p\ q \rangle$, *of* x , *THEN arg-cong*[**where** $f = \text{set-spmf}$],
THEN eq-refl] *Cons.prem*s *False*
have $\text{cond-spmf-fst } (\text{bind-spmf } p\ (\lambda s. \text{callee1 } s\ x))\ y = \text{return-pmf } \text{None}$
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 $\dots = \text{trace-callee } \text{callee2 } q\ (xy \# xs)\ z$
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 $\text{callee1} :: ('a, 'b, 's1)\ \text{callee}$ **and** $\text{callee2} :: ('a, 'b, 's2)\ \text{callee}$
assumes *start: S p q*
and *step: $\bigwedge p\ q\ a. \llbracket S\ p\ q; a \in A \rrbracket \implies$*
 $\text{bind-spmf } p\ (\lambda s. \text{map-spmf } \text{fst } (\text{callee1 } s\ a)) = \text{bind-spmf } q\ (\lambda s. \text{map-spmf } \text{fst}$
 $(\text{callee2 } s\ a))$
and *sim: $\bigwedge p\ q\ a\ res\ res'\ b\ s''\ s'. \llbracket S\ p\ q; a \in A; res \in \text{set-spmf } p; (b, s'') \in$*
 $\text{set-spmf } (\text{callee1 } res\ a); res' \in \text{set-spmf } q; (b, s') \in \text{set-spmf } (\text{callee2 } res'\ a) \rrbracket$
 $\implies S\ (\text{cond-spmf-fst } (\text{bind-spmf } p\ (\lambda s. \text{callee1 } s\ a))\ b)$
 $(\text{cond-spmf-fst } (\text{bind-spmf } q\ (\lambda s. \text{callee2 } s\ a))\ b) \vee$
 $\text{trace-callee-eq } \text{callee1 } \text{callee2 } A\ (\text{cond-spmf-fst } (\text{bind-spmf } p\ (\lambda s. \text{callee1 } s$
 $a))\ b)\ (\text{cond-spmf-fst } (\text{bind-spmf } q\ (\lambda s. \text{callee2 } s\ a))\ b)$
shows *trace-callee-eq callee1 callee2 A p q*

proof –

from *start* **have** $S\ p\ q \vee \text{trace-callee-eq } \text{callee1 } \text{callee2 } A\ p\ q$ **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 \implies 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- \mathcal{I}* (\mathcal{I} -uniform A UNIV :: ('a, 'b) \mathcal{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*: $0 \leq advantage\ A\ res1\ res2$
by(simp add: advantage-def)

lemma *comp-converter-of-resource-conv-parallel-converter*:
(converter-of-resource res $|_{\infty}$ 1_C) \odot conv = converter-of-resource res $|_{\infty}$ 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 $|_{\infty}$ converter-of-resource res) \odot conv = conv $|_{\infty}$ 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 $|_{\infty}$ map-converter f'' g'' f' g' conv2 =
map-converter (map-sum f f'') (map-sum g g'') f' g' (conv1 $|_{\infty}$ 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:*

$conv1 \mid_{\infty} map\text{-}converter\ f\ g\ id\ id\ conv2 = map\text{-}converter\ (map\text{-}sum\ id\ f)\ (map\text{-}sum\ id\ g)\ id\ id\ (conv1 \mid_{\infty} conv2)$

by(*rule parallel-converter-map-converter[where f=id and g=id and f'=id and g'=id, simplified]*)

lemma *parallel-converter-assoc2:*

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

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:*

$converter\text{-}of\text{-}resource\ res1 \mid_{\infty} converter\text{-}of\text{-}resource\ res2 = converter\text{-}of\text{-}resource\ (res1 \parallel res2)$

by(*coinduction arbitrary: res1 res2*)

(*auto 4 3 simp add: rel-fun-def map-lift-spmf spmf-rel-map intro!: rel-spmf-refl split!: sum.split*)

lemma *map-Inr-parallel-converter:*

$map\text{-}converter\ Inr\ f\ g\ h\ (conv1 \mid_{\infty} conv2) = map\text{-}converter\ id\ (f \circ Inr)\ g\ h\ conv2$
(*is ?lhs = ?rhs*)

proof –

have *?lhs = map-converter Inr f id id (map-converter id id g h conv1 \mid_{∞} 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 \mid_{∞} conv2) = map-converter id (f \circ 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 \circ Inr) id id (map-converter id id g h conv2) = ?rhs* **by** *simp*

finally show *?thesis .*

qed

lemma *map-Inl-parallel-converter:*

$map\text{-}converter\ Inl\ f\ g\ h\ (conv1 \mid_{\infty} conv2) = map\text{-}converter\ id\ (f \circ Inl)\ g\ h\ conv1$
(*is ?lhs = ?rhs*)

proof –

have *?lhs = map-converter Inl f id id (map-converter id id g h conv1 \mid_{∞} 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 \mid_{∞} conv2) = map-converter id (f \circ 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 ∘ 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 |_∞ conv2) = left-interface conv1 |_∞ 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 |_∞ conv2) = right-interface conv1 |_∞ 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
 |_∞ conv2) = conv2 |_∞ 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:
  [[  $\mathcal{I}1, \mathcal{I} \vdash_C \text{conv}1 \sim \text{conv}1'$ ;  $\mathcal{I}2, \mathcal{I} \vdash_C \text{conv}2 \sim \text{conv}2'$  ]]
   $\implies \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2, \mathcal{I} \vdash_C \text{parallel-converter } \text{conv}1 \text{ conv}2 \sim \text{parallel-converter } \text{conv}1' \text{ conv}2'$ 
  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 simplifying exec-gpv
lemma
  exec-gpv-parallel-oracle-right:
    exec-gpv (oracle1 †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= $\lambda l r. l = \text{Inr } r$  and R= $\lambda l r. l = \text{Inr } r$ , 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 †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= $\lambda l r. l = \text{Inl } r$  and R= $\lambda l r. l = \text{Inl } r$ , 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* \Rightarrow (*'in, 'out*) *resource-obsf* **where**
run-resource (*obsf-resource res*) = ($\lambda x.$
map-spmf (*map-prod id obsf-resource*)
(*map-spmf* (*map-prod id*) ($\lambda resF. case\ resF\ of\ OK\ res' \Rightarrow res' \mid Fault \Rightarrow$
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*) ($\lambda resF. obsf-resource\ (case\ resF\ of\ OK\ res' \Rightarrow res' \mid$
Fault $\Rightarrow fail-resource)$))
(*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* \mathcal{I} (*obsf-resource res*)

by(*coinduction arbitrary: res*) *auto*

lemma *WT-obsf-resource* [*WT-intro, simp*]: *exception- \mathcal{I}* $\mathcal{I} \vdash res\ obsf-resource\ res$
 \checkmark **if** $\mathcal{I} \vdash res\ res\ \checkmark$

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'* ($\lambda x. x = \text{Some True}$) *id option-of-exception*
(*gpv-stop* \mathcal{D})

lemma *WT-obsf-distinguisher* [*WT-intro*]:
exception- \mathcal{I} $\mathcal{I} \vdash_g$ *obsf-distinguisher* $\mathcal{A} \checkmark$ **if** [*WT-intro*]: $\mathcal{I} \vdash_g \mathcal{A} \checkmark$
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 $\mathcal{I} \mathcal{A} \mathcal{I} \vdash_g \mathcal{A} \checkmark$*
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) = ($\lambda x.$
map-gpv (map-prod id obsf-converter) id
(*map-gpv* ($\lambda convF. \text{case } convF \text{ of } Fault \Rightarrow (Fault, fail-converter) \mid OK (a, conv')$)
⇒ (*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' ($\lambda y. \text{case } y \text{ of } None \Rightarrow (Fault, const-converter Fault) \mid Some(x,$

```

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) 1 {(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, sim-
plified 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*: $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv} \checkmark$
and *WT-resource*: $\mathcal{I}' \vdash_{\text{res}} \text{res} \checkmark$
shows $\text{outs-}\mathcal{I} \mathcal{I} \vdash_R \text{attach} (\text{obsf-converter } \text{conv}) (\text{obsf-resource } \text{res}) \sim \text{obsf-resource}$
(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* $(\lambda((b, \text{conv}'), \text{res}'). (b, \text{conv}' \triangleright \text{res}'))$
(exec-gpv run-resource
(TRY map-gpv' (case-option (Fault, const-converter Fault) $(\lambda(x, \text{conv}'). (OK$
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 $\dots = \text{TRY map-spmf ?attach (exec-gpv run-resource ?gpv (obsf-resource$
res)) ELSE return-spmf (Fault, const-resource Fault)
by(*rule run-lossless-resource.exec-gpv-try-gpv*[**where** $\mathcal{I} = \text{exception-}\mathcal{I} \mathcal{I}'$])
(use eq-resource-on in <auto intro!: WT-gpv-map-gpv' WT-gpv-stop pfi-
nite-gpv-stop[THEN iffD2] dest: WT-converterD pfinite-converterD lossless-resourceD)
also have $\dots = \text{TRY map-spmf } (\lambda(\text{rc}, \text{res}'). \text{case } \text{rc} \text{ of } \text{None} \Rightarrow (\text{Fault},$
const-resource Fault) | Some (x, conv') \Rightarrow (OK x, obsf-converter conv' \triangleright \text{res}'))
((exec-gpv run-resource (map-gpv' id id option-of-exception (gpv-stop
(run-converter conv out))) (obsf-resource res)) \uparrow \{(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: op-
tion.split)
also have $\dots = \text{TRY map-spmf ?f (map-spmf (map-prod Some obsf-resource)$
(exec-gpv run-resource (run-converter conv out) res)) ELSE ?z
by(*simp only: exec-gpv-obsf-resource*)
also have *rel-spmf* *?X* \dots *?rhs* **using** *eq-resource-on*
by(*auto simp add: spmf.map-comp o-def spmf-rel-map intro!: rel-spmf-try-spmf*
rel-spmf-refl)
(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-I \mathcal{I} , *exception-I* $\mathcal{I}' \vdash_C$ *obsf-converter* \surd **if** $\mathcal{I}, \mathcal{I}' \vdash_C$ \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* \equiv *inline1*
and *inline1b* \equiv *inline1*
shows *bind-spmf* (*inline1a run-converter* (*map-gpv' id id option-of-exception*
(*gpv-stop gpv*)) (*obsf-converter conv*))
 $(\lambda xy. \text{case } xy \text{ of } \text{Inl } (None, conv') \Rightarrow \text{return-pmf } None \mid \text{Inl } (Some\ x, conv') \Rightarrow \text{return-spmf } (\text{Inl } (x, conv') \mid \text{Inr } y \Rightarrow \text{return-spmf } (\text{Inr } y)) =$
 $\text{map-spmf } (\text{map-sum } (\text{apsnd } \text{obsf-converter}))$
 $(\text{apsnd } (\text{map-prod } (\lambda rpv \text{ input. case input of } Fault \Rightarrow Done\ (Fault, \text{const-converter } Fault) \mid OK\ \text{input}' \Rightarrow$
 $\text{map-gpv}'\ (\lambda res. \text{case res of } None \Rightarrow (Fault, \text{const-converter } Fault) \mid$
 $Some\ (x, conv') \Rightarrow (OK\ x, \text{obsf-converter } conv')) \text{ id option-of-exception } (\text{try-gpv}$
 $(\text{gpv-stop } (rpv\ \text{input}'))\ (Done\ None))))$
 $(\lambda rpv \text{ input. case input of } Fault \Rightarrow Done\ None \mid OK\ \text{input}' \Rightarrow \text{map-gpv}'\ \text{id id}$
 $\text{option-of-exception } (\text{gpv-stop } (rpv\ \text{input}'))))))$
(*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]*)
 $(\text{auto split!: option.split intro!: ord-spmf-bindI[OF step.hyps, THEN}$
 $\text{spmf.leq-trans}] \text{ord-spmf-reflI})$
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])
 $(\text{auto simp add: fun-eq-iff map'-try-gpv split: exception.split})$
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] in-
line1a-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))) (λx. 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 (λxyz. case xyz of Inl (x, conv) ⇒ Pure (Some x, conv)
| Inr (out, rpv, rpv') ⇒ IO out (λinput. 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))

```

```

(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)))

```

(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 (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 op-
tion-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) (λy. 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-refl)
  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 (λp. 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)) (λx. case x of None ⇒ Fail | Some x ⇒ Done x)
    let ?gpv2' = bind-gpv (map-gpv' id id option-of-exception (gpv-stop (rpv
input))) (λx. case x of None ⇒ Fail | Some x ⇒ Done x)
    let ?f = λx. bind-gpv (inline run-converter (map-gpv' id id op-
tion-of-exception (gpv-stop (rpv' (fst x)))) (obsf-converter (snd x))) (λp. case fst p
of None ⇒ Fail | Some x' ⇒ Done (Some x', snd p))
    let ?f' = 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))))
    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=λx. bind-gpv (inline -
x -) -] option.case-distrib[where h=λx. bind-gpv (map-gpv' - - - x) -] cong: excep-
tion.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:  $\mathcal{I}, \mathcal{I}' \vdash_C \text{conv1} \checkmark, \mathcal{I}'' \vdash_C \text{conv2} \checkmark$ 
and pfinite1: pfinite-converter  $\mathcal{I} \mathcal{I}' \text{conv1}$ 
shows exception- $\mathcal{I} \mathcal{I}, \text{exception-}\mathcal{I} \mathcal{I}'' \vdash_C \text{obsf-converter (comp-converter conv1}
\text{conv2)} \sim \text{comp-converter (obsf-converter conv1) (obsf-converter conv2)}$ 
using WT pfinite1 supply eq- $\mathcal{I}$ -gpv-map-gpv[simp del]

```

```

proof(coinduction arbitrary: conv1 conv2)
  case eq-I-converter
  show ?case (is eq-I-gpv ?X - ?lhs ?rhs)
  proof -
    have eq-I-gpv (=) (exception-I I') ?rhs (TRY map-gpv ( $\lambda((b, conv1'), conv2')$ ).
    (b, conv1'  $\odot$  conv2') id
      (inline run-converter
        (map-gpv'
          (case-option (Fault, const-converter Fault)
            ( $\lambda(x, conv')$ . (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-I-gpv - - - ?rhs2 is eq-I-gpv - - - (try-gpv (map-gpv ?f - ?inline) ?else))
      using eq-I-converter
      apply simp
      apply(rule run-colossless-converter.inline-try-gpv[where I=exception-I 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))
        ( $\lambda(x, conv')$ . case x of None  $\Rightarrow$  Fail | Some x'  $\Rightarrow$  Done (x, conv'))

    also have ?rhs2 = try-gpv (map-gpv ?f id
      (map-gpv ( $\lambda(xo, conv')$ . 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))
        ( $\lambda(x, conv')$ . 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= $\lambda x. try-gpv x -$ ] 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 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 ( $\lambda(xo, conv')$ . 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=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 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
  applyclarsimp
  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
  applyclarsimp

```

```

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-I-converter-reflI)
apply simp
done
finally (eq-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-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* ($\lambda x.$ *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 ∉ 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 $A \vdash_C \text{callee1}(s) \approx \text{callee2}(s')$
shows $A \vdash_C \text{obsf-oracle callee1}(OK s) \approx \text{obsf-oracle callee2}(OK s')$
using *assms[THEN trace-callee-eq-obsf-oracleI]* **by** *simp*

lemma *trace-eq-obsf-resourceI*:

```

assumes  $A \vdash_R \text{res1} \approx \text{res2}$ 
shows  $A \vdash_R \text{obsf-resource } \text{res1} \approx \text{obsf-resource } \text{res2}$ 
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  $\text{eg1} \equiv \text{exec-gpv}$  and  $\text{eg2} \equiv \text{exec-gpv}$  shows
    spmf (map-spmf fst ( $\text{eg1}$  (obsf-oracle oracle) (obsf-distinguisher gpv) (OK s)))
  True =
    spmf (map-spmf fst ( $\text{eg2}$  oracle gpv s)) True
  (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs  $\leq$  ?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  $\mathcal{A}$ ) (obsf-resource res)) True = spmf
(connect  $\mathcal{A}$  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  $\mathcal{A}$ ) (obsf-resource ideal-resource) (obsf-resource real-resource)
=
  advantage  $\mathcal{A}$  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*)
 $foldl\text{-}spmf :: ('b \Rightarrow 'a \Rightarrow 'b\ spmf) \Rightarrow 'b\ spmf \Rightarrow 'a\ list \Rightarrow 'b\ spmf$
where
 $foldl\text{-}spmf\text{-}Nil: foldl\text{-}spmf\ f\ p\ [] = p$
 $| foldl\text{-}spmf\text{-}Cons: foldl\text{-}spmf\ f\ p\ (x \# xs) = foldl\text{-}spmf\ f\ (bind\text{-}spmf\ p\ (\lambda a. f\ a\ x))\ xs$

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

lemma *foldl-spmf-bind-spmf*: $foldl\text{-}spmf\ f\ (bind\text{-}spmf\ p\ g)\ xs = bind\text{-}spmf\ p\ (\lambda a. foldl\text{-}spmf\ f\ (g\ a)\ xs)$
by(*induction xs arbitrary: g simp-all*)

lemma *bind-foldl-spmf-return*:
 $bind\text{-}spmf\ p\ (\lambda x. foldl\text{-}spmf\ f\ (return\text{-}spmf\ x)\ xs) = foldl\text{-}spmf\ f\ p\ xs$
by(*simp add: foldl-spmf-bind-spmf[symmetric]*)

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

lemma *foldl-spmf-identity* [*simp*]: $foldl\text{-}spmf\ (\lambda s\ x. return\text{-}spmf\ s)\ p\ xs = p$
by(*induction xs arbitrary: p simp-all*)

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

lemma *foldl-spmf-Cons'*:
 $foldl\text{-}spmf\ f\ (return\text{-}spmf\ a)\ (x \# xs) = bind\text{-}spmf\ (f\ a\ x)\ (\lambda a'. foldl\text{-}spmf\ f\ (return\text{-}spmf\ a')\ xs)$
by(*simp add: bind-foldl-spmf-return*)

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

lemma
foldl-spmf-helper:
assumes $\bigwedge x. h\ (f\ x) = x$
assumes $\bigwedge x. f\ (h\ x) = x$
shows $foldl\text{-}spmf\ (\lambda a\ e. map\text{-}spmf\ h\ (g\ (f\ a)\ e))\ acc\ es =$

```

    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 ( $\lambda a e. \text{map-spmf } (f (p a)) (g (q a) e)$ ) acc es =
    bind-spmf acc ( $\lambda acc'. \text{map-spmf } (f (p acc')) (\text{foldl-spmf } g (\text{return-spmf } (q acc'))$ )
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 ( $\lambda s e. \text{map-prod } (\lambda-. c) (\lambda r. f r e) s$ ) (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 ( $\lambda(l, r) e. \text{map-spmf } (\lambda l'. (l', r)) (f l e)$ ) (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 ( $\lambda(l, -) e. \text{map-spmf } (\lambda l'. (l', c')) (f l e)$ ) (return-spmf (l, c)) es =
    map-spmf ( $\lambda l'. (l', \text{if } es = [] \text{ then } c \text{ else } c')$ ) (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 ( $\lambda s e. \text{map-prod } (\lambda l. f l e) (\lambda-. c) s$ ) (sl, c) l =

```

Pair (foldl (λs 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 (λr'. (l, r')) (f r e)) (return-spmf (l, r)) es =
map-spmf (λr'. (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 (λr'. (c', r')) (f r e)) (return-spmf (c, r)) es =
map-spmf (λr'. (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 (λx. bind-spmf (foldl-spmf g init (events x)) (λy.*
h x y)) =

bind-spmf (bind-spmf f (λx. 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
  >=> (λs-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 \Rightarrow (unit, 'a, 'b) gpv **where**
pauses [] = Done ()
| *pauses* (x # xs) = Pause x (λ -. *pauses* xs)

lemma *WT-gpv-pauses* [*WT-intro*]:
 $\mathcal{I} \vdash_g$ *pauses* xs \surd **if** set xs \subseteq 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 *eI* :: ('a, 'b) $\mathcal{I} \Rightarrow$ ('a, 'b \times 'c) \mathcal{I} **is** $\lambda \mathcal{I} x. \mathcal{I} x \times UNIV$.

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

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

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

lemma *eI-inverse [simp]*: *map-I* id *fst* (*eI* \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) *oracle* = ('state, 'input, 'output \times 'event list)

oracle'

definition

parallel-eoracle ::
(*'s1, 'e1, 'i1, 'o1*) *eoracle* \Rightarrow (*'s2, 'e2, 'i2, 'o2*) *eoracle* \Rightarrow
(*'s1* \times *'s2, 'e1* + *'e2, 'i1* + *'i2, 'o1* + *'o2*) *eoracle*

where

parallel-eoracle eoracle1 eoracle2 state \equiv
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* \Rightarrow (*'s, 'e2, 'i2, 'o2*) *eoracle* \Rightarrow
(*'s, 'e1* + *'e2, 'i1* + *'i2, 'o1* + *'o2*) *eoracle*

where

plus-eoracle eoracle1 eoracle2 state \equiv
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'* \Rightarrow (*'s-event* \times *'s, 'e1, 'i, 'o*) *eoracle* \Rightarrow
(*'s-event* \times *'s, 'e2, 'i, 'o*) *eoracle*

where

translate-eoracle translator eoracle state inp \equiv
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*) \leftarrow *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

$(\text{'state}, \text{'event}) \text{ handler} = \text{'state} \Rightarrow \text{'event} \Rightarrow \text{'state} \text{ spmf}$

fun

$\text{parallel-handler} :: (\text{'s1}, \text{'e1}) \text{ handler} \Rightarrow (\text{'s2}, \text{'e2}) \text{ handler} \Rightarrow (\text{'s1} \times \text{'s2}, \text{'e1} + \text{'e2}) \text{ handler}$

where

$\text{parallel-handler left - s (Inl e1)} = \text{map-spmf } (\lambda s1'. (s1', \text{snd } s)) (\text{left (fst } s) \text{ e1})$
 $| \text{parallel-handler - right s (Inr e2)} = \text{map-spmf } (\lambda s2'. (\text{fst } s, s2')) (\text{right (snd } s) \text{ e2})$

definition

$\text{plus-handler} :: (\text{'s}, \text{'e1}) \text{ handler} \Rightarrow (\text{'s}, \text{'e2}) \text{ handler} \Rightarrow (\text{'s}, \text{'e1} + \text{'e2}) \text{ handler}$

where

$\text{plus-handler left right s} \equiv \text{case-sum (left } s) (\text{right } s)$

lemma parallel-handler-left:

$\text{map-fun id (map-fun Inl id) (parallel-handler left right)} =$
 $(\lambda (s-l, s-r) q. \text{map-spmf } (\lambda s-l'. (s-l', s-r)) (\text{left } s-l \text{ } q))$

by force

lemma parallel-handler-right:

$\text{map-fun id (map-fun Inr id) (parallel-handler left right)} =$
 $(\lambda (s-l, s-r) q. \text{map-spmf } (\lambda s-r'. (s-l, s-r')) (\text{right } s-r \text{ } q))$

by force

lemma in-set-spmf-parallel-handler:

$s' \in \text{set-spmf (parallel-handler left right s } x) \iff$
 $(\text{case } x \text{ of Inl } e \Rightarrow \text{fst } s' \in \text{set-spmf (left (fst } s) \text{ } e) \wedge \text{snd } s' = \text{snd } s$
 $| \text{Inr } e \Rightarrow \text{snd } s' \in \text{set-spmf (right (snd } s) \text{ } e) \wedge \text{fst } s' = \text{fst } s)$

by(cases s; cases s'; auto split: sum.split)

4.3 Fused Resource Construction

codatatype

$(\text{'s-core}, \text{'event}, \text{'iadv-core}, \text{'iusr-core}, \text{'oadv-core}, \text{'ousr-core}) \text{ core} =$
 Core

$(\text{cpoke}: (\text{'s-core}, \text{'event}) \text{ handler})$

$(\text{cfunc-adv}: (\text{'s-core}, \text{'iadv-core}, \text{'oadv-core}) \text{ oracle})$

$(\text{cfunc-usr}: (\text{'s-core}, \text{'iusr-core}, \text{'ousr-core}) \text{ oracle})$

declare $\text{core.sel-transfer}[\text{transfer-rule del}]$

declare $\text{core.ctr-transfer}[\text{transfer-rule del}]$

declare $\text{core.case-transfer}[\text{transfer-rule del}]$

context

includes lifting-syntax

begin

inductive

```

rel-core':::
  ('s-core  $\Rightarrow$  's-core'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('event  $\Rightarrow$  'event'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('iadv-core  $\Rightarrow$  'iadv-core'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('iusr-core  $\Rightarrow$  'iusr-core'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('oadv-core  $\Rightarrow$  'oadv-core'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('ousr-core  $\Rightarrow$  'ousr-core'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('s-core, 'event, 'iadv-core, 'iusr-core, 'oadv-core, 'ousr-core) core  $\Rightarrow$ 
  ('s-core', 'event', 'iadv-core', 'iusr-core', 'oadv-core', 'ousr-core') core  $\Rightarrow$  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  $\implies$  E  $\implies$  rel-spmf S) cpoke cpoke' and
  (S  $\implies$  IA  $\implies$  rel-spmf (rel-prod OA S)) cfunc-adv cfunc-adv' and
  (S  $\implies$  IU  $\implies$  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 rela-
tor-eq)
done

```

lemma

```

rel-core'-mono [relator-mono]:
  rel-core' S E IA IU OA OU  $\leq$  rel-core' S E' IA' IU' OA' OU'
if E'  $\leq$  E IA'  $\leq$  IA IU'  $\leq$  IU OA  $\leq$  OA' OU  $\leq$  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]:

```

$(rel\text{-}core' S E IA IU OA OU \implies S \implies E \implies rel\text{-}spmf S)$ *cpoke*

by(rule *rel-funI*; erule *rel-core'.cases*; simp)

lemma

cfunc-adv-parametric [*transfer-rule*]:

$(rel\text{-}core' S E IA IU OA OU \implies S \implies IA \implies rel\text{-}spmf (rel\text{-}prod OA S))$ *cfunc-adv cfunc-adv*

by(rule *rel-funI*; erule *rel-core'.cases*; simp)

lemma

cfunc-usr-parametric [*transfer-rule*]:

$(rel\text{-}core' S E IA IU OA OU \implies S \implies IU \implies rel\text{-}spmf (rel\text{-}prod OU S))$ *cfunc-usr cfunc-usr*

by(rule *rel-funI*; erule *rel-core'.cases*; simp)

lemma

Core-parametric [*transfer-rule*]:

$((S \implies E \implies rel\text{-}spmf S) \implies (S \implies IA \implies rel\text{-}spmf (rel\text{-}prod OA S))) \implies (S \implies IU \implies rel\text{-}spmf (rel\text{-}prod OU S))$

$\implies rel\text{-}core' S E IA IU OA OU$ *Core Core*

by(rule *rel-funI*) + *simp*

lemma

case-core-parametric [*transfer-rule*]:

$((S \implies E \implies rel\text{-}spmf S) \implies$

$(S \implies IA \implies rel\text{-}spmf (rel\text{-}prod OA S)) \implies$

$(S \implies IU \implies rel\text{-}spmf (rel\text{-}prod OU S)) \implies X) \implies$

$rel\text{-}core' S E IA IU OA OU \implies X)$ *case-core case-core*

by(rule *rel-funI*) + (*auto 4 4 split: core.split dest: rel-funD*)

lemma

corec-core-parametric [*transfer-rule*]:

$((X \implies S \implies E \implies rel\text{-}spmf S) \implies$

$(X \implies S \implies IA \implies rel\text{-}spmf (rel\text{-}prod OA S)) \implies$

$(X \implies S \implies IU \implies rel\text{-}spmf (rel\text{-}prod OU S)) \implies$

$X \implies rel\text{-}core' S E IA IU OA OU)$ *corec-core corec-core*

by(rule *rel-funI*) + (*auto simp add: core.corec dest: rel-funD*)

primcorec *map-core'* ::

$('event' \Rightarrow 'event) \Rightarrow$

$('iadv\text{-}core' \Rightarrow 'iadv\text{-}core) \Rightarrow$

$('iusr\text{-}core' \Rightarrow 'iusr\text{-}core) \Rightarrow$

$('oadv\text{-}core \Rightarrow 'oadv\text{-}core') \Rightarrow$

$('ousr\text{-}core \Rightarrow 'ousr\text{-}core') \Rightarrow$

$('s\text{-}core, 'event, 'iadv\text{-}core, 'iusr\text{-}core, 'oadv\text{-}core, 'ousr\text{-}core) core \Rightarrow$

$('s\text{-}core, 'event', 'iadv\text{-}core', 'iusr\text{-}core', 'oadv\text{-}core', 'ousr\text{-}core') core$

where

$cpoke (map\text{-}core' e ia iu oa ou core) = (id \dashrightarrow e \dashrightarrow id) (cpoke core)$

| *cfunc-adv* (*map-core'* *e ia iu oa ou core*) = (*id* \dashrightarrow *ia* \dashrightarrow *map-spmf*
(*map-prod oa id*)) (*cfunc-adv core*)
| *cfunc-usr* (*map-core'* *e ia iu oa ou core*) = (*id* \dashrightarrow *iu* \dashrightarrow *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*) *I* \Rightarrow (*'iusr, 'ousr*) *I* \Rightarrow (*'s* \Rightarrow *bool*) \Rightarrow (*'s,*
'event, 'iadv, 'iusr, 'oadv, 'ousr) *core* \Rightarrow *bool*

for *I-adv I-usr I core* **where**

WT-core I-adv I-usr I core **if**

$\bigwedge s e s'. \llbracket s' \in \text{set-spmf } (\text{cpoke core } s e); I s \rrbracket \Longrightarrow I s'$

$\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{cfunc-adv core } s x); x \in \text{outs-}I \text{ } I\text{-adv}; I s \rrbracket \Longrightarrow$
 $y \in \text{responses-}I \text{ } I\text{-adv } x \wedge I s'$

$\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{cfunc-usr core } s x); x \in \text{outs-}I \text{ } I\text{-usr}; I s \rrbracket \Longrightarrow y$
 $\in \text{responses-}I \text{ } I\text{-usr } x \wedge I s'$

lemma *WT-coreD*:

assumes *WT-core I-adv I-usr I core*

shows *WT-coreD-cpoke*: $\bigwedge s e s'. \llbracket s' \in \text{set-spmf } (\text{cpoke core } s e); I s \rrbracket \Longrightarrow I s'$

and *WT-coreD-cfunc-adv*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{cfunc-adv core } s x);$
 $x \in \text{outs-}I \text{ } I\text{-adv}; I s \rrbracket \Longrightarrow y \in \text{responses-}I \text{ } I\text{-adv } x \wedge I s'$

and *WT-coreD-cfund-usr*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{cfunc-usr core } s x);$
 $x \in \text{outs-}I \text{ } I\text{-usr}; I s \rrbracket \Longrightarrow y \in \text{responses-}I \text{ } I\text{-usr } x \wedge I s'$

using *assms* **by**(*auto elim!*: *WT-core.cases*)

lemma *WT-coreD-foldl-spmf-cpoke*:

assumes *WT-core I-adv I-usr I core*

and $s' \in \text{set-spmf } (\text{foldl-spmf } (\text{cpoke core}) p es)$

and $\forall s \in \text{set-spmf } p. I s$

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:  $\bigwedge s. \mathcal{I}\text{-adv} \vdash c \text{ cfunc-adv core } s \checkmark$ 
and usr:  $\bigwedge s. \mathcal{I}\text{-usr} \vdash c \text{ cfunc-usr core } s \checkmark$ 
shows WT-core  $\mathcal{I}\text{-adv} \mathcal{I}\text{-usr} (\lambda\cdot. \text{True}) \text{ core}$ 
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  $\Rightarrow$  's-rest'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('event  $\Rightarrow$  'event'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('iadv-rest  $\Rightarrow$  'iadv-rest'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('iusr-rest  $\Rightarrow$  'iusr-rest'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('oadv-rest  $\Rightarrow$  'oadv-rest'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('ousr-rest  $\Rightarrow$  'ousr-rest'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('more  $\Rightarrow$  'more'  $\Rightarrow$  bool)  $\Rightarrow$ 
  ('s-rest, 'event, 'iadv-rest, 'iusr-rest, 'oadv-rest, 'ousr-rest, 'more) rest-scheme
 $\Rightarrow$ 
  ('s-rest', 'event', 'iadv-rest', 'iusr-rest', 'oadv-rest', 'ousr-rest', 'more') rest-scheme
 $\Rightarrow$  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  $\text{====>}$  IA  $\text{====>}$  rel-spmf (rel-prod (rel-prod OA (list-all2 E)) S)) rfunc-adv
rfunc-adv' and
  (S  $\text{====>}$  IU  $\text{====>}$  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]:

($M \implies (S \implies IA \implies \text{rel-spmf } (\text{rel-prod } (\text{rel-prod } OA \text{ (list-all2 } E)) S))$)
 $\implies (S \implies IU \implies \text{rel-spmf } (\text{rel-prod } (\text{rel-prod } OU \text{ (list-all2 } E)) S))$
 $\implies \text{rel-rest}' S E IA IU OA OU M$) *Rest Rest*
by(rule *rel-funI*) + *simp*

lemma *case-rest-scheme-parametric* [transfer-rule]:

(($M \implies (S \implies IA \implies \text{rel-spmf } (\text{rel-prod } (\text{rel-prod } OA \text{ (list-all2 } E)) S)) \implies (S \implies IU \implies \text{rel-spmf } (\text{rel-prod } (\text{rel-prod } OU \text{ (list-all2 } E)) S)) \implies X$)
 $\implies \text{rel-rest}' S E IA IU OA OU M \implies X$) *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]:

(($X \implies M$)
 $(X \implies S \implies IA \implies \text{rel-spmf } (\text{rel-prod } (\text{rel-prod } OA \text{ (list-all2 } E)) S)) \implies (X \implies S \implies IU \implies \text{rel-spmf } (\text{rel-prod } (\text{rel-prod } OU \text{ (list-all2 } E)) S)) \implies X$)
 $X \implies \text{rel-rest}' S E IA IU OA OU M$) *corec-rest-scheme corec-rest-scheme*
by(rule *rel-funI*)+(auto *simp add: rest-scheme.corec dest: rel-funD*)

primcorec *map-rest'* ::

(*'event* \Rightarrow *'event'*) \Rightarrow
(*'iadv-rest'* \Rightarrow *'iadv-rest'*) \Rightarrow
(*'iusr-rest'* \Rightarrow *'iusr-rest'*) \Rightarrow
(*'oadv-rest'* \Rightarrow *'oadv-rest'*) \Rightarrow
(*'ousr-rest'* \Rightarrow *'ousr-rest'*) \Rightarrow
(*'more* \Rightarrow *'more'*) \Rightarrow
(*'s-rest, 'event, 'iadv-rest, 'iusr-rest, 'oadv-rest, 'ousr-rest, 'more*) *rest-scheme*
 \Rightarrow
(*'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* \dashrightarrow *ia* \dashrightarrow *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* \dashrightarrow *iu* \dashrightarrow *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) $\mathcal{I} \Rightarrow$ ('iusr, 'ousr) $\mathcal{I} \Rightarrow$ ('s \Rightarrow bool) \Rightarrow ('s, 'event, 'iadv, 'iusr, 'oadv, 'ousr) rest-wstate \Rightarrow bool

```

for  $\mathcal{I}$ -adv  $\mathcal{I}$ -usr  $\mathcal{I}$  rest where
  WT-rest  $\mathcal{I}$ -adv  $\mathcal{I}$ -usr  $\mathcal{I}$  rest if
     $\bigwedge s x y es s'. \llbracket ((y, es), s') \in \text{set-spmf} (\text{rfunc-adv rest } s x); x \in \text{outs-}\mathcal{I} \mathcal{I}\text{-adv}; I s \rrbracket \implies y \in \text{responses-}\mathcal{I} \mathcal{I}\text{-adv } x \wedge I s'$ 
     $\bigwedge s x y es s'. \llbracket ((y, es), s') \in \text{set-spmf} (\text{rfunc-usr rest } s x); x \in \text{outs-}\mathcal{I} \mathcal{I}\text{-usr}; I s \rrbracket \implies y \in \text{responses-}\mathcal{I} \mathcal{I}\text{-usr } x \wedge I s'$ 
     $I (\text{rinit rest})$ 

```

lemma *WT-restD*:

```

assumes WT-rest  $\mathcal{I}$ -adv  $\mathcal{I}$ -usr  $\mathcal{I}$  rest
shows WT-restD-rfunc-adv:  $\bigwedge s x y es s'. \llbracket ((y, es), s') \in \text{set-spmf} (\text{rfunc-adv rest } s x); x \in \text{outs-}\mathcal{I} \mathcal{I}\text{-adv}; I s \rrbracket \implies y \in \text{responses-}\mathcal{I} \mathcal{I}\text{-adv } x \wedge I s'$ 
and WT-restD-rfunc-usr:  $\bigwedge s x y es s'. \llbracket ((y, es), s') \in \text{set-spmf} (\text{rfunc-usr rest } s x); x \in \text{outs-}\mathcal{I} \mathcal{I}\text{-usr}; I s \rrbracket \implies y \in \text{responses-}\mathcal{I} \mathcal{I}\text{-usr } x \wedge I s'$ 
and WT-restD-rinit:  $I (\text{rinit rest})$ 
using assms by(auto elim!: WT-rest.cases)

```

abbreviation

```

fuse-cfunc ::
('o  $\Rightarrow$  'x)  $\Rightarrow$  ('s-core, 'i, 'o) oracle'  $\Rightarrow$  ('s-core  $\times$  's-rest, 'i, 'x) oracle'
where
  fuse-cfunc redirect cfunc state inp  $\equiv$  do {
    let handle = map-prod redirect (prod.swap o Pair (snd state));
      (os-cfunc :: 'o  $\times$  's-core)  $\leftarrow$  cfunc (fst state) inp;
      return-spmf (handle os-cfunc)
    }

```

abbreviation

```

fuse-rfunc ::
  ('o ⇒ 'x) ⇒ ('s-rest, 'e, 'i, 'o) eoracle ⇒ ('s-core, 'e) handler ⇒
  ('s-core × 's-rest, 'i, 'x) oracle'
where
fuse-rfunc redirect rfunc notify state inp ≡
  bind-spmf
    (rfunc (snd state) inp)
  (λ((o-rfunc, e-lst), s-rfunc).
    bind-spmf
      (foldl-spmf notify (return-spmf (fst state)) e-lst)
      (λs-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 ⇒
    ('s-core × '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  $\Rightarrow$ 
  (('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)  $\vdash_{res}$  resource
rest  $\checkmark$ 
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  $\times$  '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 (λs-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 1I 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 1I 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 ⊕I I-adv2) (I-usr1 ⊕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 = λorc (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 = λorc (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

```

$map\text{-}spmf\ eorc\text{-}of\ (exec\text{-}gpv\ (orc\text{-}of\ (rfunc\text{-}usr\ rest))\ (cnu\text{-}usr\ s\text{-}usr\ iusr))\ (s\text{-}rest,\ [])$

end

lemma

$attach\text{-}rest\text{-}id\text{-}oracle\text{-}adv: rfunc\text{-}adv\ (attach\text{-}rest\ 1_I\ cnu\ f\text{-}init\ rest) =$
 $(\lambda(s\text{-}cnu,\ s\text{-}core)\ q.\ map\text{-}spmf\ (\lambda(out,\ s\text{-}core').\ (out,\ s\text{-}cnu,\ s\text{-}core'))\ (rfunc\text{-}adv\ rest\ s\text{-}core\ q))$

by($simp\ add: id\text{-}oracle\text{-}def\ split\text{-}def\ map\text{-}spmf\ conv\text{-}bind\text{-}spmf\ fun\text{-}eq\text{-}iff$)

lemma

$attach\text{-}rest\text{-}id\text{-}oracle\text{-}usr: rfunc\text{-}usr\ (attach\text{-}rest\ cnu\ 1_I\ f\text{-}init\ rest) =$
 $(\lambda(s\text{-}cnu,\ s\text{-}core)\ q.\ map\text{-}spmf\ (\lambda(out,\ s\text{-}core').\ (out,\ s\text{-}cnu,\ s\text{-}core'))\ (rfunc\text{-}usr\ rest\ s\text{-}core\ q))$

by($simp\ add: id\text{-}oracle\text{-}def\ split\text{-}def\ map\text{-}spmf\ conv\text{-}bind\text{-}spmf$)

5 Traces

type-synonym ($'event,\ 'iadv\text{-}core,\ 'iusr\text{-}core,\ 'oadv\text{-}core,\ 'ousr\text{-}core$) $trace\text{-}core =$
 $('event + 'iadv\text{-}core \times 'oadv\text{-}core + 'iusr\text{-}core \times 'ousr\text{-}core)\ list$
 $\Rightarrow ('event \Rightarrow real)$
 $\times ('iadv\text{-}core \Rightarrow 'oadv\text{-}core\ spmf)$
 $\times ('iusr\text{-}core \Rightarrow 'ousr\text{-}core\ spmf)$

context

fixes $core :: ('s\text{-}core,\ 'event,\ 'iadv\text{-}core,\ 'iusr\text{-}core,\ 'oadv\text{-}core,\ 'ousr\text{-}core)\ core$

begin

primrec $trace\text{-}core' :: 's\text{-}core\ spmf \Rightarrow ('event,\ 'iadv\text{-}core,\ 'iusr\text{-}core,\ 'oadv\text{-}core,\ 'ousr\text{-}core)\ trace\text{-}core$ **where**

$trace\text{-}core'\ S\ [] =$

$(\lambda e.\ weight\text{-}spmf'\ (bind\text{-}spmf\ S\ (\lambda s.\ cpoke\ core\ s\ e)),$

$\lambda ia.\ bind\text{-}spmf\ S\ (\lambda s.\ map\text{-}spmf\ fst\ (cfunc\text{-}adv\ core\ s\ ia)),$

$\lambda iu.\ bind\text{-}spmf\ S\ (\lambda s.\ map\text{-}spmf\ fst\ (cfunc\text{-}usr\ core\ s\ iu)))$

| $trace\text{-}core'\ S\ (obs\ \# tr) = (case\ obs\ of$

$Inl\ e \Rightarrow trace\text{-}core'\ (mk\text{-}lossless\ (bind\text{-}spmf\ S\ (\lambda s.\ cpoke\ core\ s\ e)))\ tr$

| $Inr\ (Inl\ (ia,\ oa)) \Rightarrow trace\text{-}core'\ (cond\text{-}spmf\text{-}fst\ (bind\text{-}spmf\ S\ (\lambda s.\ cfunc\text{-}adv\ core\ s\ ia))\ oa)\ tr$

| $Inr\ (Inr\ (iu,\ ou)) \Rightarrow trace\text{-}core'\ (cond\text{-}spmf\text{-}fst\ (bind\text{-}spmf\ S\ (\lambda s.\ cfunc\text{-}usr\ core\ s\ iu))\ ou)\ tr$

)

end

declare $trace\text{-}core'.simps\ [simp\ del]$

case-of-simps $trace\text{-}core'\text{-}unfold: trace\text{-}core'.simps[unfolded\ weight\text{-}spmf'\text{-}def]$

simps-of-case $trace\text{-}core'\text{-}simps\ [simp]: trace\text{-}core'\text{-}unfold$

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*
 \Rightarrow (*'s-core', 'event, 'iadv-core, 'iusr-core, 'oadv-core, 'ousr-core*) *core*
 \Rightarrow *'event set* \Rightarrow *'iadv-core set* \Rightarrow *'iusr-core set*
 \Rightarrow *'s-core spmf* \Rightarrow *'s-core' spmf* \Rightarrow **bool** **where**
trace-core-eq core1 core2 E IA IU p q \longleftrightarrow
 $(\forall tr. set\ tr \subseteq E \langle + \rangle (IA \times UNIV) \langle + \rangle (IU \times UNIV) \longrightarrow$
 $rel-prod\ (eq-onp\ (\lambda e. e \in E) ==> (=))\ (rel-prod\ (eq-onp\ (\lambda ia. ia \in IA) ==> (=))$
 $(=))\ (eq-onp\ (\lambda iu. iu \in IU) ==> (=)))$
 $(trace-core'\ core1\ p\ tr)\ (trace-core'\ core2\ q\ tr))$

end

lemma *trace-core-eqD*:

assumes *trace-core-eq core1 core2 E IA IU p q*
and *set tr* $\subseteq E \langle + \rangle (IA \times UNIV) \langle + \rangle (IU \times UNIV)$
shows *trace-core-eqD-cpoke*:
 $e \in E \implies fst\ (trace-core'\ core1\ p\ tr)\ e = fst\ (trace-core'\ core2\ q\ tr)\ e$
and *trace-core-eqD-cfunc-adv*:
 $ia \in IA \implies fst\ (snd\ (trace-core'\ core1\ p\ tr))\ ia = fst\ (snd\ (trace-core'\ core2\ q\ tr))\ ia$
and *trace-core-eqD-cfunc-usr*:
 $iu \in IU \implies snd\ (snd\ (trace-core'\ core1\ p\ tr))\ iu = snd\ (snd\ (trace-core'\ core2\ q\ tr))\ iu$
using *assms* **by**(*auto simp add: trace-core-eq-def rel-fun-def eq-onp-def rel-prod-sel*)

lemma *trace-core-eqI*:

assumes $\bigwedge tr\ e. \llbracket set\ tr \subseteq E \langle + \rangle (IA \times UNIV) \langle + \rangle (IU \times UNIV); e \in E \rrbracket$
 $\implies fst\ (trace-core'\ core1\ p\ tr)\ e = fst\ (trace-core'\ core2\ q\ tr)\ e$
and $\bigwedge tr\ ia. \llbracket set\ tr \subseteq E \langle + \rangle (IA \times UNIV) \langle + \rangle (IU \times UNIV); ia \in IA \rrbracket$
 $\implies fst\ (snd\ (trace-core'\ core1\ p\ tr))\ ia = fst\ (snd\ (trace-core'\ core2\ q\ tr))\ ia$
and $\bigwedge tr\ iu. \llbracket set\ tr \subseteq E \langle + \rangle (IA \times UNIV) \langle + \rangle (IU \times UNIV); iu \in IU \rrbracket$
 $\implies snd\ (snd\ (trace-core'\ core1\ p\ tr))\ iu = snd\ (snd\ (trace-core'\ core2\ q\ tr))\ iu$
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 = $(\lambda-. 0, \lambda-. return-pmf\ None, \lambda-. return-pmf$

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=tr¹]*)
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*: $\bigwedge p q e. \llbracket S p q; e \in E \rrbracket \implies$
weight-spmf (bind-spmf p (λs. cpoke core1 s e)) = weight-spmf (bind-spmf q
(*λs. cpoke core2 s e*))
and *sim-cpoke*: $\bigwedge p q e. \llbracket S p q; e \in E \rrbracket \implies$
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*: $\bigwedge p q ia. \llbracket S p q; ia \in IA \rrbracket \implies$
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*: $\bigwedge p q ia s1 s2 s1' s2' oa. \llbracket S p q; ia \in IA;$
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) \rrbracket*
 $\implies S (cond-spmf-fst (bind-spmf p (λs1. cfunc-adv core1 s1 ia)) oa) (cond-spmf-fst$
(*bind-spmf q (λs2. cfunc-adv core2 s2 ia) oa*))
and *step-cfunc-usr*: $\bigwedge p q iu. \llbracket S p q; iu \in IU \rrbracket \implies$
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*: $\bigwedge p q iu s1 s2 s1' s2' ou. \llbracket S p q; iu \in IU;$
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) \rrbracket*
 $\implies S (cond-spmf-fst (bind-spmf p (λs1. cfunc-usr core1 s1 iu)) ou) (cond-spmf-fst$
(*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*)

```

 $\wedge$ 
  ( $\forall ia \in IA. fst (snd (trace-core' core1 p tr)) ia = fst (snd (trace-core' core2 q tr)) ia$ )  $\wedge$ 
  ( $\forall iu \in IU. snd (snd (trace-core' core1 p tr)) iu = snd (snd (trace-core' core2 q tr)) iu$ )
  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.prem(1) have tr: set tr  $\subseteq E <+> IA \times UNIV <+> IU \times UNIV$ 
  by simp
    from Cons.prem(1)
    consider (cpoke) e where a = Inl e e  $\in E$ 
      | (cfunc-adv) ia oa where a = Inr (Inl (ia, oa)) ia  $\in IA$ 
      | (cfunc-usr) iu ou where a = Inr (Inr (iu, ou)) iu  $\in IU$  by auto
    then show ?case
    proof cases
      case cpoke
      then show ?thesis using tr Cons.prem(2) by(auto simp add: sim-cpoke
intro!: Cons.IH)
    next
      case cfunc-adv
      let ?p = bind-spmf p ( $\lambda s1. cfunc-adv core1 s1 ia$ )
      let ?q = bind-spmf q ( $\lambda s2. cfunc-adv core2 s2 ia$ )
      show ?thesis
      proof(cases oa  $\in fst ' set-spmf ?p$ )
        case True
        with step-cfunc-adv[OF Cons.prem(2) cfunc-adv(2), THEN arg-cong[where
f=set-spmf]]
        have oa  $\in fst ' set-spmf ?q$ 
          unfolding set-map-spmf[symmetric] by(simp only: map-bind-spmf o-def)
        then show ?thesis using True Cons.prem 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.prem(2) cfunc-adv(2), THEN arg-cong[where
f=set-spmf]] False
        have oa': oa  $\notin fst ' set-spmf ?q$ 
          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.premis(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.premis 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.premis(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 =
 $(\lambda e. \text{weight-spmf } (\text{bind-spmf } (\text{trace-core-aux core p tr}) (\lambda s. \text{cpoke core s e})),$
 $\lambda ia. \text{bind-spmf } (\text{trace-core-aux core p tr}) (\lambda s. \text{map-spmf fst } (\text{cfunc-adv core s}$
 $ia)),$
 $\lambda iu. \text{bind-spmf } (\text{trace-core-aux core p tr}) (\lambda s. \text{map-spmf fst } (\text{cfunc-usr core s}$
 $iu)))$
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*

$:: ('s\text{-core}, 'event, 'iadv\text{-core}, 'iusr\text{-core}, 'oadv\text{-core}, 'ousr\text{-core}) \text{ core}$
 $\Rightarrow ('s\text{-core}', 'event', 'iadv\text{-core}', 'iusr\text{-core}', 'oadv\text{-core}', 'ousr\text{-core}') \text{ core}$
 $\Rightarrow 'event \text{ set} \Rightarrow 'iadv\text{-core set} \Rightarrow 'iusr\text{-core set}$
 $\Rightarrow 's\text{-core spmf} \Rightarrow 's\text{-core}' \text{ spmf} \Rightarrow 's\text{-core spmf} \Rightarrow 's\text{-core}' \text{ spmf} \Rightarrow \text{bool}$
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 $\text{set } tr \subseteq E \langle + \rangle IA \times UNIV \langle + \rangle IU \times UNIV$

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*:
 $e \in E \Longrightarrow \text{weight-spmf } (\text{bind-spmf } p' (\lambda s. \text{cpoke core1 s e})) = \text{weight-spmf}$
 $(\text{bind-spmf } q' (\lambda s. \text{cpoke core2 s e}))$
 (**is** *PROP ?thesis1*)
and *trace-core-closure-step-cfunc-adv*:
 $ia \in IA \Longrightarrow \text{bind-spmf } p' (\lambda s1. \text{map-spmf fst } (\text{cfunc-adv core1 s1 ia})) = \text{bind-spmf}$
 $q' (\lambda s2. \text{map-spmf fst } (\text{cfunc-adv core2 s2 ia}))$
 (**is** *PROP ?thesis2*)
and *trace-core-closure-step-cfunc-usr*:
 $iu \in IU \Longrightarrow \text{bind-spmf } p' (\lambda s1. \text{map-spmf fst } (\text{cfunc-usr core1 s1 iu})) = \text{bind-spmf}$
 $q' (\lambda s2. \text{map-spmf fst } (\text{cfunc-usr core2 s2 iu}))$
 (**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 $S \equiv \text{trace-core-closure } \text{core1 } \text{core2 } E \text{ IA } IU \text{ p } q$
assumes $S \text{ p' } q'$
shows *trace-core-closure-sim-cpoke:*
 $e \in E \implies S (\text{mk-lossless } (\text{bind-spmf } p' (\lambda s. \text{cpoke } \text{core1 } s \ e))) (\text{mk-lossless } (\text{bind-spmf } q' (\lambda s. \text{cpoke } \text{core2 } s \ e)))$
(is *PROP ?thesis1***)**
and *trace-core-closure-sim-cfunc-adv: ia ∈ IA*
 $\implies S (\text{cond-spmf-fst } (\text{bind-spmf } p' (\lambda s1. \text{cfunc-adv } \text{core1 } s1 \ ia)) \ oa) (\text{cond-spmf-fst } (\text{bind-spmf } q' (\lambda s2. \text{cfunc-adv } \text{core2 } s2 \ ia)) \ oa)$
(is *PROP ?thesis2***)**
and *trace-core-closure-sim-cfunc-usr: iu ∈ IU*
 $\implies S (\text{cond-spmf-fst } (\text{bind-spmf } p' (\lambda s1. \text{cfunc-usr } \text{core1 } s1 \ iu)) \ ou) (\text{cond-spmf-fst } (\text{bind-spmf } q' (\lambda s2. \text{cfunc-usr } \text{core2 } s2 \ iu)) \ ou)$
(is *PROP ?thesis3***)**

proof –
from *assms(2)* **obtain** *tr* **where** $p: p' = \text{trace-core-aux } \text{core1 } p \ tr$
and $q: q' = \text{trace-core-aux } \text{core2 } q \ tr$
and $tr: \text{set } tr \subseteq E \langle + \rangle IA \times UNIV \langle + \rangle IU \times UNIV$ **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 $S \text{ p } q$
and $\bigwedge p \ q \ e. \llbracket S \text{ p } q; e \in E \rrbracket \implies$
 $\text{weight-spmf } (\text{bind-spmf } p (\lambda s. \text{cpoke } \text{core1 } s \ e)) = \text{weight-spmf } (\text{bind-spmf } q (\lambda s. \text{cpoke } \text{core2 } s \ e))$
and $\bigwedge p \ q \ e. \llbracket S \text{ p } q; e \in E \rrbracket \implies$
 $S (\text{mk-lossless } (\text{bind-spmf } p (\lambda s. \text{cpoke } \text{core1 } s \ e))) (\text{mk-lossless } (\text{bind-spmf } q (\lambda s. \text{cpoke } \text{core2 } s \ e)))$
and $\bigwedge p \ q \ ia. \llbracket S \text{ p } q; ia \in IA \rrbracket \implies$
 $\text{bind-spmf } p (\lambda s1. \text{map-spmf fst } (\text{cfunc-adv } \text{core1 } s1 \ ia)) = \text{bind-spmf } q (\lambda s2. \text{map-spmf fst } (\text{cfunc-adv } \text{core2 } s2 \ ia))$

```

and  $\bigwedge p q ia oa. \llbracket S p q; ia \in IA \rrbracket$ 
 $\implies S (cond\text{-}spmf\text{-}fst (bind\text{-}spmf p (\lambda s1. cfunc\text{-}adv core1 s1 ia)) oa) (cond\text{-}spmf\text{-}fst$ 
 $(bind\text{-}spmf q (\lambda s2. cfunc\text{-}adv core2 s2 ia)) oa)$ 
and  $\bigwedge p q iu. \llbracket S p q; iu \in IU \rrbracket \implies$ 
 $bind\text{-}spmf p (\lambda s1. map\text{-}spmf\text{-}fst (cfunc\text{-}usr core1 s1 iu)) = bind\text{-}spmf q (\lambda s2.$ 
 $map\text{-}spmf\text{-}fst (cfunc\text{-}usr core2 s2 iu))$ 
and  $\bigwedge p q iu ou. \llbracket S p q; iu \in IU \rrbracket$ 
 $\implies S (cond\text{-}spmf\text{-}fst (bind\text{-}spmf p (\lambda s1. cfunc\text{-}usr core1 s1 iu)) ou) (cond\text{-}spmf\text{-}fst$ 
 $(bind\text{-}spmf q (\lambda s2. cfunc\text{-}usr core2 s2 iu)) ou)$ 
proof –
  show thesis
  by(rule that[where  $S = trace\text{-}core\text{-}closure core1 core2 E IA IU p q$ ])
  (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  $\times$  'oadv-rest  $\times$  'event list + 'iusr-rest  $\times$  'ousr-rest  $\times$  'event list) list
 $\implies$  ('iadv-rest  $\implies$  ('oadv-rest  $\times$  'event list) spmf)
 $\times$  ('iusr-rest  $\implies$  ('ousr-rest  $\times$  '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  $\implies$  ('event, 'iadv-rest, 'iusr-rest, 'oadv-rest,
'ousr-rest) trace-rest where
  trace-rest' S  $\llbracket$  =
    ( $\lambda ia. bind\text{-}spmf S (\lambda s. map\text{-}spmf\text{-}fst (rfunc\text{-}adv rest s ia)),$ 
 $\lambda iu. bind\text{-}spmf S (\lambda s. map\text{-}spmf\text{-}fst (rfunc\text{-}usr rest s iu))$ )
| trace-rest' S (obs # tr) = (case obs of
  Inl (ia, oa)  $\implies$  trace-rest' (cond-spmf-fst (bind-spmf S ( $\lambda s. rfunc\text{-}adv rest s ia$ ))
oa) tr
  | Inr (iu, ou)  $\implies$  trace-rest' (cond-spmf-fst (bind-spmf S ( $\lambda s. rfunc\text{-}usr rest s iu$ ))
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  $\implies$  rel-spmf S  $\implies$  >

```

$list-all2 (rel-sum (rel-prod IA (=)) (rel-prod IU (=))) == =>$
 $rel-prod (IA == => (=)) (IU == => (=))$
 $trace-rest' trace-rest'$
unfolding $trace-rest'$ -def by *transfer-prover*

definition $trace-rest-eq$

$:: ('s-rest, 'event, 'iadv-rest, 'iusr-rest, 'oadv-rest, 'ousr-rest, 'more1) rest-scheme$
 $\Rightarrow ('s-rest', 'event, 'iadv-rest, 'iusr-rest, 'oadv-rest, 'ousr-rest, 'more2) rest-scheme$
 $\Rightarrow 'iadv-rest set \Rightarrow 'iusr-rest set$
 $\Rightarrow 's-rest spmf \Rightarrow 's-rest' spmf \Rightarrow bool$ **where**
 $trace-rest-eq rest1 rest2 IA IU p q \longleftrightarrow$
 $(\forall tr. set tr \subseteq (IA \times UNIV) <+> (IU \times UNIV) \longrightarrow$
 $rel-prod (eq-onp (\lambda ia. ia \in IA) == => (=)) (eq-onp (\lambda iu. iu \in IU) == => (=))$
 $(trace-rest' rest1 p tr) (trace-rest' rest2 q tr))$

end

lemma $trace-rest-eqD$:

assumes $trace-rest-eq rest1 rest2 IA IU p q$
and $set tr \subseteq (IA \times UNIV) <+> (IU \times UNIV)$
shows $trace-rest-eqD-rfunc-adv$:
 $ia \in IA \implies fst (trace-rest' rest1 p tr) ia = fst (trace-rest' rest2 q tr) ia$
and $trace-rest-eqD-rfunc-usr$:
 $iu \in IU \implies snd (trace-rest' rest1 p tr) iu = snd (trace-rest' rest2 q tr) iu$
using *assms* by(*auto simp add: trace-rest-eq-def rel-fun-def rel-prod-sel eq-onp-def*)

lemma $trace-rest-eqI$:

assumes $\bigwedge tr ia. \llbracket set tr \subseteq (IA \times UNIV) <+> (IU \times UNIV); ia \in IA \rrbracket$
 $\implies fst (trace-rest' rest1 p tr) ia = fst (trace-rest' rest2 q tr) ia$
and $\bigwedge tr iu. \llbracket set tr \subseteq (IA \times UNIV) <+> (IU \times UNIV); iu \in IU \rrbracket$
 $\implies snd (trace-rest' rest1 p tr) iu = snd (trace-rest' rest2 q tr) iu$
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*]:

$trace-rest' rest (return-pmf None) tr = (\lambda-. return-pmf None, \lambda-. return-pmf None)$
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 $rel-rest' S (=) (eq-onp (\lambda ia. ia \in IA)) (eq-onp (\lambda iu. iu \in IU)) (=) (=) M rest rest'$

$rel-spmf S p q$

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**

$y=tr]$)
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* \Rightarrow 's-rest' *spmf* \Rightarrow *bool*
assumes *start*: *S* *p* *q*
and *step-rfunc-adv*: $\bigwedge p$ *q* *ia*. $\llbracket S$ *p* *q*; *ia* \in *IA* $\rrbracket \Longrightarrow$
 $bind\text{-}spmf$ *p* ($\lambda s1$. $map\text{-}spmf$ *fst* (*rfunc-adv* *rest1* *s1* *ia*)) = $bind\text{-}spmf$ *q* ($\lambda s2$.
 $map\text{-}spmf$ *fst* (*rfunc-adv* *rest2* *s2* *ia*))
and *sim-rfunc-adv*: $\bigwedge p$ *q* *ia* *s1* *s2* *s1'* *s2'* *oa*. $\llbracket S$ *p* *q*; *ia* \in *IA*;
s1 \in *set-spmf* *p*; *s2* \in *set-spmf* *q*; (*oa*, *s1'*) \in *set-spmf* (*rfunc-adv* *rest1* *s1* *ia*);
(*oa*, *s2'*) \in *set-spmf* (*rfunc-adv* *rest2* *s2* *ia*) \rrbracket
 $\Longrightarrow S$ (*cond-spmf-fst* ($bind\text{-}spmf$ *p* ($\lambda s1$. *rfunc-adv* *rest1* *s1* *ia*)) *oa*) (*cond-spmf-fst*
($bind\text{-}spmf$ *q* ($\lambda s2$. *rfunc-adv* *rest2* *s2* *ia*)) *oa*)
and *step-rfunc-usr*: $\bigwedge p$ *q* *iu*. $\llbracket S$ *p* *q*; *iu* \in *IU* $\rrbracket \Longrightarrow$
 $bind\text{-}spmf$ *p* ($\lambda s1$. $map\text{-}spmf$ *fst* (*rfunc-usr* *rest1* *s1* *iu*)) = $bind\text{-}spmf$ *q* ($\lambda s2$.
 $map\text{-}spmf$ *fst* (*rfunc-usr* *rest2* *s2* *iu*))
and *sim-rfunc-usr*: $\bigwedge p$ *q* *iu* *s1* *s2* *s1'* *s2'* *ou*. $\llbracket S$ *p* *q*; *iu* \in *IU*;
s1 \in *set-spmf* *p*; *s2* \in *set-spmf* *q*; (*ou*, *s1'*) \in *set-spmf* (*rfunc-usr* *rest1* *s1* *iu*);
(*ou*, *s2'*) \in *set-spmf* (*rfunc-usr* *rest2* *s2* *iu*) \rrbracket
 $\Longrightarrow S$ (*cond-spmf-fst* ($bind\text{-}spmf$ *p* ($\lambda s1$. *rfunc-usr* *rest1* *s1* *iu*)) *ou*) (*cond-spmf-fst*
($bind\text{-}spmf$ *q* ($\lambda s2$. *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 \times 'oadv-rest \times 'event list + 'iusr-rest \times 'ousr-rest \times 'event
list) list
assume *set* *tr* \subseteq *IA* \times *UNIV* $\langle + \rangle$ *IU* \times *UNIV*
then have ($\forall ia \in IA$. *fst* (*trace-rest'* *rest1* *p* *tr*) *ia* = *fst* (*trace-rest'* *rest2* *q* *tr*)
ia) \wedge
($\forall iu \in IU$. *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.prem*(1) **have** *tr*: *set* *tr* \subseteq *IA* \times *UNIV* $\langle + \rangle$ *IU* \times *UNIV* **by** *simp*
from *Cons.prem*(1)
consider (*rfunc-adv*) *ia* *oa* **where** *a* = *Inl* (*ia*, *oa*) *ia* \in *IA*
| (*rfunc-usr*) *iu* *ou* **where** *a* = *Inr* (*iu*, *ou*) *iu* \in *IU* **by** *auto*
then show ?*case*
proof *cases*
case *rfunc-adv*
let ?*p* = $bind\text{-}spmf$ *p* ($\lambda s1$. *rfunc-adv* *rest1* *s1* *ia*)

```

let ?q = bind-spmf q (λs2. rfunc-adv rest2 s2 ia)
show ?thesis
proof(cases oa ∈ fst ‘ set-spmf ?p)
  case True
  with step-rfunc-adv[OF Cons.premis(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.premis 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.premis(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 (λs1. rfunc-usr rest1 s1 iu)
  let ?q = bind-spmf q (λs2. rfunc-usr rest2 s2 iu)
  show ?thesis
  proof(cases ou ∈ fst ‘ set-spmf ?p)
    case True
    with step-rfunc-usr[OF Cons.premis(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.premis 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.premis(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 $ia \in IA \implies \text{fst} (\text{trace-rest}' \text{ rest1 } p \text{ tr}) \text{ ia} = \text{fst} (\text{trace-rest}' \text{ rest2 } q \text{ tr}) \text{ ia}$
and $iu \in IU \implies \text{snd} (\text{trace-rest}' \text{ rest1 } p \text{ tr}) \text{ iu} = \text{snd} (\text{trace-rest}' \text{ rest2 } q \text{ tr}) \text{ iu}$
for $ia \text{ iu}$ **by** *blast+*
qed

context

fixes $\text{rest} :: ('s\text{-rest}, 'event, 'iadv\text{-rest}, 'iusr\text{-rest}, 'oadv\text{-rest}, 'ousr\text{-rest}, 'more)$
rest-scheme

begin

fun *trace-rest-aux*

$:: 's\text{-rest} \text{ spmf} \Rightarrow ('iadv\text{-rest} \times 'oadv\text{-rest} \times 'event \text{ list} + 'iusr\text{-rest} \times 'ousr\text{-rest} \times 'event \text{ list}) \text{ list} \Rightarrow 's\text{-rest} \text{ spmf}$ **where**

$\text{trace-rest-aux } p \ [] = p$
 $| \text{trace-rest-aux } p \ (\text{Inl } (ia, oaes) \# \text{tr}) = \text{trace-rest-aux} (\text{cond-spmf-fst} (\text{bind-spmf } p \ (\lambda s. \text{rfunc-adv } \text{rest } s \text{ ia})) \text{ oaes}) \text{ tr}$
 $| \text{trace-rest-aux } p \ (\text{Inr } (iu, oues) \# \text{tr}) = \text{trace-rest-aux} (\text{cond-spmf-fst} (\text{bind-spmf } p \ (\lambda s. \text{rfunc-usr } \text{rest } s \text{ iu})) \text{ oues}) \text{ tr}$

end

lemma *trace-rest-conv-trace-rest-aux*:

$\text{trace-rest}' \text{ rest } p \ \text{tr} =$
 $(\lambda ia. \text{bind-spmf} (\text{trace-rest-aux } \text{rest } p \ \text{tr}) (\lambda s. \text{map-spmf } \text{fst} (\text{rfunc-adv } \text{rest } s \ \text{ia})),$
 $\lambda iu. \text{bind-spmf} (\text{trace-rest-aux } \text{rest } p \ \text{tr}) (\lambda s. \text{map-spmf } \text{fst} (\text{rfunc-usr } \text{rest } s \ \text{iu})))$
by (*induction p tr rule: trace-rest-aux.induct*) *simp-all*

lemma *trace-rest-aux-append*:

$\text{trace-rest-aux } \text{rest } p \ (\text{tr} \ @ \ \text{tr}') = \text{trace-rest-aux } \text{rest} \ (\text{trace-rest-aux } \text{rest } p \ \text{tr}) \ \text{tr}'$
by (*induction p tr rule: trace-rest-aux.induct*) *auto*

inductive *trace-rest-closure*

$:: ('s\text{-rest}, 'event, 'iadv\text{-rest}, 'iusr\text{-rest}, 'oadv\text{-rest}, 'ousr\text{-rest}, 'more) \text{ rest-scheme}$
 $\Rightarrow ('s\text{-rest}', 'event, 'iadv\text{-rest}, 'iusr\text{-rest}, 'oadv\text{-rest}, 'ousr\text{-rest}, 'more') \text{ rest-scheme}$
 $\Rightarrow 'iadv\text{-rest} \text{ set} \Rightarrow 'iusr\text{-rest} \text{ set}$

$\Rightarrow 's\text{-rest} \text{ spmf} \Rightarrow 's\text{-rest}' \text{ spmf} \Rightarrow 's\text{-rest} \text{ spmf} \Rightarrow 's\text{-rest}' \text{ spmf} \Rightarrow \text{bool}$

for $\text{rest1 } \text{rest2 } IA \ IU \ p \ q$ **where**

$\text{trace-rest-closure } \text{rest1 } \text{rest2 } IA \ IU \ p \ q \ (\text{trace-rest-aux } \text{rest1 } p \ \text{tr}) \ (\text{trace-rest-aux } \text{rest2 } q \ \text{tr})$

if $\text{set } \text{tr} \subseteq IA \times UNIV \lt+> IU \times UNIV$

lemma *trace-rest-closure-start*: $\text{trace-rest-closure } \text{rest1 } \text{rest2 } IA \ IU \ p \ q \ p \ q$

by (*simp add: trace-rest-closure.simps exI[where x=[]]*)

lemma *trace-rest-closure-step*:

assumes $\text{trace-rest-eq } \text{rest1 } \text{rest2 } IA \ IU \ p \ q$

and $\text{trace-rest-closure } \text{rest1 } \text{rest2 } IA \ IU \ p \ q \ p' \ q'$

shows $\text{trace-rest-closure-step-rfunc-adv}$:

$ia \in IA \implies \text{bind-spmf } p' (\lambda s1. \text{map-spmf fst } (\text{rfunc-adv } \text{rest1 } s1 \text{ ia})) = \text{bind-spmf } q' (\lambda s2. \text{map-spmf fst } (\text{rfunc-adv } \text{rest2 } s2 \text{ ia}))$
 (is *PROP* ?thesis1)
and *trace-rest-closure-step-rfunc-usr*:
 $iu \in IU \implies \text{bind-spmf } p' (\lambda s1. \text{map-spmf fst } (\text{rfunc-usr } \text{rest1 } s1 \text{ iu})) = \text{bind-spmf } q' (\lambda s2. \text{map-spmf fst } (\text{rfunc-usr } \text{rest2 } s2 \text{ iu}))$
 (is *PROP* ?thesis2)

proof –

from *assms*(2) **obtain** *tr* **where** $p: p' = \text{trace-rest-aux } \text{rest1 } p \text{ tr}$
and $q: q' = \text{trace-rest-aux } \text{rest2 } q \text{ tr}$
and $tr: \text{set } tr \subseteq IA \times UNIV \langle + \rangle IU \times UNIV$ **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 $S \equiv \text{trace-rest-closure } \text{rest1 } \text{rest2 } IA \text{ IU } p \text{ q}$
assumes $S \text{ } p' \text{ } q'$

shows *trace-rest-closure-sim-rfunc-adv*: $ia \in IA$
 $\implies S (\text{cond-spmf-fst } (\text{bind-spmf } p' (\lambda s1. \text{rfunc-adv } \text{rest1 } s1 \text{ ia})) \text{ oa})$
 $(\text{cond-spmf-fst } (\text{bind-spmf } q' (\lambda s2. \text{rfunc-adv } \text{rest2 } s2 \text{ ia})) \text{ oa})$
 (is *PROP* ?thesis1)
and *trace-rest-closure-sim-rfunc-usr*: $iu \in IU$
 $\implies S (\text{cond-spmf-fst } (\text{bind-spmf } p' (\lambda s1. \text{rfunc-usr } \text{rest1 } s1 \text{ iu})) \text{ ou})$
 $(\text{cond-spmf-fst } (\text{bind-spmf } q' (\lambda s2. \text{rfunc-usr } \text{rest2 } s2 \text{ iu})) \text{ ou})$
 (is *PROP* ?thesis2)

proof –

from *assms*(2) **obtain** *tr* **where** $p: p' = \text{trace-rest-aux } \text{rest1 } p \text{ tr}$
and $q: q' = \text{trace-rest-aux } \text{rest2 } q \text{ tr}$
and $tr: \text{set } tr \subseteq IA \times UNIV \langle + \rangle IU \times UNIV$ **unfolding** *S-def* **by** *cases*
show *PROP* ?thesis1 **using** $p \text{ } q \text{ } tr$
by(*auto simp add: S-def trace-rest-closure.simps trace-rest-aux-append intro!*:
exI[**where** $x=tr$ @ [*Inl* (-, -)]])
show *PROP* ?thesis2 **using** $p \text{ } q \text{ } tr$
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 $S \text{ } p \text{ } q$
and $\bigwedge p \text{ } q \text{ } ia. \llbracket S \text{ } p \text{ } q; ia \in IA \rrbracket \implies$
 $\text{bind-spmf } p (\lambda s1. \text{map-spmf fst } (\text{rfunc-adv } \text{rest1 } s1 \text{ ia})) = \text{bind-spmf } q (\lambda s2.$
 $\text{map-spmf fst } (\text{rfunc-adv } \text{rest2 } s2 \text{ ia}))$
and $\bigwedge p \text{ } q \text{ } ia \text{ } oa. \llbracket S \text{ } p \text{ } q; ia \in IA \rrbracket$
 $\implies S (\text{cond-spmf-fst } (\text{bind-spmf } p (\lambda s1. \text{rfunc-adv } \text{rest1 } s1 \text{ ia})) \text{ oa}) (\text{cond-spmf-fst}$

$(\text{bind-spmf } q (\lambda s2. \text{rfunc-adv } \text{rest2 } s2 \text{ ia})) \text{ oa}$
and $\bigwedge p \ q \ \text{iu}. \llbracket S \ p \ q; \ \text{iu} \in \text{IU} \rrbracket \implies$
 $\text{bind-spmf } p (\lambda s1. \text{map-spmf } \text{fst} (\text{rfunc-usr } \text{rest1 } s1 \ \text{iu})) = \text{bind-spmf } q (\lambda s2. \text{map-spmf } \text{fst} (\text{rfunc-usr } \text{rest2 } s2 \ \text{iu}))$
and $\bigwedge p \ q \ \text{iu} \ \text{ou}. \llbracket S \ p \ q; \ \text{iu} \in \text{IU} \rrbracket$
 $\implies S (\text{cond-spmf-fst} (\text{bind-spmf } p (\lambda s1. \text{rfunc-usr } \text{rest1 } s1 \ \text{iu})) \ \text{ou}) (\text{cond-spmf-fst} (\text{bind-spmf } q (\lambda s2. \text{rfunc-usr } \text{rest2 } s2 \ \text{iu})) \ \text{ou})$
proof –
show *thesis*
by(*rule that*[**where** $S = \text{trace-rest-closure } \text{rest1 } \text{rest2 } \text{IA } \text{IU } p \ q$]
(auto intro: trace-rest-closure-start trace-rest-closure-step[*OF assms*] *trace-rest-closure-sim*
))
qed

definition *callee-of-core*

$:: ('s\text{-core}, 'event, 'iadv\text{-core}, 'iusr\text{-core}, 'oadv\text{-core}, 'ousr\text{-core}) \text{ core}$
 $\implies ('s\text{-core}, 'event + 'iadv\text{-core} + 'iusr\text{-core}, \text{unit} + 'oadv\text{-core} + 'ousr\text{-core})$
oracle' **where**
 $\text{callee-of-core } \text{core} =$
 $\text{map-fun } \text{id} (\text{map-fun } \text{id} (\text{map-spmf} (\text{Pair } ()))) (\text{cpoke } \text{core}) \oplus_{\text{O}} \text{cfunc-adv } \text{core}$
 $\oplus_{\text{O}} \text{cfunc-usr } \text{core}$

lemma *callee-of-core-simps* [*simp*]:

$\text{callee-of-core } \text{core } s (\text{Inl } e) = \text{map-spmf} (\text{Pair } (\text{Inl } ())) (\text{cpoke } \text{core } s \ e)$
 $\text{callee-of-core } \text{core } s (\text{Inr } (\text{Inl } iadv\text{-core})) = \text{map-spmf} (\text{apfst } (\text{Inr} \circ \text{Inl})) (\text{cfunc-adv } \text{core } s \ iadv\text{-core})$
 $\text{callee-of-core } \text{core } s (\text{Inr } (\text{Inr } iusr\text{-core})) = \text{map-spmf} (\text{apfst } (\text{Inr} \circ \text{Inr})) (\text{cfunc-usr } \text{core } s \ iusr\text{-core})$
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 $WT: \text{WT-core } \mathcal{I}\text{-adv } \mathcal{I}\text{-usr } I \ \text{core}$
and $I: I \ s$
shows $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv} \oplus_{\mathcal{I}} \mathcal{I}\text{-usr}) \vdash_c \text{callee-of-core } \text{core } s \ \checkmark$
apply(*rule WT-calleeI*)
subgoal for $x \ y \ s'$ **using** $I \ WT\text{-coreD}$ [*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 $WT: \text{WT-core } \mathcal{I}\text{-adv } \mathcal{I}\text{-usr } I \ \text{core}$
shows *callee-invariant-on* (*callee-of-core* core) I ($\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv} \oplus_{\mathcal{I}} \mathcal{I}\text{-usr})$)
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*

$:: ('s\text{-rest}, 'event, 'iadv\text{-rest}, 'iusr\text{-rest}, 'oadv\text{-rest}, 'ousr\text{-rest}, 'more) \text{ rest-scheme}$
 $\Rightarrow ('s\text{-rest}, 'iadv\text{-rest} + 'iusr\text{-rest}, 'oadv\text{-rest} \times 'event \text{ list} + 'ousr\text{-rest} \times 'event$
list) oracle' **where**
 $callee\text{-of-rest rest} = rfunc\text{-adv rest} \oplus_O rfunc\text{-usr rest}$

lemma *callee-of-rest-simps* [*simp*]:

$callee\text{-of-rest rest s (Inl iadv\text{-rest})} = map\text{-spmf (apfst Inl) (rfunc\text{-adv rest s}$
iadv-rest)
 $callee\text{-of-rest rest s (Inr iusr\text{-rest})} = map\text{-spmf (apfst Inr) (rfunc\text{-usr rest s}$
iusr-rest)
by(*simp-all add: callee-of-rest-def*)

lemma *WT-callee-of-rest* [*WT-intro*]:

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

fun *fuse-callee*

$:: ('iadv\text{-core} + 'iadv\text{-rest}) + ('iusr\text{-core} + 'iusr\text{-rest}) \Rightarrow$
 $(('oadv\text{-core} + 'oadv\text{-rest}) + ('ousr\text{-core} + 'ousr\text{-rest}),$
 $('event + 'iadv\text{-core} + 'iusr\text{-core}) + ('iadv\text{-rest} + 'iusr\text{-rest}),$
 $(unit + 'oadv\text{-core} + 'ousr\text{-core}) + ('oadv\text{-rest} \times 'event \text{ list} + 'ousr\text{-rest} \times$
 $'event \text{ list})) \text{ gpv}$
where
 $fuse\text{-callee (Inl (Inl iadv\text{-core}))} = \text{Pause (Inl (Inr (Inl iadv\text{-core}))) } (\lambda x. \text{ case } x \text{ of}$
 $\text{Inl (Inr (Inl oadv\text{-core}))} \Rightarrow \text{Done (Inl (Inl oadv\text{-core}))}$
 $| - \Rightarrow \text{Fail})$
 $| fuse\text{-callee (Inl (Inr iadv\text{-rest}))} = \text{Pause (Inr (Inl iadv\text{-rest})) } (\lambda x. \text{ case } x \text{ of}$
 $\text{Inr (Inl (oadv\text{-rest}, es))} \Rightarrow \text{bind-gpv (pauses (map (Inl } \circ \text{ Inl) es)) } (\lambda -. \text{ Done}$
 $(\text{Inl (Inr oadv\text{-rest})))}$
 $| - \Rightarrow \text{Fail})$
 $| fuse\text{-callee (Inr (Inl iusr\text{-core}))} = \text{Pause (Inl (Inr (Inr iusr\text{-core}))) } (\lambda x. \text{ case } x \text{ of}$
 $\text{Inl (Inr (Inr oadv\text{-core}))} \Rightarrow \text{Done (Inr (Inl oadv\text{-core}))}$
 $| fuse\text{-callee (Inr (Inr iusr\text{-rest}))} = \text{Pause (Inr (Inr iusr\text{-rest})) } (\lambda x. \text{ case } x \text{ of}$
 $\text{Inr (Inr (ousr\text{-rest}, es))} \Rightarrow \text{bind-gpv (pauses (map (Inl } \circ \text{ Inl) es)) } (\lambda -. \text{ Done}$
 $(\text{Inr (Inr ousr\text{-rest})))}$

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

definition *fuse-converter*

$:: (('iadv\text{-core} + 'iadv\text{-rest}) + ('iusr\text{-core} + 'iusr\text{-rest}),$
 $('oadv\text{-core} + 'oadv\text{-rest}) + ('ousr\text{-core} + 'ousr\text{-rest}),$

('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 (λs-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: $\bigwedge p q e. S\text{-core } p q \implies e \in E$

$\implies \text{weight-spmf } (\text{bind-spmf } p (\lambda s. \text{cpoke core1 } s e)) = \text{weight-spmf } (\text{bind-spmf } p$
 $q (\lambda s. \text{cpoke core2 } s e))$

and sim-cpoke: $\bigwedge p q e. S\text{-core } p q \implies e \in E$

$\implies S\text{-core } (\text{mk-lossless } (\text{bind-spmf } p (\lambda s. \text{cpoke core1 } s e))) (\text{mk-lossless}$
 $(\text{bind-spmf } q (\lambda s. \text{cpoke core2 } s e)))$

and step-cfunc-adv: $\bigwedge p q ia. \llbracket S\text{-core } p q; ia \in IA \rrbracket$

$\implies \text{bind-spmf } p (\lambda s1. \text{map-spmf fst } (\text{cfunc-adv core1 } s1 ia)) = \text{bind-spmf } q$
 $(\lambda s2. \text{map-spmf fst } (\text{cfunc-adv core2 } s2 ia))$

and sim-cfunc-adv: $\bigwedge p q ia oa. \llbracket S\text{-core } p q; ia \in IA \rrbracket \implies$

S-core (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: $\bigwedge p q iu. \llbracket S\text{-core } p q; iu \in IU \rrbracket$

```

    ⇒ 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 ou. [ S-core p q; iu ∈ IU ] ⇒
    S-core (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)
  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
  end

```

```

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*: $\bigwedge p q ia. \llbracket S\text{-rest } p q; ia \in IA \rrbracket$

$\implies \text{bind-spmf } p (\lambda s1. \text{map-spmf } \text{fst} (\text{rfunc-adv } \text{rest1 } s1 ia)) = \text{bind-spmf } q$
 $(\lambda s2. \text{map-spmf } \text{fst} (\text{rfunc-adv } \text{rest2 } s2 ia))$

and *sim-rfunc-adv*: $\bigwedge p q ia oa. \llbracket S\text{-rest } p q; ia \in IA \rrbracket \implies$

S-rest (*cond-spmf-fst* (*bind-spmf* *p* ($\lambda s1. \text{rfunc-adv } \text{rest1 } s1 ia$)) *oa*)
(*cond-spmf-fst* (*bind-spmf* *q* ($\lambda s2. \text{rfunc-adv } \text{rest2 } s2 ia$)) *oa*)

and *step-rfunc-usr*: $\bigwedge p q iu. \llbracket S\text{-rest } p q; iu \in IU \rrbracket$

$\implies \text{bind-spmf } p (\lambda s1. \text{map-spmf } \text{fst} (\text{rfunc-usr } \text{rest1 } s1 iu)) = \text{bind-spmf } q$
 $(\lambda s2. \text{map-spmf } \text{fst} (\text{rfunc-usr } \text{rest2 } s2 iu))$

and *sim-rfunc-usr*: $\bigwedge p q iu ou. \llbracket S\text{-rest } p q; iu \in IU \rrbracket \implies$

S-rest (*cond-spmf-fst* (*bind-spmf* *p* ($\lambda s1. \text{rfunc-usr } \text{rest1 } s1 iu$)) *ou*)
(*cond-spmf-fst* (*bind-spmf* *q* ($\lambda s2. \text{rfunc-usr } \text{rest2 } s2 iu$)) *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 ≫≡ (λx. (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 ≫≡ (λx. (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*]:

$(\mathcal{IAC} \oplus_{\mathcal{I}} \text{map-}\mathcal{I} \text{ id fst } \mathcal{IAR}) \oplus_{\mathcal{I}} (\mathcal{IUC} \oplus_{\mathcal{I}} \text{map-}\mathcal{I} \text{ id fst } \mathcal{IUR}), (\mathcal{IE} \oplus_{\mathcal{I}} (\mathcal{IAC} \oplus_{\mathcal{I}} \mathcal{IUC})) \oplus_{\mathcal{I}} (\mathcal{IAR} \oplus_{\mathcal{I}} \mathcal{IUR}) \vdash_C \text{fuse-converter } \checkmark$
if $\forall x. \forall (y, es) \in \text{responses-}\mathcal{I} \mathcal{IAR} x. \text{set } es \subseteq \text{outs-}\mathcal{I} \mathcal{IE} \forall x. \forall (y, es) \in \text{responses-}\mathcal{I} \mathcal{IUR} x. \text{set } es \subseteq \text{outs-}\mathcal{I} \mathcal{IE}$
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-}\mathcal{I} \mathcal{IE}) (*outs-}\mathcal{I} \mathcal{ICA}) (*outs-}\mathcal{I} \mathcal{ICU})
(return-spmf s-core) (*return-spmf s-core'*)
and *rest*: *trace-rest-eq* *rest1* *rest2* (*outs-}\mathcal{I} \mathcal{IRA}) (*outs-}\mathcal{I} \mathcal{IRU}) (*return-spmf*
s-rest) (*return-spmf s-rest'*)
and *IC1*: *callee-invariant-on* (*callee-of-core* *core1*) *IC1* ($\mathcal{IE} \oplus_{\mathcal{I}} (\mathcal{ICA} \oplus_{\mathcal{I}} \mathcal{ICU})$)
IC1 s-core
and *IC2*: *callee-invariant-on* (*callee-of-core* *core2*) *IC2* ($\mathcal{IE} \oplus_{\mathcal{I}} (\mathcal{ICA} \oplus_{\mathcal{I}} \mathcal{ICU})$)
IC2 s-core'
and *IR1*: *callee-invariant-on* (*callee-of-rest* *rest1*) *IR1* ($\mathcal{IRA} \oplus_{\mathcal{I}} \mathcal{IRU}$) *IR1*
s-rest
and *IR2*: *callee-invariant-on* (*callee-of-rest* *rest2*) *IR2* ($\mathcal{IRA} \oplus_{\mathcal{I}} \mathcal{IRU}$) *IR2*
s-rest'
and *E1* [*WT-intro*]: $\forall x. \forall (y, es) \in \text{responses-}\mathcal{I} \mathcal{IRA} x. \text{set } es \subseteq \text{outs-}\mathcal{I} \mathcal{IE}$
and *E2* [*WT-intro*]: $\forall x. \forall (y, es) \in \text{responses-}\mathcal{I} \mathcal{IRU} x. \text{set } es \subseteq \text{outs-}\mathcal{I} \mathcal{IE}$
shows *trace-callee-eq* (*fused-resource.fuse* *core1* *rest1*) (*fused-resource.fuse* *core2*
rest2)
 $((\text{outs-}\mathcal{I} \mathcal{ICA} \langle + \rangle \text{outs-}\mathcal{I} \mathcal{IRA}) \langle + \rangle (\text{outs-}\mathcal{I} \mathcal{ICU} \langle + \rangle \text{outs-}\mathcal{I} \mathcal{IRU}))$
*(return-spmf (s-core, s-rest)) (return-spmf (s-core', s-rest'))******

proof –

let $?IC = \mathcal{IE} \oplus_{\mathcal{I}} (\mathcal{ICA} \oplus_{\mathcal{I}} \mathcal{ICU})$
let $?IR = \mathcal{IRA} \oplus_{\mathcal{I}} \mathcal{IRU}$
let $?I' = ?IC \oplus_{\mathcal{I}} ?IR$
let $?I = (\mathcal{ICA} \oplus_{\mathcal{I}} \text{map-}\mathcal{I} \text{ id fst } \mathcal{IRA}) \oplus_{\mathcal{I}} (\mathcal{ICU} \oplus_{\mathcal{I}} \text{map-}\mathcal{I} \text{ id fst } \mathcal{IRU})$

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 $IC1$? IC **by fact**
interpret $IC2$: callee-invariant-on callee-of-core core2 $IC2$? IC **by fact**
interpret $IR1$: callee-invariant-on callee-of-rest rest1 $IR1$? IR **by fact**
interpret $IR2$: callee-invariant-on callee-of-rest rest2 $IR2$? IR **by fact**

from core **have** outs- \mathcal{I} ? $IC \vdash_C$ 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} ? $IC \vdash_R$ RES (callee-of-core core1) s -core \approx RES (callee-of-core core2) s -core' **by simp**
moreover have outs- \mathcal{I} ? $IR \vdash_C$ 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} ? $IR \vdash_R$ 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 \parallel RES (callee-of-rest rest1) s -rest \approx
RES (callee-of-core core2) s -core' \parallel RES (callee-of-rest rest2) s -rest'
by(simp add: trace-eq'-parallel-resource)
hence outs- \mathcal{I} ? $\mathcal{I} \vdash_R$ fuse-converter \triangleright (RES (callee-of-core core1) s -core \parallel RES (callee-of-rest rest1) s -rest) \approx
fuse-converter \triangleright (RES (callee-of-core core2) s -core' \parallel 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 \Rightarrow 's2 spmf \Rightarrow bool) \Rightarrow 's1 spmf \Rightarrow 's2 spmf \Rightarrow 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 $\bigwedge x :: \text{nat. } x \in \text{set-spmf } p \implies S$ (f x) (g x)

lemma trace-eq-simcl-bindI [intro?]: trace-eq-simcl S (bind-spmf p f) (bind-spmf p g)

if $\bigwedge x. x \in \text{set-spmf } p \implies S$ (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 *: $\bigwedge x :: 'a. x \in \text{set-spmf } p \implies \text{trace-eq-simcl } S$ (f x) (g x)

proof –

obtain $P :: 'a \Rightarrow \text{nat spmf}$ **and** F G **where**

```

** :  $\bigwedge x. x \in \text{set-spmf } p \implies f x = \text{bind-spmf } (P x) (F x) \wedge g x = \text{bind-spmf } (P x) (G x) \wedge (\forall y \in \text{set-spmf } (P x). S (F x y) (G x y))$ 
  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 ( $\lambda x. \text{map-spmf } (Pair x) (P x)$ )) ( $\lambda(x, y). F x y$ )
  by(simp add: bind-map-spmf o-def ** cong: bind-spmf-cong)
moreover have bind-spmf p g = bind-spmf (bind-spmf p ( $\lambda x. \text{map-spmf } (Pair x) (P x)$ )) ( $\lambda(x, y). G x y$ )
  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  $\forall x \in \text{set-spmf } p. \text{trace-eq-simcl } S (f x) q$ 
proof -
  have trace-eq-simcl S (bind-spmf p f) (bind-spmf p ( $\lambda-. q$ ))
    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  $\forall x \in \text{set-spmf } p. \text{trace-eq-simcl } S (f x) q \text{ lossless-spmf } p$ 
  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  $\forall x \in \text{set-spmf } q. \text{trace-eq-simcl } S p (f x)$ 
proof -
  have trace-eq-simcl S (bind-spmf q ( $\lambda-. p$ )) (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  $\forall x \in \text{set-spmf } q. \text{trace-eq-simcl } S p (f x) \text{ lossless-spmf } q$ 
  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  $\Rightarrow$  's2 spmf  $\Rightarrow$  bool
proof -
  have trace-eq-simcl S (bind-spmf (return-pmf None) (undefined :: nat  $\Rightarrow$  's1 spmf)) (bind-spmf (return-pmf None) (undefined :: nat  $\Rightarrow$  '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 $\forall x \in \text{set-spmf } p. S$ (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 $\forall x \in \text{set-spmf } p. \text{trace-eq-simcl } S$ (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 $\forall x \in \text{set-spmf } q. \text{trace-eq-simcl } S$ 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)
 $\longleftrightarrow S$ (return-spmf x) (return-spmf y)
apply(rule iffI)
subgoal by(erule trace-eq-simcl.cases; clarsimp dest!: sym[**where** $s = \text{return-spmf } -$])
(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: $\bigwedge p q a. \llbracket S p q; a \in A \rrbracket \implies$
 $\text{bind-spmf } p (\lambda s. \text{map-spmf } \text{fst} (\text{callee1 } s a)) = \text{bind-spmf } q (\lambda s. \text{map-spmf } \text{fst} (\text{callee2 } s a))$
and sim: $\bigwedge p q a \text{ res } b s'. \llbracket S p q; a \in A; \text{res} \in \text{set-spmf } q; (b, s') \in \text{set-spmf} (\text{callee2 } \text{res } a) \rrbracket$
 $\implies \text{trace-eq-simcl } S (\text{cond-spmf-fst} (\text{bind-spmf } p (\lambda s. \text{callee1 } s a)) b)$
 $(\text{cond-spmf-fst} (\text{bind-spmf } q (\lambda s. \text{callee2 } s a)) b)$
and start: trace-eq-simcl S p q **and** $a: a \in A$
shows trace-eq-simcl-callee-step: $\text{bind-spmf } p (\lambda s. \text{map-spmf } \text{fst} (\text{callee1 } s a)) =$
 $\text{bind-spmf } q (\lambda s. \text{map-spmf } \text{fst} (\text{callee2 } s a))$ (**is** ?step)
and trace-eq-simcl-callee-sim: $\bigwedge \text{res } b s'. \llbracket \text{res} \in \text{set-spmf } q; (b, s') \in \text{set-spmf} (\text{callee2 } \text{res } a) \rrbracket$
 $\implies \text{trace-eq-simcl } S (\text{cond-spmf-fst} (\text{bind-spmf } p (\lambda s. \text{callee1 } s a)) b)$
 $(\text{cond-spmf-fst} (\text{bind-spmf } q (\lambda s. \text{callee2 } s a)) b)$ (**is** $\bigwedge \text{res } b$
 $s'. \llbracket ?\text{res } \text{res}; ?b \text{ res } b s' \rrbracket \implies ?\text{sim } \text{res } b s'$)

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 $?Y = \text{cond-bind-spmf-fst } X (\lambda y. \text{map-spmf fst } (\text{bind-spmf } (f y) (\lambda s. \text{callee1 } s a))) b$
let $?Y' = \text{cond-bind-spmf-fst } X (\lambda y. \text{map-spmf fst } (\text{bind-spmf } (g y) (\lambda s. \text{callee2 } s a))) b$
have $\text{cond-spmf-fst } (\text{bind-spmf } p (\lambda s. \text{callee1 } s a)) b = \text{bind-spmf } ?Y (\lambda x. \text{cond-spmf-fst } (\text{bind-spmf } (f x) (\lambda s. \text{callee1 } s a)) b)$
unfolding *bind-nat* **by**(*simp add: cond-spmf-fst-bind o-def*)
moreover have $\text{cond-spmf-fst } (\text{bind-spmf } q (\lambda s. \text{callee2 } s a)) b = \text{bind-spmf } ?Y' (\lambda x. \text{cond-spmf-fst } (\text{bind-spmf } (g x) (\lambda s. \text{callee2 } s a)) b)$
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* $S (\text{cond-spmf-fst } (\text{bind-spmf } p (\lambda s. \text{callee1 } s a)) b)$
 $(\text{cond-spmf-fst } (\text{bind-spmf } q (\lambda s. \text{callee2 } s a)) b)$ **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 $\text{callee1} :: ('a, 'b, 's1) \text{ callee}$ **and** $\text{callee2} :: ('a, 'b, 's2) \text{ callee}$
assumes *start*: $S p q$
and *step*: $\bigwedge p q a. \llbracket S p q; a \in A \rrbracket \implies$
 $\text{bind-spmf } p (\lambda s. \text{map-spmf fst } (\text{callee1 } s a)) = \text{bind-spmf } q (\lambda s. \text{map-spmf fst } (\text{callee2 } s a))$
and *sim*: $\bigwedge p q a \text{ res } b s'. \llbracket S p q; a \in A; \text{res} \in \text{set-spmf } q; (b, s') \in \text{set-spmf } (\text{callee2 } \text{res } a) \rrbracket$
 $\implies \text{trace-eq-simcl } S (\text{cond-spmf-fst } (\text{bind-spmf } p (\lambda s. \text{callee1 } s a)) b)$
 $(\text{cond-spmf-fst } (\text{bind-spmf } q (\lambda s. \text{callee2 } s a)) b)$
shows *trace-callee-eq* $\text{callee1 } \text{callee2 } A p q$
proof –
let $?S = \text{trace-eq-simcl } S$
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 $\text{core1} :: ('s\text{-core}, 'event, 'iadv\text{-core}, 'iusr\text{-core}, 'oadv\text{-core}, 'ousr\text{-core}) \text{ core}$
and $\text{core2} :: ('s\text{-core}', 'event, 'iadv\text{-core}', 'iusr\text{-core}', 'oadv\text{-core}', 'ousr\text{-core}') \text{ core}$
and $S :: 's\text{-core} \text{ spmf} \Rightarrow 's\text{-core}' \text{ spmf} \Rightarrow \text{bool}$
assumes *start*: $S p q$
and *step-cpoke*: $\bigwedge p q e. \llbracket S p q; e \in E \rrbracket \implies$
 $\text{weight-spmf } (\text{bind-spmf } p (\lambda s. \text{cpoke } \text{core1 } s e)) = \text{weight-spmf } (\text{bind-spmf } q (\lambda s. \text{cpoke } \text{core2 } s e))$
and *sim-cpoke*: $\bigwedge p q e. \llbracket S p q; e \in E \rrbracket \implies$
 $\text{trace-eq-simcl } S (\text{mk-lossless } (\text{bind-spmf } p (\lambda s. \text{cpoke } \text{core1 } s e))) (\text{mk-lossless } (\text{bind-spmf } q (\lambda s. \text{cpoke } \text{core2 } s e)))$

```

(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) ]
    ⇒ 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: ∧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) ]
    ⇒ 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
spm-f-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 ( $\lambda y. \text{map-spmf fst } (f y \gg (\lambda s1. \text{cfunc-adv}$ 
core1 s1 ia))) oa =
    cond-bind-spmf-fst X ( $\lambda y. \text{map-spmf fst } (g y \gg (\lambda s2. \text{cfunc-adv}$ 
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 ( $\lambda y. \text{map-spmf fst } (f y \gg (\lambda s1. \text{cfunc-usr}$ 
core1 s1 iu))) ou =
    cond-bind-spmf-fst X ( $\lambda y. \text{map-spmf fst } (g y \gg (\lambda s2. \text{cfunc-usr}$ 
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 (λs-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 (λs-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 (λs-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-oracle-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

$('a, 'b) \text{ state-iso} = ('a \Rightarrow 'b) \times ('b \Rightarrow 'a)$

definition

$\text{state-iso} :: ('a, 'b) \text{ state-iso} \Rightarrow \text{bool}$

where

$\text{state-iso} \equiv (\lambda(f, g). \text{type-definition } f \text{ } g \text{ } UNIV)$

definition

$\text{apply-state-iso} :: ('s1, 's2) \text{ state-iso} \Rightarrow ('s1, 'i, 'o) \text{ oracle}' \Rightarrow ('s2, 'i, 'o) \text{ oracle}'$

where

$\text{apply-state-iso} \equiv (\lambda(f, g). \text{map-fun } g \text{ (map-fun id (map-spmf (map-prod id f)))))$

lemma *apply-state-iso-id*: $\text{apply-state-iso } (id, id) = id$

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

lemma *apply-state-iso-compose*: $\text{apply-state-iso } si1 \text{ (apply-state-iso } si2 \text{ oracle)} =$

$\text{apply-state-iso } (\text{map-prod } (\lambda f. f \circ (\text{fst } si2)) \text{ ((o) (snd } si2)) \text{ } si1) \text{ oracle}$

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-refl*)

qed

6.1 Parallel State Isomorphism

definition

parallel-state-iso :: $((s\text{-core1} \times s\text{-core2}) \times (s\text{-rest1} \times s\text{-rest2}), (s\text{-core1} \times s\text{-rest1}) \times (s\text{-core2} \times s\text{-rest2}))$ *state-iso*

where

parallel-state-iso =

$(\lambda((s11, s12), (s21, s22)). ((s11, s21), (s12, s22)), \lambda((s11, s21), (s12, s22)). ((s11, s12), (s21, s22)))$

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 =

$(\lambda(((s11, s12), s13), (s21, s22), s23). (((s11, s21), s12, s22), s13, s23), \lambda(((s11, s21), s12, s22), s13, s23). (((s11, s12), s13), (s21, s22), s23)))$

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 Assoc-Swap State Isomorphism

definition

```

iso-swapar
where
  iso-swapar = ( $\lambda((sm, s1), s2). (s1, sm, s2), \lambda(s1, sm, s2). ((sm, s1), s2))$ )

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  $\Rightarrow$  'b)  $\Rightarrow$  ('i-cnv, 'o-cnv, 'i-res, 'o-res) converter  $\Rightarrow$ 
    ('a + 'i-cnv, 'b + 'o-cnv, 'i-res, 'o-res) converter
  where
    run-converter (ldummy-converter f conv) = ( $\lambda inp. case\ inp\ of$ 
      Inl x  $\Rightarrow map\ gpv\ (map\ prod\ Inl\ (\lambda -. ldummy\ converter\ f\ conv))\ id\ (Done\ (f\ x,$ 
    ()))
      | Inr x  $\Rightarrow map\ gpv\ (map\ prod\ Inr\ (\lambda c. ldummy\ converter\ f\ c))\ id\ (run\ converter$ 
    conv x))

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

lemma ldummy-converter-of-callee:
  ldummy-converter f (converter-of-callee callee state) =
    converter-of-callee ( $\lambda s\ q. case\ sum\ (\lambda ql. Done\ (Inl\ (f\ ql),\ s))\ (\lambda qr. map\ gpv$ 
  (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 ( $\lambda s q. \text{case-sum } (\lambda ql. \text{map-gpv } (\text{map-prod } \text{Inl } \text{id}) \text{id } (\text{callee } s
  ql)) (\lambda qr. \text{Done } (\text{Inr } (f \text{ qr}), s)) \text{ } q) \text{ state}$ 
  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  $\equiv$  (cnv1  $|_{=}$  cnv2)  $\odot$  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  $\equiv$  (parallel-wiring  $|_{=}$  parallel-wiring)  $\odot$  parallel-wiring

```

definition*fused-wiring_w***where** $fused-wiring_w \equiv (parallel-wiring_w \mid_w parallel-wiring_w) \circ_w parallel-wiring_w$ **schematic-goal***wiring-fused-wiring*[*wiring-intro*]: *wiring* ?*I*1 ?*I*2 *fused-wiring* *fused-wiring_w***unfolding** *fused-wiring-def* *fused-wiring_w-def***by**(*rule wiring-intro*)**+****schematic-goal** *WT-fused-wiring* [*WT-intro*]: ?*I*1, ?*I*2 \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** $attach-c1f22-c1f22 = (((1_C \mid= 1_C) \mid= ((wiring-c1r22-c1r22 \text{ cnv1 } \text{ cnv2}) \mid= 1_C))$ $\odot fused-wiring) \triangleright (res1 \parallel res2)$ **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-wiringw (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-wiringw-def apply-state-iso-def parallel-state-iso-def
parallel-wiringw-def)
  apply(simp add: apply-wiring-def comp-wiring-def parallel2-wiring-def lassocw-def
swap-lassocw-def rassocw-def swapw-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-oracle-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-oracle-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 ‡I cnv-adv-rest) ‡I cnv-usr-core ‡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 =  $\lambda((s11, s12), s13), (s21, s22), s23). (((s11, s21), s12, s22), s13, s23)$ 
  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 ( $\lambda s1'. (s1', suc, sur)$ )) id (left-gpv (map-gpv
(map-prod Inl ( $\lambda s1'. (s1', sar)$ )) id (cnv-adv-core sac qac))))
        (sc, sr)) =
        (map-spmf (map-prod (Inl  $\circ$  Inl) (?tri  $\circ$  prod.swap  $\circ$  Pair ((sar, sur), sr)))
        (map-spmf ( $\lambda((oadv, s-adv'), s-core'). (oadv, (s-adv', suc), s-core')$ )
        (exec-gpv (cfunc-adv core) (cnv-adv-core sac qac) se)))
      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]
spmfm.map-comp o-def split-def)
    next
      case ARest
      have bind-spmf (foldl-spmf (cpoke core) (return-spmf sc) es) ( $\lambda sc'$ .
        map-spmf rprodl (exec-gpv (fused-resource.fuse core rest)
        (left-gpv (map-gpv (map-prod Inl ( $\lambda s1'. (s1', suc, sur)$ )) id (right-gpv (map-gpv
(map-prod Inr (Pair sac)) id (cnv-adv-rest sar qar))))
        (sc', sr))) =
        bind-spmf
        (exec-gpv ( $\lambda(s, es) q. map-spmf ( $\lambda((out, e), s'). (out, s', es @ e)$ ) (rfunc-adv$ 
rest s q)) (cnv-adv-rest sar qar) (sr, es))
        (map-spmf (map-prod id ?tri)  $\circ$ 
        ( $\lambda((o-rfunc, e-lst), s-rfunc). map-spmf ( $\lambda s$ -notify. (Inl (Inr o-rfunc),$ 
```

```

s-notify, s-rfunc))
      (map-spmf (Pair (sac, suc)) (foldl-spmf (cpoke core) (return-spmf sc)
e-1st))) o
      (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
spmfm.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
spmfm.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 (λs1'. (s1', sur))) id (cnv-usr-core suc que))))))
(sc, sr)) =
(map-spmf (map-prod (Inr o Inl) (?tri o prod.swap o 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 que) 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]
spmfm.map-comp o-def split-def)
next
  case URest
  have bind-spmf (foldl-spmf (cpoke core) (return-spmf sc) es) (λsc'.
    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 (λs-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
spmfm.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
spmfm.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 spmfm.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':

$$(CNV\ cnv\ adv\ core\ s\ a\ c \mid =\ CNV\ cnv\ adv\ rest\ s\ a\ r) \mid = (CNV\ cnv\ usr\ core\ s\ u\ c \mid =\ CNV\ cnv\ usr\ rest\ s\ u\ r) \triangleright$$

```

  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 (λs-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-ucore*

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

qed

end

— Moving interfaces between rest and core

lemma

fuse-ishift-core-to-rest:

assumes $cpoke\ core' = (\lambda s. case-sum\ (\lambda q. fn\ s\ q)\ (cpoke\ core\ s))$
and $cfunc-adv\ core = cfunc-adv\ core'$
and $cfunc-usr\ core = cfunc-usr\ core' \oplus_O (\lambda s\ i. map-spmf\ (Pair\ (h-out\ i))\ (fn\ s\ i))$
and $rfunc-adv\ rest' = (\lambda s\ q. map-spmf\ (apfst\ (apsnd\ (map\ Inr))))\ (rfunc-adv\ rest\ s\ q)$
and $rfunc-usr\ rest' = plus-eoracle\ (\lambda s\ i. return-spmf\ ((h-out\ i, [i]), s))\ (rfunc-usr\ rest)$
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 $x-ifunc \equiv (\lambda ifunc\ core\ (se, sc)\ q. do\ \{$
 $((out, es), se') \leftarrow ifunc\ se\ q;$
 $sc' \leftarrow foldl-spmf\ (cpoke\ core)\ (return-spmf\ sc)\ es;$
 $return-spmf\ (out, se', sc')\ \}$
assumes $cpoke\ core' = cpoke\ (translate-core\ etran\ core)$
and $cfunc-adv\ core' = \dagger(cfunc-adv\ core) \oplus_O\ x-ifunc\ ifunc\ core$
and $cfunc-usr\ core' = cfunc-usr\ (translate-core\ etran\ core)$

```

and rinit rest = (einit, rinit rest')
and rfunc-adv rest = (λs 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 (rassow |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 rassow-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 *I-real* :: ('a, 'b) \mathcal{I}
 and *I-ideal* :: ('c, 'd) \mathcal{I}
 and *I-common* :: ('e, 'f) \mathcal{I}
 and *adv* :: real
 assumes *WT-real* [*WT-intro*]: $\mathcal{I}\text{-real} \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \vdash_{\text{res}} \text{real-resource} \checkmark$
 and *WT-ideal* [*WT-intro*]: $\mathcal{I}\text{-ideal} \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \vdash_{\text{res}} \text{ideal-resource} \checkmark$
 and *WT-sim* [*WT-intro*]: $\mathcal{I}\text{-real}, \mathcal{I}\text{-ideal} \vdash_C \text{sim} \checkmark$
 and *pfinite-sim* [*pfinite-intro*]: *pfinite-converter* $\mathcal{I}\text{-real} \mathcal{I}\text{-ideal} \text{sim}$
 and *adv-nonneg*: $0 \leq \text{adv}$

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 *I-real* :: ('a, 'b) \mathcal{I}
 and *I-common* :: ('e, 'f) \mathcal{I}
 and *A* :: ('a + 'e, 'b + 'f) *distinguisher-obsf*
 and *adv* :: real
 assumes *adv*: [*exception-I* ($\mathcal{I}\text{-real} \oplus_{\mathcal{I}} \mathcal{I}\text{-common}$) $\vdash_g \mathcal{A} \checkmark$]
 $\implies \text{advantage } \mathcal{A} (\text{obsf-resource } (\text{sim} \mid= 1_C \triangleright \text{ideal-resource})) (\text{obsf-resource } (\text{real-resource})) \leq \text{adv}$

locale *constructive-security-obsf* = *constructive-security-aux-obsf* *real-resource ideal-resource sim I-real I-ideal I-common adv*
 + *constructive-security-sim-obsf* *real-resource ideal-resource sim I-real I-common 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 *I-real* :: ('a, 'b) \mathcal{I}
 and *I-ideal* :: ('c, 'd) \mathcal{I}
 and *I-common* :: ('e, 'f) \mathcal{I}
 and *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 $|_ = 1_C \triangleright$ 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 $cnv \checkmark$ exception- \mathcal{I} \mathcal{I} , exception- \mathcal{I} (\mathcal{I} -real $\oplus_{\mathcal{I}}$ \mathcal{I} -common) \vdash_C $cnv' \checkmark$ exception- \mathcal{I} $\mathcal{I} \vdash g \mathcal{A} \checkmark$*

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} ($cnv' \triangleright$ obsf-resource (sim $|_ = 1_C \triangleright$ ideal-resource)) = connect-obsf \mathcal{A} ($cnv \triangleright$ obsf-resource (sim $|_ = 1_C \triangleright$ ideal-resource))*

connect-obsf \mathcal{A} ($cnv' \triangleright$ obsf-resource real-resource) = connect-obsf \mathcal{A} ($cnv \triangleright$ 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 $|_ = 1_C \triangleright$ ideal-resource)) (obsf-resource real-resource) =*

advantage (absorb \mathcal{A} cnv) (obsf-resource (sim $|_ = 1_C \triangleright$ ideal-resource)) (obsf-resource real-resource)

unfolding *advantage-def distinguish-attach[symmetric]* **by** *simp*

also have $\dots \leq adv$ **by**(*rule adv*)(*rule WT-intro*)+

finally show *advantage (absorb \mathcal{A} $cnv')$ (obsf-resource (sim $|_ = 1_C \triangleright$ 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 I-real I-ideal I-common A adv*

and *cong*: *I-real, I-ideal ⊢_C sim ~ sim'*

and *pfinite* [*pfinite-intro*]: *pfinite-converter I-real I-ideal sim'*

shows *constructive-security-obsf real-resource ideal-resource sim' I-real I-ideal I-common A adv*

proof

interpret *constructive-security-obsf real-resource ideal-resource sim I-real I-ideal I-common A adv* **by fact**

show *I-real ⊕_I I-common ⊢_{res} real-resource √ I-ideal ⊕_I I-common ⊢_{res} ideal-resource √* **by**(*rule WT-intro*)**+**

from *cong* **show** [*WT-intro*]: *I-real, I-ideal ⊢_C sim' √* **by**(*rule eq-I-converterD-WT1*)(*rule WT-intro*)

show *pfinite-converter I-real I-ideal sim'* **by fact**

show $0 \leq \text{adv}$ **by**(*rule adv-nonneg*)

assume *WT* [*WT-intro*]: *exception-I (I-real ⊕_I I-common) ⊢_g A √*

have *connect-obsf A (obsf-resource (sim' |₌ 1_C ▷ ideal-resource)) = connect-obsf A (obsf-resource (sim |₌ 1_C ▷ ideal-resource))*

by(*rule connect-eq-resource-cong WT-intro obsf-resource-eq-I-cong eq-I-attach-on' parallel-converter2-eq-I-cong cong[symmetric] eq-I-converter-reflI | simp*)**+**

with *adv[OF WT]*

show *advantage A (obsf-resource (sim' |₌ 1_C ▷ ideal-resource)) (obsf-resource real-resource) ≤ 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 I-real I-ideal (I-common-core ⊕_I I-common-rest) adv*

and *adv*: $\llbracket \text{exception-I (I-real } \oplus_{\mathcal{I}} (\text{I-common-core } \oplus_{\mathcal{I}} \text{I-common-rest)) } \vdash_{\mathcal{G}} \mathcal{A} \sqrt{\quad} \rrbracket$

$\implies \text{advantage } \mathcal{A} (\text{obsf-resource (sim |}_{=} (1_{\mathcal{C}} |_{=} 1_{\mathcal{C}}) \triangleright \text{ideal-resource})) (\text{obsf-resource (real-resource)}) \leq \text{adv}$

shows *constructive-security-obsf real-resource ideal-resource sim I-real I-ideal (I-common-core ⊕_I I-common-rest) A adv*

proof –

interpret *constructive-security-aux-obsf real-resource ideal-resource sim I-real I-ideal I-common-core ⊕_I I-common-rest* **by fact**

show *?thesis*

proof

assume *A* [*WT-intro*]: *exception-I (I-real ⊕_I (I-common-core ⊕_I I-common-rest)) ⊢_g A √*

hence *outs*: *outs-gpv (exception-I (I-real ⊕_I (I-common-core ⊕_I I-common-rest))) A ⊆ outs-I (I-real ⊕_I (I-common-core ⊕_I I-common-rest))*

unfolding *WT-gpv-iff-outs-gpv* **by** *simp*
have *connect-obsf* \mathcal{A} (*obsf-resource* (*sim* $\models 1_C \triangleright$ *ideal-resource*)) = *connect-obsf*
 \mathcal{A} (*obsf-resource* (*sim* $\models 1_C \models 1_C \triangleright$ *ideal-resource*))
by(*rule connect-cong-trace trace-eq-obsf-resourceI eq-resource-on-imp-trace-eq*
eq-I-attach-on) +
(*rule WT-intro parallel-converter2-eq-I-cong eq-I-converter-reflI paral-*
lel-converter2-id-id[symmetric] order-refl outs) +
then show *advantage* \mathcal{A} (*obsf-resource* (*sim* $\models 1_C \triangleright$ *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 I-middle I-inner*
I-common (*absorb* \mathcal{A} (*obsf-converter* (*sim-outer* $\models 1_C$))) *adv1*
assumes *constructive-security-obsf real middle sim-outer I-real I-middle I-common*
 \mathcal{A} *adv2*
shows *constructive-security-obsf real ideal (sim-outer \odot sim-inner) I-real I-inner*
I-common \mathcal{A} (*adv1* + *adv2*)
proof
let $?A =$ *absorb* \mathcal{A} (*obsf-converter* (*sim-outer* $\models 1_C$))
interpret *inner*: *constructive-security-obsf middle ideal sim-inner I-middle I-inner*
I-common $?A$ *adv1* **by** *fact*
interpret *outer*: *constructive-security-obsf real middle sim-outer I-real I-middle*
I-common \mathcal{A} *adv2* **by** *fact*

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

assume *WT-adv[WT-intro]*: *exception-I (I-real $\oplus_{\mathcal{I}}$ I-common) \vdash_g \mathcal{A} \checkmark
have *eq1*: *connect-obsf (absorb* \mathcal{A} (*obsf-converter* (*sim-outer* $\models 1_C$))) (*obsf-resource*
(*sim-inner* $\models 1_C \triangleright$ *ideal*)) =
connect-obsf \mathcal{A} (*obsf-resource* (*sim-outer* \odot *sim-inner* $\models 1_C \triangleright$ *ideal*))
unfolding *distinguish-attach[symmetric]*
apply(*rule connect-eq-resource-cong*)
apply(*rule WT-intro*)
apply(*simp del: outs-plus-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* $\models 1_C$))) (*obsf-resource**

middle) =
 connect-obsf \mathcal{A} (obsf-resource (sim-outer $\models 1_C \triangleright$ 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 $\models 1_C \triangleright$ ideal)) (obsf-resource
 middle) \leq adv1
 by(rule inner.adv)(rule WT-intro)+
 moreover have advantage \mathcal{A} (obsf-resource (sim-outer $\models 1_C \triangleright$ 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 $\models 1_C \triangleright$ 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') $\vdash_g \mathcal{A} \checkmark$*
 \implies *constructive-security-sim-obsf real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -common*
(absorb \mathcal{A} (obsf-converter (w-adv-real \models w-usr))) adv
(is - \implies constructive-security-sim-obsf - - - - ? \mathcal{A} -)

assumes *WT-usr [WT-intro]: \mathcal{I} -common', \mathcal{I} -common \vdash_C w-usr \checkmark*
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 \checkmark*
and *WT-w-adv-ideal [WT-intro]: \mathcal{I} -ideal', \mathcal{I} -ideal \vdash_C w-adv-ideal \checkmark*
and *WT-adv-ideal-inv [WT-intro]: \mathcal{I} -ideal, \mathcal{I} -ideal' \vdash_C w-adv-ideal-inv \checkmark*
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 \models w-usr \triangleright real-resource) (w-adv-ideal*
 \models 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

(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 \checkmark*
and *\mathcal{I} -ideal' $\oplus_{\mathcal{I}}$ \mathcal{I} -common' \vdash_{res} ?ideal \checkmark*
and *\mathcal{I} -real', \mathcal{I} -ideal' \vdash_C ?sim \checkmark 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')  $\vdash_g$   $\mathcal{A}$   $\checkmark$ 
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  $\models_{1_C}$   $\triangleright$  ideal-resource)) (obsf-resource
real-resource)  $\leq$  adv
  by(rule adv)(rule WT-intro)+

have ideal: connect-obsf ? $\mathcal{A}$  (obsf-resource (sim  $\models_{1_C}$   $\triangleright$  ideal-resource)) =
connect-obsf  $\mathcal{A}$  (obsf-resource (?sim  $\models_{1_C}$   $\triangleright$  ?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  $\models_{1_C}$ )  $\triangleright$  ?ideal)) (obsf-resource ?real)
 $\leq$  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 I-real I-ideal I-common (absorb A (obsf-converter (w-adv-real \models w-usr)))* *adv*
assumes *WT-usr* [*WT-intro*]: *I-common'*, *I-common* \vdash_C *w-usr* \checkmark
and *pfinite* [*pfinite-intro*]: *pfinite-converter I-common' I-common w-usr*
and *WT-adv-real* [*WT-intro*]: *I-real'*, *I-real* \vdash_C *w-adv-real* \checkmark
and *WT-w-adv-ideal* [*WT-intro*]: *I-ideal'*, *I-ideal* \vdash_C *w-adv-ideal* \checkmark
and *WT-adv-ideal-inv* [*WT-intro*]: *I-ideal*, *I-ideal'* \vdash_C *w-adv-ideal-inv* \checkmark
and *ideal-inverse*: *I-ideal*, *I-ideal* \vdash_C *w-adv-ideal-inv* \odot *w-adv-ideal* $\sim 1_C$
and *pfinite-real* [*pfinite-intro*]: *pfinite-converter I-real' I-real w-adv-real*
and *pfinite-ideal* [*pfinite-intro*]: *pfinite-converter I-ideal I-ideal' w-adv-ideal-inv*
shows *constructive-security-obsf (w-adv-real \models w-usr \triangleright real-resource) (w-adv-ideal \models w-usr \triangleright ideal-resource) (w-adv-real \odot sim \odot w-adv-ideal-inv) I-real' I-ideal' I-common' A adv*

proof –

interpret *constructive-security-obsf real-resource ideal-resource sim I-real I-ideal I-common absorb A (obsf-converter (w-adv-real \models 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 I-real I-ideal I-common adv*
and *sec2*: *exception-I (I-real \oplus_I I-common') \vdash_g A* \checkmark
 \implies *constructive-security-sim-obsf real-resource ideal-resource sim I-real I-common (absorb A (obsf-converter (1_C \models conv)))* *adv*
and *WT-conv* [*WT-intro*]: *I-common'*, *I-common* \vdash_C *conv* \checkmark
and *pfinite* [*pfinite-intro*]: *pfinite-converter I-common' I-common conv*
shows *constructive-security-obsf (1_C \models conv \triangleright real-resource) (1_C \models conv \triangleright ideal-resource) sim I-real I-ideal I-common' 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 (I-real1 \oplus_I I-real2) (I-ideal1 \oplus_I I-ideal2) I-common adv*
and *sec2*: *exception-I ((I-real1 \oplus_I I-real2) \oplus_I I-common') \vdash_g A* \checkmark
 \implies *constructive-security-sim-obsf real-resource ideal-resource sim (I-real1 \oplus_I I-real2) I-common (absorb A (obsf-converter ((1_C \models 1_C) \models conv)))* *adv*
assumes *WT-conv* [*WT-intro*]: *I-common'*, *I-common* \vdash_C *conv* \checkmark
and *pfinite* [*pfinite-intro*]: *pfinite-converter I-common' I-common conv*
shows *constructive-security-obsf ((1_C \models 1_C) \models conv \triangleright real-resource) ((1_C \models 1_C) \models conv \triangleright ideal-resource) sim (I-real1 \oplus_I I-real2) (I-ideal1 \oplus_I I-ideal2) I-common' A adv*
(is constructive-security-obsf ?real ?ideal - ?I-real ?I-ideal - - -)

proof –

interpret *constructive-security-aux-obsf real-resource ideal-resource sim I-real1*

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

theorem $\text{constructive-security-obsf-trivial}$:

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

assume $\text{WT} [\text{WT-intro}]: \text{exception-}\mathcal{I} (\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}\text{-common}) \vdash_g \mathcal{A} \checkmark$
have $\text{connect-obsf } \mathcal{A} (\text{obsf-resource } (1_C \models 1_C \triangleright \text{res})) = \text{connect-obsf } \mathcal{A} (\text{obsf-resource } (1_C \triangleright \text{res}))$
by(rule $\text{connect-req-resource-cong}$ [$OF \text{ WT}$])($\text{fastforce intro: WT-intro eq-}\mathcal{I}\text{-attach-on' obsf-resource-eq-}\mathcal{I}\text{-cong parallel-converter2-id-id}$)+
then show $\text{advantage } \mathcal{A} (\text{obsf-resource } (1_C \models 1_C \triangleright \text{res})) (\text{obsf-resource } \text{res}) \leq 0$
unfolding advantage-def **by** simp
qed simp

lemma $\text{parallel-constructive-security-aux-obsf}$ [locale-witness]:

assumes $\text{constructive-security-aux-obsf real1 ideal1 sim1 } \mathcal{I}\text{-real1 } \mathcal{I}\text{-inner1 } \mathcal{I}\text{-common1 } \text{adv1}$
assumes $\text{constructive-security-aux-obsf real2 ideal2 sim2 } \mathcal{I}\text{-real2 } \mathcal{I}\text{-inner2 } \mathcal{I}\text{-common2 } \text{adv2}$
shows $\text{constructive-security-aux-obsf} (\text{parallel-wiring } \triangleright \text{real1 } \parallel \text{real2}) (\text{parallel-wiring } \triangleright \text{ideal1 } \parallel \text{ideal2}) (\text{sim1 } \models \text{sim2})$
 $(\mathcal{I}\text{-real1 } \oplus_{\mathcal{I}} \mathcal{I}\text{-real2}) (\mathcal{I}\text{-inner1 } \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2}) (\mathcal{I}\text{-common1 } \oplus_{\mathcal{I}} \mathcal{I}\text{-common2})$
 $(\text{adv1} + \text{adv2})$

proof

interpret $\text{sec1: constructive-security-aux-obsf real1 ideal1 sim1 } \mathcal{I}\text{-real1 } \mathcal{I}\text{-inner1 } \mathcal{I}\text{-common1 } \text{adv1}$ **by fact**

interpret $\text{sec2: constructive-security-aux-obsf real2 ideal2 sim2 } \mathcal{I}\text{-real2 } \mathcal{I}\text{-inner2 } \mathcal{I}\text{-common2 } \text{adv2}$ **by fact**

show $(\mathcal{I}\text{-real1 } \oplus_{\mathcal{I}} \mathcal{I}\text{-real2}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \oplus_{\mathcal{I}} \mathcal{I}\text{-common2}) \vdash_{\text{res}} \text{parallel-wiring } \triangleright \text{real1 } \parallel \text{real2} \checkmark$

and $(\mathcal{I}\text{-inner1 } \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1 } \oplus_{\mathcal{I}} \mathcal{I}\text{-common2}) \vdash_{\text{res}} \text{parallel-wiring } \triangleright \text{ideal1 } \parallel \text{ideal2} \checkmark$

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

theorem *parallel-constructive-security-obsf*:

assumes *constructive-security-obsf real1 ideal1 sim1* $\mathcal{I}\text{-real1} \mathcal{I}\text{-inner1} \mathcal{I}\text{-common1}$
(*absorb* \mathcal{A} (*obsf-converter* (*parallel-wiring* \odot *parallel-converter* 1_C (*converter-of-resource*
(*sim2* $\mid = 1_C \triangleright \text{ideal2}$)))))) *adv1*
(**is** *constructive-security-obsf* - - - - - ?*A1* -)
assumes *constructive-security-obsf real2 ideal2 sim2* $\mathcal{I}\text{-real2} \mathcal{I}\text{-inner2} \mathcal{I}\text{-common2}$
(*absorb* \mathcal{A} (*obsf-converter* (*parallel-wiring* \odot *parallel-converter* (*converter-of-resource*
real1 1_C)))) *adv2*
(**is** *constructive-security-obsf* - - - - - ?*A2* -)
shows *constructive-security-obsf* (*parallel-wiring* \triangleright *real1* \parallel *real2*) (*parallel-wiring*
 \triangleright *ideal1* \parallel *ideal2*) ($\text{sim1} \mid = \text{sim2}$)
($\mathcal{I}\text{-real1} \oplus_{\mathcal{I}} \mathcal{I}\text{-real2}$) ($\mathcal{I}\text{-inner1} \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2}$) ($\mathcal{I}\text{-common1} \oplus_{\mathcal{I}} \mathcal{I}\text{-common2}$)
 \mathcal{A} ($\text{adv1} + \text{adv2}$)

proof –

interpret *sec1*: *constructive-security-obsf real1 ideal1 sim1* $\mathcal{I}\text{-real1} \mathcal{I}\text{-inner1}$
 $\mathcal{I}\text{-common1}$?*A1* *adv1* **by** *fact*

interpret *sec2*: *constructive-security-obsf real2 ideal2 sim2* $\mathcal{I}\text{-real2} \mathcal{I}\text{-inner2}$
 $\mathcal{I}\text{-common2}$?*A2* *adv2* **by** *fact*

show ?*thesis*

proof

assume *WT* [*WT-intro*]: *exception- \mathcal{I}* ($(\mathcal{I}\text{-real1} \oplus_{\mathcal{I}} \mathcal{I}\text{-real2}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1}$
 $\oplus_{\mathcal{I}} \mathcal{I}\text{-common2})) \vdash_g \mathcal{A} \checkmark$

have **: *outs- \mathcal{I}* ($(\mathcal{I}\text{-real1} \oplus_{\mathcal{I}} \mathcal{I}\text{-real2}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-common1} \oplus_{\mathcal{I}} \mathcal{I}\text{-common2})) \vdash_R$
($(1_C \mid = \text{sim2}) \mid = 1_C \mid = 1_C$) \odot *parallel-wiring* \triangleright *real1* \parallel *ideal2* \sim
parallel-wiring \odot (*converter-of-resource* *real1* $\mid_{\infty} 1_C$) \triangleright *sim2* $\mid = 1_C \triangleright$ *ideal2*

unfolding *comp-parallel-wiring*

by(rule *eq-resource-on-trans*, rule *eq- \mathcal{I} -attach-on*[**where** *conv'*=*parallel-wiring*
 \odot ($1_C \mid = \text{sim2} \mid = 1_C$)])
, (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- \mathcal{I} -cong parallel-converter2-id-id
eq- \mathcal{I} -converter-refl)

have *ideal2*:

connect-obsf ?*A2* (*obsf-resource* (*sim2* $\mid = 1_C \triangleright$ *ideal2*)) =

connect-obsf \mathcal{A} (*obsf-resource* ($(1_C \mid = \text{sim2}) \mid = (1_C \mid = 1_C) \triangleright$ *parallel-wiring*
 \triangleright *real1* \parallel *ideal2*))

unfolding *distinguish-attach*[*symmetric*]

apply(rule *connect- \mathcal{I} -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  $\triangleright$  real1 || 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) |= ( $1_C$  |=  $1_C$ )  $\triangleright$  parallel-wiring  $\triangleright$  real1 || ideal2))
(obsf-resource (parallel-wiring  $\triangleright$  real1 || 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 |=  $1_C$   $\triangleright$  ideal1)) =
  connect-obsf  $\mathcal{A}$  (obsf-resource ((sim1 |= sim2) |= ( $1_C$  |=  $1_C$ )  $\triangleright$  parallel-wiring
 $\triangleright$  ideal1 || ideal2))
proof –
  have *:((outs- $\mathcal{I}$   $\mathcal{I}$ -real1  $\langle + \rangle$  outs- $\mathcal{I}$   $\mathcal{I}$ -real2)  $\langle + \rangle$  outs- $\mathcal{I}$   $\mathcal{I}$ -common1  $\langle + \rangle$ 
outs- $\mathcal{I}$   $\mathcal{I}$ -common2)  $\vdash_R$ 
(sim1 |= sim2) |= ( $1_C$  |=  $1_C$ )  $\triangleright$  parallel-wiring  $\triangleright$  ideal1 || ideal2  $\sim$ 
parallel-wiring  $\odot$  ( $1_C$  | $\propto$  converter-of-resource (sim2 |=  $1_C$   $\triangleright$  ideal2))  $\triangleright$  sim1
|=  $1_C$   $\triangleright$  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 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 \odot ((1C |= 1C) |= sim2 |= 1C) ▷ real1 || ideal2 \sim parallel-wiring \odot (1C $|_{\infty}$ converter-of-resource (sim2 |= 1C ▷ ideal2)) ▷ real1
by(rule eq-resource-on-trans, rule eq- \mathcal{I} -attach-on[**where** conv'=parallel-wiring] \odot (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-refl)

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 A
(obsf-resource ((sim1 |= sim2) |= (1C |= 1C) ▷ parallel-wiring ▷ ideal1 || ideal2))
(obsf-resource ((1C |= sim2) |= (1C |= 1C) ▷ parallel-wiring ▷ real1 || ideal2))
 \leq adv1
unfolding advantage-def ideal1[symmetric] real1[symmetric] **by**(rule sec1.adv[unfolded advantage-def])(rule WT-intro)+

from adv1 adv2 **show** advantage A (obsf-resource ((sim1 |= sim2) |= (1C |= 1C) ▷ parallel-wiring ▷ ideal1 || ideal2))
(obsf-resource (parallel-wiring ▷ real1 || real2)) \leq 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* $\oplus_{\mathcal{I}}$ *real1-rest*) (\mathcal{I} -*ideal1-core* $\oplus_{\mathcal{I}}$ *ideal1-rest*) (\mathcal{I} -*common1-core* $\oplus_{\mathcal{I}}$ *common1-rest*) (*absorb* \mathcal{A} (*obsf-converter* (*fused-wiring* \odot *parallel-converter* 1_C (*converter-of-resource* (*sim2* $\models 1_C \triangleright$ *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* $\oplus_{\mathcal{I}}$ *real2-rest*) (\mathcal{I} -*ideal2-core* $\oplus_{\mathcal{I}}$ *ideal2-rest*) (\mathcal{I} -*common2-core* $\oplus_{\mathcal{I}}$ *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* \triangleright *real1* \parallel *real2*) (*fused-wiring* \triangleright *ideal1* \parallel *ideal2*)

(*parallel-wiring* \odot (*sim1* \models *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* \triangleright *real1* \parallel *real2*) (*parallel-wiring* \triangleright *ideal1* \parallel *ideal2*) (*sim1* \models *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* \models *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} \checkmark

proof -

interpret *constructive-security-obsf*

parallel-wiring \triangleright *real1* \parallel *real2*

parallel-wiring \triangleright *ideal1* \parallel *ideal2*

sim1 \models *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* \models *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  $\Rightarrow$  ('a + 'e, 'b + 'f) resource
    and ideal-resource :: security  $\Rightarrow$  ('c + 'e, 'd + 'f) resource
    and sim :: security  $\Rightarrow$  ('a, 'b, 'c, 'd) converter
    and  $\mathcal{I}$ -real :: security  $\Rightarrow$  ('a, 'b)  $\mathcal{I}$ 
    and  $\mathcal{I}$ -ideal :: security  $\Rightarrow$  ('c, 'd)  $\mathcal{I}$ 
    and  $\mathcal{I}$ -common :: security  $\Rightarrow$  ('e, 'f)  $\mathcal{I}$ 
    and  $\mathcal{A}$  :: security  $\Rightarrow$  ('a + 'e, 'b + 'f) distinguisher-obsf
  assumes constructive-security-aux-obsf:  $\bigwedge \eta$ .
    constructive-security-aux-obsf (real-resource  $\eta$ ) (ideal-resource  $\eta$ ) (sim  $\eta$ ) ( $\mathcal{I}$ -real
 $\eta$ ) ( $\mathcal{I}$ -ideal  $\eta$ ) ( $\mathcal{I}$ -common  $\eta$ ) 0
    and adv:  $\llbracket \bigwedge \eta$ . exception- $\mathcal{I}$  ( $\mathcal{I}$ -real  $\eta \oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\eta$ )  $\vdash_g \mathcal{A} \eta \sqrt{\quad} \rrbracket$ 
       $\implies$  negligible ( $\lambda \eta$ . advantage ( $\mathcal{A} \eta$ ) (obsf-resource (sim  $\eta$ )  $\mid=$   $1_C \triangleright$  ideal-resource
 $\eta$ )) (obsf-resource (real-resource  $\eta$ )))
  begin

  sublocale constructive-security-aux-obsf
    real-resource  $\eta$ 
    ideal-resource  $\eta$ 
    sim  $\eta$ 
     $\mathcal{I}$ -real  $\eta$ 
     $\mathcal{I}$ -ideal  $\eta$ 
     $\mathcal{I}$ -common  $\eta$ 

```

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) (\mathcal{I} -real η)
 (\mathcal{I} -ideal η) (\mathcal{I} -common η) (\mathcal{A} η)
 (advantage (\mathcal{A} η) (obsf-resource (sim η $|=$ $1_C \triangleright$ ideal-resource η)) (obsf-resource
 (real-resource η)))
by(rule constructive-security-obsf-refl)

end

lemma *constructive-security-obsf'I*:
assumes $\bigwedge \eta$. constructive-security-obsf (real-resource η) (ideal-resource η) (sim
 η) (\mathcal{I} -real η) (\mathcal{I} -ideal η) (\mathcal{I} -common η) (\mathcal{A} η) (adv η)
and ($\bigwedge \eta$. exception- \mathcal{I} (\mathcal{I} -real $\eta \oplus_{\mathcal{I}}$ \mathcal{I} -common η) \vdash_g \mathcal{A} η \checkmark) \implies negligible adv
shows constructive-security-obsf' real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -ideal
 \mathcal{I} -common \mathcal{A}
proof –
interpret constructive-security-obsf
 real-resource η
 ideal-resource η
 sim η
 \mathcal{I} -real η
 \mathcal{I} -ideal η
 \mathcal{I} -common η
 \mathcal{A} η
 adv η
for η **by** fact
show ?thesis
proof
show negligible ($\lambda \eta$. advantage (\mathcal{A} η) (obsf-resource (sim η $|=$ $1_C \triangleright$ ideal-resource
 η)) (obsf-resource (real-resource η)))
if $\bigwedge \eta$. exception- \mathcal{I} (\mathcal{I} -real $\eta \oplus_{\mathcal{I}}$ \mathcal{I} -common η) \vdash_g \mathcal{A} η \checkmark **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 $\bigwedge \mathcal{A} ::$ security \implies ('a + 'b, 'c + 'd) distinguisher-obsf.
 \llbracket $\bigwedge \eta$. interaction-bounded-by (λ -. True) (\mathcal{A} η) (bound η);
 $\bigwedge \eta$. lossless \implies plossless-gpv (exception- \mathcal{I} (\mathcal{I} -real $\eta \oplus_{\mathcal{I}}$ \mathcal{I} -common η)) (\mathcal{A} η)
 \rrbracket
 \implies constructive-security-obsf' real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -ideal
 \mathcal{I} -common \mathcal{A}
and correct: \exists *cnv*. $\forall \mathcal{D}$. ($\forall \eta$. \mathcal{I} -ideal $\eta \oplus_{\mathcal{I}}$ \mathcal{I} -common $\eta \vdash_g \mathcal{D}$ η \checkmark) \longrightarrow
 ($\forall \eta$. interaction-any-bounded-by (\mathcal{D} η) (bound η)) \longrightarrow

$(\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv } (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D} \eta)) \longrightarrow$
 $(\forall \eta. \text{wiring } (\mathcal{I}\text{-ideal } \eta) (\mathcal{I}\text{-real } \eta) (\text{cnv } \eta) (w \eta)) \wedge$
 $\text{Negligible.negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (\text{ideal-resource } \eta) (\text{cnv } \eta \mid =$
 $1_C \triangleright \text{real-resource } \eta))$
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 $\langle \lambda \cdot. \text{Done undefined} \rangle$*
by(*rule assms*) *simp-all*
show $\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{\text{res}} \text{real-resource } \eta \checkmark$
and $\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{\text{res}} \text{ideal-resource } \eta \checkmark$
and $\mathcal{I}\text{-real } \eta, \mathcal{I}\text{-ideal } \eta \vdash_C \text{sim } \eta \checkmark$ **for** η **by**(*rule WT-intro*)**+**

show $\exists \text{cnv}. \forall \mathcal{D}. (\forall \eta. \mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{D} \eta \checkmark) \longrightarrow$
 $(\forall \eta. \text{interaction-any-bounded-by } (\mathcal{D} \eta) (\text{bound } \eta)) \longrightarrow$
 $(\forall \eta. \text{lossless} \longrightarrow \text{plossless-gpv } (\mathcal{I}\text{-ideal } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{D} \eta)) \longrightarrow$
 $(\forall \eta. \text{wiring } (\mathcal{I}\text{-ideal } \eta) (\mathcal{I}\text{-real } \eta) (\text{cnv } \eta) (w \eta)) \wedge$
 $\text{Negligible.negligible } (\lambda \eta. \text{advantage } (\mathcal{D} \eta) (\text{ideal-resource } \eta) (\text{cnv } \eta \mid =$
 $1_C \triangleright \text{real-resource } \eta))$
by fact
next
fix $\mathcal{A} :: \text{security} \Rightarrow ('a + 'b, 'c + 'd) \text{distinguisher}$
assume $\text{WT-adv } [\text{WT-intro}]: \bigwedge \eta. \mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_g \mathcal{A} \eta \checkmark$
and $\text{bound } [\text{interaction-bound}]: \bigwedge \eta. \text{interaction-any-bounded-by } (\mathcal{A} \eta) (\text{bound } \eta)$
and $\text{lossless}: \bigwedge \eta. \text{lossless} \implies \text{plossless-gpv } (\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta) (\mathcal{A} \eta)$
let $?A = \lambda \eta. \text{obsf-distinguisher } (\mathcal{A} \eta)$
interpret *constructive-security-obsf' real-resource ideal-resource sim \mathcal{I} -real \mathcal{I} -ideal \mathcal{I} -common ?A*
proof(*rule assms*)
show *interaction-any-bounded-by* ($?A \eta$) (*bound* η) **for** η **by**(*rule interaction-bound*)**+**
show *plossless-gpv* (*exception- \mathcal{I}* ($\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta$)) ($?A \eta$) **if** *lossless*
for η
using *WT-adv*[*of* η] *lossless that* **by**(*simp*)
qed
have *negligible* ($\lambda \eta. \text{advantage } (?A \eta) (\text{obsf-resource } (\text{sim } \eta \mid = 1_C \triangleright \text{ideal-resource } \eta)) (\text{obsf-resource } (\text{real-resource } \eta)))$
by(*rule adv*)(*rule WT-intro*)**+**
then show *negligible* ($\lambda \eta. \text{advantage } (\mathcal{A} \eta) (\text{sim } \eta \mid = 1_C \triangleright \text{ideal-resource } \eta)$
(*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 I-middle I-inner I-common* ($\lambda\eta. \text{absorb } (\mathcal{A} \ \eta) \ (\text{obsf-converter } (\text{sim-outer } \eta \mid = 1_C))$)
assumes *constructive-security-obsf' real middle sim-outer I-real I-middle I-common* \mathcal{A}
shows *constructive-security-obsf' real ideal* ($\lambda\eta. \text{sim-outer } \eta \odot \text{sim-inner } \eta$) *I-real I-inner I-common* \mathcal{A}
proof(*rule constructive-security-obsf'I*)
let $?A = \lambda\eta. \text{absorb } (\mathcal{A} \ \eta) \ (\text{obsf-converter } (\text{sim-outer } \eta \mid = 1_C))$
interpret *inner: constructive-security-obsf' middle ideal sim-inner I-middle I-inner I-common ?A by fact*
interpret *outer: constructive-security-obsf' real middle sim-outer I-real I-middle I-common A by fact*

let $?adv1 = \lambda\eta. \text{advantage } (?A \ \eta) \ (\text{obsf-resource } (\text{sim-inner } \eta \mid = 1_C \triangleright \text{ideal } \eta))$
(obsf-resource (middle η))
let $?adv2 = \lambda\eta. \text{advantage } (\mathcal{A} \ \eta) \ (\text{obsf-resource } (\text{sim-outer } \eta \mid = 1_C \triangleright \text{middle } \eta))$
(obsf-resource (real η))
let $?adv = \lambda\eta. ?adv1 \ \eta + ?adv2 \ \eta$
show *constructive-security-obsf (real η) (ideal η) (sim-outer $\eta \odot$ sim-inner η) (I-real η) (I-inner η) (I-common η) ($\mathcal{A} \ \eta$) (?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-I (I-real $\eta \oplus_{\mathcal{I}}$ I-common η) $\vdash_g \mathcal{A} \ \eta \ \checkmark$ 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 I-real I-ideal I-common* ($\lambda\eta. \text{absorb } (\mathcal{A} \ \eta) \ (\text{obsf-converter } (1_C \mid = \text{conv } \eta))$)
assumes *WT-conv [WT-intro]: $\bigwedge\eta. \mathcal{I}\text{-common}' \ \eta, \mathcal{I}\text{-common } \eta \vdash_C \text{conv } \eta \ \checkmark$*
and *pfinite [pfinite-intro]: $\bigwedge\eta. \text{pfinite-converter } (\mathcal{I}\text{-common}' \ \eta) \ (\mathcal{I}\text{-common } \eta) \ (\text{conv } \eta)$*
shows *constructive-security-obsf'*
 $(\lambda\eta. 1_C \mid = \text{conv } \eta \triangleright \text{real-resource } \eta) \ (\lambda\eta. 1_C \mid = \text{conv } \eta \triangleright \text{ideal-resource } \eta) \ \text{sim}$
 $\mathcal{I}\text{-real } \mathcal{I}\text{-ideal } \mathcal{I}\text{-common}' \ \mathcal{A}$
proof(*rule constructive-security-obsf'I*)
let $?A = \lambda\eta. \text{absorb } (\mathcal{A} \ \eta) \ (\text{obsf-converter } (1_C \mid = \text{conv } \eta))$
interpret *constructive-security-obsf' real-resource ideal-resource sim I-real I-ideal I-common ?A by fact*
let $?adv = \lambda\eta. \text{advantage } (?A \ \eta) \ (\text{obsf-resource } (\text{sim } \eta \mid = 1_C \triangleright \text{ideal-resource } \eta)) \ (\text{obsf-resource } (\text{real-resource } \eta))$

fix $\eta :: \text{security}$
show *constructive-security-obsf (1_C | = conv $\eta \triangleright$ real-resource η) (1_C | = conv $\eta \triangleright$ ideal-resource η) (sim η) (I-real η) (I-ideal η) (I-common' η) ($\mathcal{A} \ \eta$) (?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]: $\wedge \eta. \text{exception-}\mathcal{I} (\mathcal{I}\text{-real } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common}' \eta)$*
 $\vdash g \mathcal{A} \eta \checkmark$
by(*rule adv*)(*rule WT-intro*)+
qed

theorem *constructive-security-obsf'-trivial*:

fixes *res*
assumes [*WT-intro*]: $\wedge \eta. \mathcal{I} \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common } \eta \vdash_{\text{res}} \text{res } \eta \checkmark$
shows *constructive-security-obsf' res res ($\lambda \cdot 1_C$) $\mathcal{I} \mathcal{I} \mathcal{I}\text{-common } \mathcal{A}$*
proof(*rule constructive-security-obsf'I*)
show *constructive-security-obsf (res η) (res η) $1_C (\mathcal{I} \eta) (\mathcal{I} \eta) (\mathcal{I}\text{-common } \eta) (\mathcal{A} \eta) 0$ for η*
using *assms* **by**(*rule constructive-security-obsf-trivial*)
qed *simp*

theorem *parallel-constructive-security-obsf'*:

assumes *constructive-security-obsf' real1 ideal1 sim1 $\mathcal{I}\text{-real1} \mathcal{I}\text{-inner1} \mathcal{I}\text{-common1}$*
($\lambda \eta. \text{absorb } (\mathcal{A} \eta) (\text{obsf-converter } (\text{parallel-wiring} \odot \text{parallel-converter } 1_C (\text{converter-of-resource } (\text{sim2 } \eta \models 1_C \triangleright \text{ideal2 } \eta))))$)
(is *constructive-security-obsf' - - - - - ?A1*)
assumes *constructive-security-obsf' real2 ideal2 sim2 $\mathcal{I}\text{-real2} \mathcal{I}\text{-inner2} \mathcal{I}\text{-common2}$*
($\lambda \eta. \text{absorb } (\mathcal{A} \eta) (\text{obsf-converter } (\text{parallel-wiring} \odot \text{parallel-converter } (\text{converter-of-resource } (\text{real1 } \eta)) 1_C))$)
(is *constructive-security-obsf' - - - - - ?A2*)
shows *constructive-security-obsf' ($\lambda \eta. \text{parallel-wiring} \triangleright \text{real1 } \eta \parallel \text{real2 } \eta$) ($\lambda \eta. \text{parallel-wiring} \triangleright \text{ideal1 } \eta \parallel \text{ideal2 } \eta$) ($\lambda \eta. \text{sim1 } \eta \models \text{sim2 } \eta$)*
($\lambda \eta. \mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta$) ($\lambda \eta. \mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta$) ($\lambda \eta. \mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta$) \mathcal{A}
proof(*rule constructive-security-obsf'I*)
interpret *sec1: constructive-security-obsf' real1 ideal1 sim1 $\mathcal{I}\text{-real1} \mathcal{I}\text{-inner1} \mathcal{I}\text{-common1} ?A1$ by fact*
interpret *sec2: constructive-security-obsf' real2 ideal2 sim2 $\mathcal{I}\text{-real2} \mathcal{I}\text{-inner2} \mathcal{I}\text{-common2} ?A2$ by fact*
let $?adv1 = \lambda \eta. \text{advantage } (?A1 \eta) (\text{obsf-resource } (\text{sim1 } \eta \models 1_C \triangleright \text{ideal1 } \eta)) (\text{obsf-resource } (\text{real1 } \eta))$
let $?adv2 = \lambda \eta. \text{advantage } (?A2 \eta) (\text{obsf-resource } (\text{sim2 } \eta \models 1_C \triangleright \text{ideal2 } \eta)) (\text{obsf-resource } (\text{real2 } \eta))$
let $?adv = \lambda \eta. ?adv1 \eta + ?adv2 \eta$
show *constructive-security-obsf (parallel-wiring $\triangleright \text{real1 } \eta \parallel \text{real2 } \eta$) (parallel-wiring $\triangleright \text{ideal1 } \eta \parallel \text{ideal2 } \eta$)*
($\text{sim1 } \eta \models \text{sim2 } \eta$) ($\mathcal{I}\text{-real1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-real2 } \eta$) ($\mathcal{I}\text{-inner1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-inner2 } \eta$)
($\mathcal{I}\text{-common1 } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-common2 } \eta$) ($\mathcal{A} \eta$)
($?adv \eta$) **for** η
using *sec1.constructive-security-obsf'D sec2.constructive-security-obsf'D*
by(*rule parallel-constructive-security-obsf*)

```

assume [WT-intro]: exception-I ((I-real1  $\eta \oplus_{\mathcal{I}}$  I-real2  $\eta$ )  $\oplus_{\mathcal{I}}$  (I-common1  $\eta \oplus_{\mathcal{I}}$ 
I-common2  $\eta$ ))  $\vdash_g \mathcal{A} \eta \surd$  for  $\eta$ 
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  $\times$  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  $\neq x \longleftrightarrow x = \textit{Bob}$ 
  by(cases x) simp-all

lemma neq-Alice-iff [simp]:  $x \neq \textit{Alice} \longleftrightarrow x = \textit{Bob}$ 
  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)

9.2 Defining the event handler

fun *poke* :: ('key 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))
| *poke* (PState-Store, s-shell) (Event-Kernel) = do {
 key \leftarrow spmf-of-set valid-keys;
 return-spmf (State-Store key, s-shell) }
| *poke* - - = return-pmf None

lemma *in-set-spmf-poke*:
 $s' \in \text{set-spmf } (\text{poke } s \ x) \longleftrightarrow$
 $(\exists \text{ s-kernel parties party. } s = (\text{s-kernel}, \text{parties}) \wedge x = \text{Event-Shell party} \wedge \text{party} \notin \text{parties} \wedge s' = (\text{s-kernel}, \text{insert party parties})) \vee$
 $(\exists \text{ s-shell key. } s = (\text{PState-Store}, \text{s-shell}) \wedge x = \text{Event-Kernel} \wedge \text{key} \in \text{valid-keys} \wedge \text{finite valid-keys} \wedge s' = (\text{State-Store key}, \text{s-shell}))$
by(cases (s, x) rule: poke.cases)(auto simp add: set-spmf-of-set)

lemma *foldl-poke-invar*:
 $\llbracket (\text{s-kernel}', \text{parties}') \in \text{set-spmf } (\text{foldl-spmf } \text{poke } p \ \text{events}); \forall (\text{s-kernel}, \text{parties}) \in \text{set-spmf } p. \text{set-s-kernel } \text{s-kernel} \subseteq \text{valid-keys} \rrbracket$
 $\implies \text{set-s-kernel } \text{s-kernel}' \subseteq \text{valid-keys}$
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* \Rightarrow - \Rightarrow - \Rightarrow '*inp* \Rightarrow ('wrap-key \times 'key state)
spmf

where

iface-usr-func party wrap (State-Store key, parties) inp =
 (if *party* \in *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 \equiv *iface-usr-func Alice Out-Alice*

abbreviation *iface-bob* :: ('key state, *iusr-bob*, 'key *ousr-bob*) oracle'

where

iface-bob \equiv *iface-usr-func Bob Out-Bob*

abbreviation *iface-usr* :: ('key state, *iusr*, 'key *ousr*) oracle'

where

iface-usr \equiv plus-oracle *iface-alice iface-bob*

lemma *in-set-iface-usr-func* [*simp*]:

x \in set-spmf (*iface-usr-func party wrap state inp*) \longleftrightarrow
 (\exists *key parties. state* = (*State-Store key, parties*) \wedge *party* \in *parties* \wedge *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 ( $\lambda$ key. key  $\in$  valid-keys)) ( $\lambda$ -. 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)  $\vdash$ res resource rest  $\checkmark$ 
  by(rule WT-intro)+ simp

```

end

end

theory Channel

imports

../Fused-Resource

begin

10 Channel specification

```

locale ideal-channel =
  fixes

```

```

    leak :: 'msg  $\Rightarrow$  '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  $\times$  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-mpmf (s-kernel, insert party parties))

lemma poke-alt-def:

poke = (λ (s, ps) e. map-mpmf (Pair s) (case e of Event-Shell party \Rightarrow if party \in ps then return-pmf None else return-mpmf (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 \equiv *plus-oracle iface-drop (plus-oracle iface-look iface-fedit)*

lemma *in-set-spmf-iface-drop*: $ys' \in \text{set-spmf } (iface\text{-drop } s \ x) \longleftrightarrow \text{False}$
by *simp*

lemma *in-set-spmf-iface-look*: $ys' \in \text{set-spmf } (iface\text{-look } s \ x) \longleftrightarrow$
 $(\exists \text{ msg parties. } s = (\text{State-Store } \text{msg}, \text{parties}) \wedge ys' = (\text{Out-Look } (\text{leak } \text{msg}),$
 $\text{State-Store } \text{msg}, \text{parties}))$
by(*cases* (*s*, *x*) *rule: iface-look.cases*) *simp-all*

lemma *in-set-spmf-iface-fedit*: $ys' \in \text{set-spmf } (iface\text{-fedit } s \ x) \longleftrightarrow$
 $(\exists \text{ msg parties msg'. } s = (\text{State-Store } \text{msg}, \text{parties}) \wedge x = (\text{Inp-Fedit } \text{msg'}) \wedge$
 $ys' = (\text{if } \text{editable} \text{ then } (\text{Out-Fedit}, \text{State-Collect } \text{msg}', \text{parties}) \text{ else } (\text{Out-Fedit},$
 $\text{State-Collect } \text{msg}, \text{parties})))$
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* \in *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

$\text{iface-bob } (\text{State-Collect } \text{msg}, \text{parties}) - =$
 (if $\text{Bob} \in \text{parties}$ then
 $\text{return-spmf } (\text{Out-Recv } \text{msg}, \text{State-Collected}, \text{parties})$
 else
 $\text{return-pmf } \text{None}$)
 | $\text{iface-bob} - - = \text{return-pmf } \text{None}$

abbreviation $\text{iface-usr} :: ('msg \text{state}, 'msg \text{iusr}, 'msg \text{ousr}) \text{oracle}'$
where
 $\text{iface-usr} \equiv \text{plus-oracle } \text{iface-alice } \text{iface-bob}$

lemma $\text{in-set-spmf-iface-alice}: \text{ys}' \in \text{set-spmf } (\text{iface-alice } s \ x) \longleftrightarrow$
 $(\exists \text{parties } \text{msg}. s = (\text{State-Void}, \text{parties}) \wedge x = \text{Inp-Send } \text{msg} \wedge \text{Alice} \in \text{parties}$
 $\wedge \text{ys}' = (\text{Out-Send}, \text{State-Store } \text{msg}, \text{parties}))$
by(cases (s, x) rule: iface-alice.cases) simp-all

lemma $\text{in-set-spmf-iface-bob}: \text{ys}' \in \text{set-spmf } (\text{iface-bob } s \ x) \longleftrightarrow$
 $(\exists \text{msg } \text{parties}. s = (\text{State-Collect } \text{msg}, \text{parties}) \wedge \text{Bob} \in \text{parties} \wedge \text{ys}' = (\text{Out-Recv}$
 $\text{msg}, \text{State-Collected}, \text{parties}))$
by(cases (s, x) rule: iface-bob.cases) simp-all

10.5 Defining the Fused Resource

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

sublocale $\text{fused-resource } \text{core} (\text{State-Void}, \{\}) .$

10.5.1 Lemma showing that the resulting resource is well-typed

lemma WT-core [WT-intro]:
 $\text{WT-core } (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform } (\text{Inp-Fedit } ' \text{valid-messages}) \text{UNIV}))$
 $(\mathcal{I}\text{-uniform } (\text{Inp-Send } ' \text{valid-messages}) \text{UNIV} \oplus_{\mathcal{I}} (\mathcal{I}\text{-uniform } \text{UNIV } (\text{Out-Recv } ' \text{valid-messages})))$
 $(\text{pred-prod } (\text{pred-s-kernel } (\lambda \text{msg}. \text{msg} \in \text{valid-messages})) (\lambda -. \text{True})) \text{core}$
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]: $\text{WT-rest } \mathcal{I}\text{-adv-rest } \mathcal{I}\text{-usr-rest } \mathcal{I}\text{-rest } \text{rest}$

```

shows (( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$  ( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform (Inp-Fedit ‘valid-messages) UNIV))
 $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -adv-rest)  $\oplus_{\mathcal{I}}$ 
  (( $\mathcal{I}$ -uniform (Inp-Send ‘valid-messages) UNIV  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -uniform UNIV (Out-Recv
‘valid-messages))  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -usr-rest)  $\vdash_{\text{res}}$  resource rest  $\checkmark$ 
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  $\Rightarrow$  'msg False +
  sec: ideal-channel  $\lambda$ - :: '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 \Rightarrow 'msg sec.iusr-alice

```

 $\Rightarrow$  (sec.ousr-alice  $\times$  unit, key.iusr-alice + 'msg sec.iusr-alice, 'msg key.ousr-alice
+ auth.ousr-alice) gpv

```

where

```

enc-callee  $\equiv$  stateless-callee ( $\lambda$ inp. case inp of sec.Inp-Send msg  $\Rightarrow$ 

```

```

  if msg  $\in$  carrier  $\mathcal{L}$  then

```

```

    Pause

```

```

    (Inl key.Inp-Alice)

```

```

    ( $\lambda$ kout. case projl kout of key.Out-Alice key  $\Rightarrow$ 

```

```

      let cipher = key  $\oplus$  msg in

```

```

      Pause (Inr (auth.Inp-Send cipher)) ( $\lambda$ -. Done sec.Out-Send))

```

```

  else

```

```

    Fail)

```

definition *dec-callee* :: *unit* \Rightarrow *sec.iusr-bob*
 \Rightarrow (*'msg sec.ousr-bob* \times *unit*, *key.iusr-bob* + *auth.iusr-bob*, *'msg key.ousr-bob* +
'msg auth.ousr-bob) *gpv*
where
dec-callee \equiv *stateless-callee* (λ -.
Pause
(Inr auth.Inp-Recv)
 $(\lambda$ *cout. case cout of*
Inr (auth.Out-Recv cipher) \Rightarrow
Pause
(Inl key.Inp-Bob)
 $(\lambda$ *kout. case projl kout of key.Out-Bob key \Rightarrow*
Done (sec.Out-Recv (key \oplus cipher)))
 $|$ - \Rightarrow *Fail*))

11.2 Defining adversary converter

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

definition *look-callee* :: *'msg astate* \Rightarrow *sec.iadv-look*
 \Rightarrow (*'msg sec.oadv-look* \times *'msg astate*, *sec.iadv-look*, *'msg set sec.oadv-look*) *gpv*
where
look-callee \equiv λ *state inp.*
Pause
sec.Inp-Look
 $(\lambda$ *cout. case cout of*
sec.Out-Look msg-set \Rightarrow
(case state of
None \Rightarrow do {
msg \leftarrow lift-spmf (spmf-of-set (msg-set));
Done (auth.Out-Look msg, Some msg) }
 $|$ *Some msg \Rightarrow 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 \equiv
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* \times (*key.party* + *auth.party*) *set*

abbreviation *einit* :: *estate*

where
einit \equiv (*False*, $\{\}$)

definition *sec-party-of-key-party* :: *key.party* \Rightarrow *sec.party*

where

sec-party-of-key-party \equiv *key.case-party* *sec.Alice* *sec.Bob*

abbreviation *etran-base-helper* :: *estate* \Rightarrow *key.party* + *auth.party* \Rightarrow *sec.event list*

where

etran-base-helper \equiv (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* \wedge *chk* then [*out*] else []) in

let *chk-alice* = *Inl* *key.Alice* \in *s-kap* \wedge *Inr* *auth.Alice* \in *s-kap* in

let *chk-bob* = *Inl* *key.Bob* \in *s-kap* \wedge *Inr* *auth.Bob* \in *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 \equiv (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* \notin *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* \in *s-kap* \wedge *Inr* *auth.Alice* \in *s-kap* in

let *check-bob* = *Inl* *key.Bob* \in *s-kap* \wedge *Inr* *auth.Bob* \in *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* \longleftrightarrow *p* = *key.Alice*

sec-party-of-key-party *p* = *sec.Bob* \longleftrightarrow *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 \in *sec-party-of-key-party* ' $P \longleftrightarrow$ *key.Alice* \in 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 \in *sec-party-of-key-party* ' $P \longleftrightarrow$ *key.Bob* \in 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 \equiv *translate-rest* *einit* *etran* (*parallel-rest* *key-rest* *auth-rest*)

definition *ideal-resource*

where

ideal-resource \equiv (*sim* $|=$ 1_C) $|=$ 1_C $|=$ $1_C \triangleright$ (*sec.resource* *ideal-rest*)

definition *real-resource*

where

real-resource \equiv *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'* :: ((- \times '*msg* *astate* \times -) \times -) \times *estate* \times '*msg* *sec.state*

where

ideal-s-core' \equiv ((((), *None*, ()), ()), (*False*, {}), *sec.State-Void*, {})

definition *ideal-s-rest'* :: - \times '*key-s-rest* \times '*auth-s-rest*

where

ideal-s-rest' \equiv ((((), ()), *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 ‡_I look-callee ‡_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* = (λs *ql. Generative-Probabilistic-Value.Done* (*Inl key.Out-Adversary*,
s)) in
 let *handle-r* = (λs *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 ($\lambda s. \text{case-sum}$ (*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'=($\lambda ()$)])
apply(*simp add: conv-callee-parallel-id-left[symmetric, where s=($\lambda ()$)])
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 ($\lambda ()$)])*
apply(*subst attach-parallel-fuse'[where f-init=Pair ($\lambda ()$, ($\lambda ()$))]*)
apply(*simp add: fact1[symmetric] fact2[symmetric, simplified Let-def]*)
done**

qed

11.6 Wiring and simplifying the Real construction

definition *real-s-core'* :: - \times 'msg *key.state* \times 'msg *auth.state*

where

real-s-core' \equiv ((($\lambda ()$), ($\lambda ()$), ($\lambda ()$)), (*key.PState-Store*, {}), (*auth.State-Void*, {}))

definition *real-s-rest'*

where

real-s-rest' \equiv *ideal-s-rest'*

primcorec $real-core' :: ((unit \times -) \times -, -, -, -, -, -) \text{ core}$
where
 $cpoke \text{ real-core}' = (\lambda(s-advusr, s-core) \text{ event}).$
 $map-spmf \text{ (Pair } s-advusr) \text{ (parallel-handler key.poke auth.poke s-core event)}$
 $| cfunc-adv \text{ real-core}' = \dagger(\text{key.iface-adv } \dagger_O \text{ auth.iface-adv})$
 $| cfunc-usr \text{ real-core}' = (\lambda((s-adv, s-usr), s-core) \text{ iusr}).$
 $let \text{ handle-req} = lsumr \circ map-sum \text{ id} (rsuml \circ map-sum \text{ swap-sum id} \circ lsumr)$
 $\circ rsuml \text{ in}$
 $let \text{ handle-ret} = lsumr \circ (map-sum \text{ id} (rsuml \circ (map-sum \text{ swap-sum id} \circ$
 $lsumr))) \circ rsuml \text{ in}$
 $map-spmf$
 $(\lambda((ousr, s-usr'), s-core'). (ousr, (s-adv, s-usr'), s-core'))$
 $(exec-gpv$
 $(\text{key.iface-usr } \dagger_O \text{ auth.iface-usr})$
 $(map-gpv' \text{ id handle-req handle-ret } ((\text{enc-callee } \dagger_I \text{ dec-callee}) \text{ s-usr iusr}))$
 $s-core))$

definition $real-rest'$
where
 $real-rest' \equiv ideal-rest'$

11.6.1 The real attachment lemma

private lemma $WT-callee-real1: ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})) \oplus_{\mathcal{I}}$
 $((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})) \vdash c$
 $(\text{key.fuse key-rest } \dagger_O \text{ auth.fuse auth-rest}) s \checkmark$
apply($rule \text{ WT-calleeI}$)
apply($cases \text{ s}$)
apply($case-tac \text{ call}$)
apply($rename-tac [!] \text{ x}$)
apply($case-tac [!] \text{ x}$)
apply($rename-tac [!] \text{ y}$)
apply($case-tac [!] \text{ y}$)
by($auto \text{ simp add: fused-resource.fuse.simps}$)

private lemma $WT-callee-real2: (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}}$
 $(\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})) \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \vdash c$
 $\text{fused-resource.fuse (parallel-core key.core auth.core) (parallel-rest key-rest auth-rest)}$
 $s \checkmark$
apply($rule \text{ WT-calleeI}$)
apply($cases \text{ s}$)
apply($case-tac \text{ call}$)
apply($rename-tac [!] \text{ x}$)
apply($case-tac [!] \text{ x}$)
apply($rename-tac [!] \text{ y}$)
apply($case-tac [!] \text{ y}$)
apply($rename-tac [5] \text{ z}$)
apply($rename-tac [6] \text{ z}$)
apply($case-tac [5] \text{ z}$)

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*
 \ddagger_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-I-attach-on^)
    apply (rule WT-resource-of-oracle[OF WT-callee-real2])
  apply(rule parallel-converter2-eq-I-cong)
  apply(rule eq-I-converter-refl)
  apply(rule WT-intro)+
  apply(rule parallel-converter2-eq-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 I=I-full and I'=I-full ⊕I
I-full])
      apply (rule WT-gpv-I-mono)
      apply (rule WT-gpv-full)
      apply (rule I-full-le-plus-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 I=I-full and I'=I-full ⊕I
I-full])
      apply (rule WT-gpv-I-mono)
      apply (rule WT-gpv-full)
      apply (rule I-full-le-plus-I)
      apply(rule order-refl)
      apply(rule order-refl)
      apply (clarsimp simp add: enc-callee-def stateless-callee-def split!:
sec.iusr-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

$$a-I \equiv \lambda(x, y). ((((), x, ()), ()), y)$$

private abbreviation

$$a-R \equiv \lambda x. ((((), ()), ()), x)$$

abbreviation

$$\begin{aligned}
asm-act &\equiv (\lambda flg \ pset-sec \ pset-key \ pset-auth \ pset-union. \\
&\quad pset-union = pset-key <+> pset-auth \wedge \\
&\quad (flg \longrightarrow pset-sec = sec-party-of-key-party ' pset-key \cap pset-auth))
\end{aligned}$$

private inductive $S :: (((- \times 'msg \ option \times -) \times -) \times estate \times 'msg \ sec.state)$

spmf

$$\Rightarrow (- \times 'msg \ key.state \times 'msg \ auth.state) \ spmf \Rightarrow \ bool$$

where

— (Auth =a)@(Key =0)

$$\begin{aligned}
s-0-0: S &(\text{return-spmf } (a-I \ (None, \ (False, \ s-act-ka), \ sec.State-Void, \ s-act-s))) \\
&(\text{return-spmf } (a-R \ ((key.PState-Store, \ s-act-k), \ auth.State-Void, \ s-act-a)))
\end{aligned}$$

if $asm-act \ False \ s-act-s \ s-act-k \ s-act-a \ s-act-ka$ **and** $s-act-s = \{\}$

— (Auth =a)@(Key =1)

$$\begin{aligned}
| \ s-0-1: S &(\text{return-spmf } (a-I \ (None, \ (True, \ s-act-ka), \ sec.State-Void, \ s-act))) \\
&(\text{map-spmf } (\lambda key. \ a-R \ ((key.State-Store \ key, \ s-act-k), \ auth.State-Void, \ s-act-a))) \\
&(\text{spmf-of-set } (\text{carrier } \mathcal{L}))
\end{aligned}$$

if $asm-act \ True \ s-act \ s-act-k \ s-act-a \ s-act-ka$

— $../(\text{Auth} =a)@(\text{Key} =1) \ \# \ w1$

$$\begin{aligned}
| \ s-1-1: S &(\text{return-spmf } (a-I \ (None, \ (True, \ s-act-ka), \ sec.State-Store \ msg, \ s-act-s))) \\
&(\text{map-spmf } (\lambda key. \ a-R \ ((key.State-Store \ key, \ s-act-k), \ auth.State-Store \ (key \oplus \\
&\text{msg}), \ s-act-a))) \ (\text{spmf-of-set } (\text{carrier } \mathcal{L}))
\end{aligned}$$

if $asm-act \ True \ s-act-s \ s-act-k \ s-act-a \ s-act-ka$ **and** $key.Alice \in s-act-k$ **and** $auth.Alice \in s-act-a$ **and** $msg \in \text{carrier } \mathcal{L}$

$$\begin{aligned}
| \ s-2-1: S &(\text{return-spmf } (a-I \ (None, \ (True, \ s-act-ka), \ sec.State-Collect \ msg, \\
&s-act-s)))
\end{aligned}$$

$$\begin{aligned}
&(\text{map-spmf } (\lambda key. \ a-R \ ((key.State-Store \ key, \ s-act-k), \ auth.State-Collect \ (key \\
&\oplus \ msg), \ s-act-a))) \ (\text{spmf-of-set } (\text{carrier } \mathcal{L}))
\end{aligned}$$

if *asm-act* *True s-act-s s-act-k s-act-a s-act-ka* **and** *key.Alice* \in *s-act-k* **and** *auth.Alice* \in *s-act-a* **and** *msg* \in *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* \oplus *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* \oplus *msg*), *s-act-a*)))
if *asm-act* *True s-act-s s-act-k s-act-a s-act-ka* **and** *key.Alice* \in *s-act-k* **and** *auth.Alice* \in *s-act-a* **and** *msg* \in *carrier* \mathcal{L} **and** *key* \in *carrier* \mathcal{L}
| *s-2'-1*: *S* (*return-spmf* (*a-I* (*Some* (*key* \oplus *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* \oplus *msg*), *s-act-a*)))
if *asm-act* *True s-act-s s-act-k s-act-a s-act-ka* **and** *key.Alice* \in *s-act-k* **and** *auth.Alice* \in *s-act-a* **and** *msg* \in *carrier* \mathcal{L} **and** *key* \in *carrier* \mathcal{L}
| *s-3'-1*: *S* (*return-spmf* (*a-I* (*Some* (*key* \oplus *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* \in *carrier* \mathcal{L} **and** *key* \in *carrier* \mathcal{L}

private lemma *trace-eq-core: trace-core-eq ideal-core' real-core'*

UNIV (*UNIV* $\langle + \rangle$ *UNIV* $\langle + \rangle$ *UNIV* $\langle + \rangle$ (*auth.Inp-Fedit* ' *carrier* \mathcal{L}))
(*sec.Inp-Send* ' *carrier* \mathcal{L}) $\langle + \rangle$ *UNIV*
(*return-spmf ideal-s-core'*) (*return-spmf real-s-core'*)

proof —

have *inj-xor*: \llbracket *msg* \in *carrier* \mathcal{L} ; *x* \in *carrier* \mathcal{L} ; *y* \in *carrier* \mathcal{L} ; *x* \oplus *msg* = *y* \oplus *msg* $\rrbracket \implies$ *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
  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 \equiv \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 \equiv \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' :: ((- \times 'msg astate \times -) \times -) \times estate \times 'msg sec.state \Rightarrow bool
where invar-ideal' = pred-prod (pred-prod (pred-prod (λ -. True) (pred-prod (pred-option (λ x. $x \in$ carrier \mathcal{L})) (λ -. True))) (λ -. True)) (pred-prod (λ -. True) (pred-prod (sec.pred-s-kernel (λ x. $x \in$ carrier \mathcal{L})) (λ -. True)))

private definition invar-real' :: - \times ('msg key.s-kernel \times -) \times 'msg sec.s-kernel \times - \Rightarrow bool
where invar-real' = pred-prod (λ -. True) (pred-prod (pred-prod (key.pred-s-kernel (λ x. $x \in$ carrier \mathcal{L})) (λ -. True)) (pred-prod (sec.pred-s-kernel (λ x. $x \in$ 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 I-adv-core I-usr-core invar-real' real-core'*
apply(rule *WT-core.intros*)
apply(*auto simp add: I-adv-core-def 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 *I-adv-restk I-adv-resta I-usr-restk I-usr-resta*
defines *outs-adv* \equiv (*UNIV* $\langle + \rangle$ *UNIV* $\langle + \rangle$ *UNIV* $\langle + \rangle$ *sec.Inp-Fedit* ' *carrier*
 \mathcal{L}) $\langle + \rangle$ *outs-I* (*I-adv-restk* $\oplus_{\mathcal{I}}$ *I-adv-resta*)
and *outs-usr* \equiv (*sec.Inp-Send* ' *carrier* \mathcal{L} $\langle + \rangle$ *UNIV*) $\langle + \rangle$ *outs-I* (*I-usr-restk*
 $\oplus_{\mathcal{I}}$ *I-usr-resta*)
assumes *WT-key* [*WT-intro*]: *WT-rest I-adv-restk I-usr-restk I-key-rest key-rest*

and *WT-auth* [*WT-intro*]: *WT-rest I-adv-resta I-usr-resta I-auth-rest auth-rest*
shows (*outs-adv* $\langle + \rangle$ *outs-usr*) \vdash_C *fused-resource.fuse ideal-core' ideal-rest'*
(*ideal-s-core'*, *ideal-s-rest'*) \approx
fused-resource.fuse real-core' real-rest' ((*real-s-core'*, *real-s-rest'*)

proof –

define *eI-adv-rest* :: (*-*, *-* \times (*key.event* + *auth.event*) *list*) \mathcal{I}
where *eI-adv-rest* \equiv *map-I id (case-sum (map-prod Inl (map Inl)) (map-prod*
Inr (map Inr))) (eI I-adv-restk $\oplus_{\mathcal{I}}$ *eI I-adv-resta*)
define *eI-usr-rest* :: (*-*, *-* \times (*key.event* + *auth.event*) *list*) \mathcal{I}
where *eI-usr-rest* \equiv *map-I id (case-sum (map-prod Inl (map Inl)) (map-prod*
Inr (map Inr))) (eI I-usr-restk $\oplus_{\mathcal{I}}$ *eI I-usr-resta*)

note *I-defs* = *I-adv-core-def I-usr-core-def*

note *eI-defs* = *eI-adv-rest-def eI-usr-rest-def*

have *fact1[unfolded outs-plus-I]*:

trace-rest-eq ideal-rest' ideal-rest' (*outs-I* (*I-adv-restk* $\oplus_{\mathcal{I}}$ *I-adv-resta*)) (*outs-I*
(*I-usr-restk* $\oplus_{\mathcal{I}}$ *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')* ($\lambda(-, s\text{-rest}). \text{pred-prod } I\text{-key-rest } I\text{-auth-rest } s\text{-rest}$) ($e\mathcal{I}\text{-adv-rest} \oplus_{\mathcal{I}} e\mathcal{I}\text{-usr-rest}$)

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-oracle-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-oracle-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'* ($\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-core} \oplus_{\mathcal{I}} \mathcal{I}\text{-usr-core})$))

by(*rule WT-intro*)+

have *fact4*[*unfolded I-defs*]: *callee-invariant-on (callee-of-core real-core')* (*invar-real'* ($\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-core} \oplus_{\mathcal{I}} \mathcal{I}\text{-usr-core})$))

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** $\mathcal{I}E = \mathcal{I}\text{-full}$ **and** $\mathcal{I}CA = \mathcal{I}\text{-adv-core}$ **and** $\mathcal{I}CU = \mathcal{I}\text{-usr-core}$ **and** $\mathcal{I}RA = e\mathcal{I}\text{-adv-rest}$ **and** $\mathcal{I}RU = e\mathcal{I}\text{-usr-rest}$, *unfolded eI-defs I-adv-core-def I-usr-core-def, simplified*]

apply (*rule fuse-trace-eq*[**where** $\mathcal{I}E = \mathcal{I}\text{-full}$ **and** $\mathcal{I}CA = \mathcal{I}\text{-adv-core}$ **and** $\mathcal{I}CU = \mathcal{I}\text{-usr-core}$ **and** $\mathcal{I}RA = e\mathcal{I}\text{-adv-rest}$ **and** $\mathcal{I}RU = e\mathcal{I}\text{-usr-rest}$ **and** $?IR1.0 = \lambda(-, s\text{-rest}). \text{pred-prod } I\text{-key-rest } I\text{-auth-rest } s\text{-rest}$ **and** $?IR2.0 = \lambda(-, s\text{-rest}). \text{pred-prod } I\text{-key-rest } I\text{-auth-rest } s\text{-rest}$ **and** $?IC1.0 = \text{invar-ideal'}$ **and** $?IC2.0 = \text{invar-real'}$, *unfolded eI-defs I-adv-core-def 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 I-adv-core-def I-usr-core-def*]:

fixes $\mathcal{I}\text{-adv-restk } \mathcal{I}\text{-adv-resta } \mathcal{I}\text{-usr-restk } \mathcal{I}\text{-usr-resta}$

defines $\mathcal{I} \equiv (\mathcal{I}\text{-adv-core} \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-restk} \oplus_{\mathcal{I}} \mathcal{I}\text{-adv-resta})) \oplus_{\mathcal{I}} (\mathcal{I}\text{-usr-core} \oplus_{\mathcal{I}} (\mathcal{I}\text{-usr-restk} \oplus_{\mathcal{I}} \mathcal{I}\text{-usr-resta}))$

assumes [*WT-intro*]: $WT\text{-rest } \mathcal{I}\text{-adv-restk } \mathcal{I}\text{-usr-restk } I\text{-key-rest } key\text{-rest}$ **and** [*WT-intro*]: $WT\text{-rest } \mathcal{I}\text{-adv-resta } \mathcal{I}\text{-usr-resta } I\text{-auth-rest } auth\text{-rest}$ **and** *exception-I* $\mathcal{I} \vdash_g D \checkmark$

```

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:  $\mathcal{I} \vdash_{res} RES$  (fused-resource.fuse ideal-core' ideal-rest') s  $\checkmark$ 
  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 in-
var-ideal' ( $\lambda(-, s-rest).$  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:  $\mathcal{I} \vdash_{res} RES$  (fused-resource.fuse real-core' real-rest') s  $\checkmark$ 
  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 in-
var-real' ( $\lambda(-, s-rest).$  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  $\mathcal{I}$ =exception- $\mathcal{I}$   $\mathcal{I}$ ])
  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* ()

✓

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* ()

✓

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** $I=\lambda\cdot$. *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** $I=\lambda\cdot$. *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 $\mathcal{I}\text{-real} \equiv ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (\text{sec.Inp-Fedit} \text{ ' (carrier } \mathcal{L})) \text{ UNIV})))) \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-restk} \oplus_{\mathcal{I}} \mathcal{I}\text{-adv-resta}))$

and $\mathcal{I}\text{-common-core} \equiv \mathcal{I}\text{-uniform} (\text{sec.Inp-Send} \text{ ' (carrier } \mathcal{L})) \text{ UNIV} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} \text{ UNIV} (\text{sec.Out-Recv} \text{ ' carrier } \mathcal{L})$

and $\mathcal{I}\text{-common-rest} \equiv \mathcal{I}\text{-usr-restk} \oplus_{\mathcal{I}} \mathcal{I}\text{-usr-resta}$

and $\mathcal{I}\text{-ideal} \equiv (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (\text{sec.Inp-Fedit} \text{ ' (carrier } \mathcal{L})) \text{ UNIV})) \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-restk} \oplus_{\mathcal{I}} \mathcal{I}\text{-adv-resta})$

shows $\text{constructive-security-obsf} (\text{real-resource } \text{TYPE}(-) \text{ TYPE}(-) \text{ key-rest} \text{ auth-rest}) (\text{sec.resource} (\text{ideal-rest} \text{ key-rest} \text{ auth-rest})) (\text{sim} \mid = 1_C) \mathcal{I}\text{-real} \mathcal{I}\text{-ideal} (\mathcal{I}\text{-common-core} \oplus_{\mathcal{I}} \mathcal{I}\text{-common-rest}) \mathcal{A} 0$

proof

let $?\mathcal{I}\text{-common} = \mathcal{I}\text{-common-core} \oplus_{\mathcal{I}} \mathcal{I}\text{-common-rest}$

show $\mathcal{I}\text{-real} \oplus_{\mathcal{I}} ?\mathcal{I}\text{-common} \vdash_{\text{res}} \text{real-resource } \text{TYPE}(-) \text{ TYPE}(-) \text{ key-rest} \text{ auth-rest} \checkmark$

unfolding $\mathcal{I}\text{-real-def} \mathcal{I}\text{-common-core-def} \mathcal{I}\text{-common-rest-def} \text{real-resource-def} \text{attach-c1f22-c1f22-def} \text{wiring-c1r22-c1r22-def} \text{fused-wiring-def}$

by(rule $\text{WT-intro} \mid \text{simp}$)+

show $[\text{WT-intro}]: \mathcal{I}\text{-ideal} \oplus_{\mathcal{I}} ?\mathcal{I}\text{-common} \vdash_{\text{res}} \text{sec.resource} (\text{ideal-rest} \text{ key-rest} \text{ auth-rest}) \checkmark$

unfolding $\mathcal{I}\text{-common-core-def} \mathcal{I}\text{-common-rest-def} \mathcal{I}\text{-ideal-def} \text{ideal-rest-def}$

by(rule WT-intro) + simp

show $[\text{WT-intro}]: \mathcal{I}\text{-real}, \mathcal{I}\text{-ideal} \vdash_C \text{sim} \mid = 1_C \checkmark$

unfolding $\mathcal{I}\text{-real-def} \mathcal{I}\text{-ideal-def}$

apply(rule WT-intro) +

subgoal

unfolding $\text{sim-def} \text{Let-def} \text{look-callee-def}$

apply (fold $\text{conv-callee-parallel-id-right}[\text{where } s' = ()]$)

apply (fold $\text{conv-callee-parallel-id-left}[\text{where } s = ()]$)

apply (subst $\text{ldummy-converter-of-callee}$)

apply (rule $\text{WT-converter-of-callee}$)

by (auto simp add: $\text{id-oracle-def} \text{map-gpv-conv-bind}[\text{symmetric}] \text{map-lift-spmf}$ $\text{split: auth.oadv-look.split option.split}$)

by (rule WT-intro)

show $\text{pfinite-converter} \mathcal{I}\text{-real} \mathcal{I}\text{-ideal} (\text{sim} \mid = 1_C)$

unfolding $\mathcal{I}\text{-real-def} \mathcal{I}\text{-ideal-def}$

apply(rule pfinite-intro) +

subgoal

unfolding $\text{sim-def} \text{Let-def} \text{look-callee-def}$

apply (fold $\text{conv-callee-parallel-id-right}[\text{where } s' = ()]$)

apply (fold $\text{conv-callee-parallel-id-left}[\text{where } s = ()]$)

apply (subst $\text{ldummy-converter-of-callee}$)

apply(rule $\text{raw-converter-invariant.pfinite-converter-of-callee}[\text{where } I = \lambda\text{-True}]$)

```

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 ( $\mathcal{I}$ -real  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common)  $\vdash_g \mathcal{A} \checkmark$ 
show advantage  $\mathcal{A}$  (obsf-resource ((sim  $\models 1_C$ )  $\models (1_C \models 1_C) \triangleright$  sec.resource
(ideal-rest key-rest auth-rest)))
  (obsf-resource (real-resource TYPE(-) TYPE(-) key-rest auth-rest))  $\leq 0$ 
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  $\mathcal{L} ::$  security  $\Rightarrow$  ('msg, 'more) boolean-algebra-scheme
  assumes one-time-pad [locale-witness]:  $\bigwedge \eta. \text{one-time-pad } (\mathcal{L} \ \eta)$ 
begin

sublocale one-time-pad  $\mathcal{L} \ \eta$  for  $\eta ..$ 

definition real-resource' where real-resource' rest1 rest2  $\eta =$  real-resource TYPE(-)
TYPE(-)  $\eta$  (rest1  $\eta$ ) (rest2  $\eta$ )
definition ideal-resource' where ideal-resource' rest1 rest2  $\eta =$  sec.resource  $\eta$ 
(ideal-rest (rest1  $\eta$ ) (rest2  $\eta$ ))
definition sim' where sim'  $\eta =$  (sim  $\models 1_C$ )

context
  fixes
    key-rest  $::$  nat  $\Rightarrow$  ('key-s-rest, key.event, 'key-iadv-rest, 'key-iusr-rest, 'key-oadv-rest,
'key-ousr-rest) rest-wstate and
    auth-rest  $::$  nat  $\Rightarrow$  ('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-res:  $\bigwedge \eta. \text{WT-rest } (\mathcal{I}\text{-adv-restk } \eta) (\mathcal{I}\text{-usr-restk } \eta) (I\text{-key-rest } \eta) (key\text{-rest } \eta)$ 
and
    WT-auth-rest:  $\bigwedge \eta. \text{WT-rest } (\mathcal{I}\text{-adv-resta } \eta) (\mathcal{I}\text{-usr-resta } \eta) (I\text{-auth-rest } \eta) (auth\text{-rest } \eta)$ 
begin

```

```

theorem secure':
  defines  $\mathcal{I}\text{-real} \equiv \lambda\eta. ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (\text{sec.Inp-Fedit}
    ' (\text{carrier } (\mathcal{L} \eta))) \text{UNIV}))) \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-restk } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-adv-resta } \eta))$ 
    and  $\mathcal{I}\text{-common} \equiv \lambda\eta. ((\mathcal{I}\text{-uniform} (\text{sec.Inp-Send } ' (\text{carrier } (\mathcal{L} \eta))) \text{UNIV} \oplus_{\mathcal{I}}
    \mathcal{I}\text{-uniform} \text{UNIV} (\text{sec.Out-Recv } ' \text{carrier } (\mathcal{L} \eta))) \oplus_{\mathcal{I}} (\mathcal{I}\text{-usr-restk } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-usr-resta }
    \eta))$ 
    and  $\mathcal{I}\text{-ideal} \equiv \lambda\eta. (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (\text{sec.Inp-Fedit } ' (\text{carrier}
    (\mathcal{L} \eta))) \text{UNIV})) \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-restk } \eta \oplus_{\mathcal{I}} \mathcal{I}\text{-adv-resta } \eta)$ 
    shows constructive-security-obsf' (real-resource' key-rest auth-rest) (ideal-resource'
    key-rest auth-rest) sim'  $\mathcal{I}\text{-real}$   $\mathcal{I}\text{-ideal}$   $\mathcal{I}\text{-common}$   $\mathcal{A}$ 
proof(rule constructive-security-obsf'I)
    show constructive-security-obsf (real-resource' key-rest auth-rest  $\eta$ ) (ideal-resource'
    key-rest auth-rest  $\eta$ )
      (sim'  $\eta$ ) ( $\mathcal{I}\text{-real}$   $\eta$ ) ( $\mathcal{I}\text{-ideal}$   $\eta$ ) ( $\mathcal{I}\text{-common}$   $\eta$ ) ( $\mathcal{A}$   $\eta$ ) 0 for  $\eta$ 
    unfolding real-resource'-def ideal-resource'-def sim'-def  $\mathcal{I}\text{-real-def}$   $\mathcal{I}\text{-common-def}$ 
     $\mathcal{I}\text{-ideal-def}$ 
    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** $\langle || \rangle$ 50)

no-notation *plus-oracle* (**infix** $\langle \oplus_{\mathcal{O}} \rangle$ 500)

12 Diffie-Hellman construction

locale *diffie-hellman* =

auth: *ideal-channel* *id* :: 'grp \Rightarrow '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)

```

    (λrsp. case rsp of
      Inl msg ⇒ case msg of
        auth.Out-Recv gx ⇒
          let k = gx [⌈] y in
            Done (Inl (key.Out-Bob k), CState-Full (y, k))
      | Inr - ⇒ 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) ⇒ 'grp) ⇒ ('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 G);
      y ← sample-uniform (order 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 ⇒ 'grp auth.iadv + 'grp auth.iadv

⇒ (('grp auth.oadv + 'grp auth.oadv) × 'grp astate, key.iadv + 'grp isim, key.oadv + osim) gpv

where

```

  sim-callee s-gpair query =
    (let handle = (λgpair-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

$einit \equiv (False, (EState-Void, \{\}), EState-Void, \{\})$

definition $eleak :: (estate, key.event, 'grp isim, osim) eoracle$

where

$eleak \equiv (\lambda(s-flg, (s-event1, s-shell1), s-event2, s-shell2) \text{ query}.$

$\text{let handle-arg1} = (\lambda s q. \text{case } (s, q) \text{ of } (EState-Store, \text{Some } (Inr \text{ (Inr -)})) \Rightarrow (True, EState-Collect) \mid (s', -) \Rightarrow (False, s')) \text{ in}$

$\text{let handle-arg2} = (\lambda s q D. \text{case } (s, q) \text{ of } (EState-Store, \text{Inr -}) \Rightarrow D \mid - \Rightarrow \text{return-pmf None}) \text{ in}$

$\text{let } (is-ch1, s-event1') = \text{handle-arg1 } s-event1 \text{ (case-sum Some } (\lambda-. \text{None}) \text{ query)} \text{ in}$

$\text{let } (is-ch2, s-event2') = \text{handle-arg1 } s-event2 \text{ (case-sum } (\lambda-. \text{None}) \text{ Some query)} \text{ in}$

$\text{let check-pst1} = is-ch1 \wedge s-event2' \neq EState-Void \wedge \text{auth.Bob} \in s-shell1 \wedge \text{auth.Alice} \in s-shell2 \text{ in}$

$\text{let check-pst2} = is-ch2 \wedge s-event1' \neq EState-Void \wedge \text{auth.Alice} \in s-shell1 \wedge \text{auth.Bob} \in s-shell2 \text{ in}$

$\text{let e-pstfix1} = \text{if check-pst1 then } [key.Event-Shell \text{ key.Bob}] \text{ else } [] \text{ in}$

$\text{let e-pstfix2} = \text{if check-pst2 then } [key.Event-Shell \text{ key.Alice}] \text{ else } [] \text{ in}$

$\text{let e-prefix} = \text{if } \neg s-flg \text{ then } [key.Event-Kernel] \text{ else } [] \text{ in}$

$\text{let } (s-flg', event) = \text{if } is-ch2 \vee is-ch1 \text{ then } (True, e-prefix @ e-pstfix1 @ e-pstfix2) \text{ else } (s-flg, []) \text{ in}$

$\text{let result-base} = \text{return-spmf } ((Out-Simulator, event), s-flg', (s-event1', s-shell1), s-event2', s-shell2) \text{ in}$

$\text{case-sum } (\text{handle-arg2 } s-event1) (\text{handle-arg2 } s-event2) \text{ query result-base}$

fun $etran-base :: (key.party \times key.party \Rightarrow key.party \times key.party)$

$\Rightarrow (estate, \text{auth.event}, \text{key.event list}) \text{ oracle}'$

where

$etran-base \text{ mod-event } (s-flg, (s-event1, s-shell1), s-event2, s-shell2) (\text{auth.Event-Shell party}) =$

$(\text{let party-dual} = \text{auth.case-party } (\text{auth.Bob}) (\text{auth.Alice}) \text{ party in}$

$\text{let epair} = \text{auth.case-party prod.swap id party } (\text{key.Bob}, \text{key.Alice}) \text{ in}$

$\text{let } (s-event-eq, s-event-neq) = \text{auth.case-party prod.swap id party } (s-event1, s-event2) \text{ in}$

$\text{let check} = \text{party-dual} \in s-shell2 \wedge s-event-eq = EState-Collect \wedge s-event-neq \neq EState-Void \text{ in}$

$\text{let event} = \text{if check then } [key.Event-Shell ((fst o \text{mod-event}) \text{ epair})] \text{ else } [] \text{ in}$

$\text{let } s-shell1' = \text{insert party } s-shell1 \text{ in}$

$\text{if party} \in s-shell1 \text{ then}$

return-pmf None

else

$\text{return-spmf } (event, s-flg, (s-event1, s-shell1'), s-event2, s-shell2))$

fun $etran :: (estate, (\text{icnv-alice} + \text{icnv-bob}) + \text{auth.event} + \text{auth.event}, \text{key.event list}) \text{ oracle}'$

where

$etran (s-flg, (EState-Void, s-shell1), s-event2, s-shell2) (\text{Inl } (\text{Inl -})) =$

$(\text{let check} = (s-event2 = EState-Collect \wedge \text{auth.Alice} \in s-shell1 \wedge \text{auth.Bob} \in$

```

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) ⊕O (λ(se, sc) q. do {
  ((out, es), se') ← leak 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-oracle (λs q. return-spmf ((handle q, [q]), s)) (rfunc-usr (parallel-rest
auth1-rest auth2-rest)))

```

primcorec *ideal-rest*

where

$rinit\ ideal\ rest = (einit, rinit\ ideal\ rest\ alt)$
 $| rfunc\ adv\ ideal\ rest = (\lambda s\ q.\ case\ q\ of$
 $\quad Inl\ ql \Rightarrow map\ spmf\ (apfst\ (map\ prod\ Inl\ id))\ (eleak\ \dagger\ s\ ql)$
 $\quad | Inr\ qr \Rightarrow map\ spmf\ (apfst\ (map\ prod\ Inr\ id))\ (translate\ eoracle\ etran\ \dagger(rfunc\ adv\ ideal\ rest\ alt)\ s\ qr))$
 $| rfunc\ usr\ ideal\ rest = translate\ eoracle\ etran\ \dagger(rfunc\ usr\ ideal\ rest\ alt)$

definition *ideal-resource*

where

$ideal\ resource \equiv$
 $(let\ sim = CNV\ sim\ callee\ None\ in$
 $\quad attach\ (((sim\ |=\ 1_C) \odot\ lassocr_C\ |=\ 1_C\ |=\ 1_C)\ (key.\ resource\ ideal\ rest))$

definition *real-resource*

where

$real\ resource \equiv$
 $(let\ dh\ wiring = parallel\ wiring \odot (CNV\ alice\ callee\ CState\ Void\ |=\ CNV$
 $bob\ callee\ CState\ Void) \odot parallel\ wiring \odot (1_C\ |=\ swap_C)\ in$
 $\quad attach\ (((1_C\ |=\ 1_C)\ |=\ rassoc_C \odot (dh\ wiring\ |=\ 1_C)) \odot fused\ wiring)$
 $((auth.\ resource\ auth1\ rest) \parallel (auth.\ resource\ auth2\ rest))$

12.5 Wiring and simplifying the Ideal construction

abbreviation *basic-rest-sinit*

where

$basic\ rest\ sinit \equiv ((((), ()), rinit\ auth1\ rest, rinit\ auth2\ rest)$

primcorec *basic-rest* :: ((unit × unit) × -, -, -, -, -, -, -) rest-scheme

where

$rinit\ basic\ rest = (rinit\ auth1\ rest, rinit\ auth2\ rest)$
 $| rfunc\ adv\ basic\ rest = \dagger(parallel\ eoracle\ (rfunc\ adv\ auth1\ rest)\ (rfunc\ adv\ auth2\ rest))$
 $| rfunc\ usr\ basic\ rest = \dagger(parallel\ eoracle\ (rfunc\ usr\ auth1\ rest)\ (rfunc\ usr\ auth2\ rest))$

definition *ideal-s-core'* :: ('grp astate × -) × - × 'grp key.state

where

$ideal\ s\ core' \equiv ((None, ()), einit, key.PState\ Store, \{\})$

definition *ideal-s-rest'*

where

$ideal\ s\ rest' \equiv basic\ rest\ sinit$

primcorec *ideal-core'*

where

$cpoke\ ideal\ core' = (\lambda(s\ cnv, s\ event, s\ core)\ event.\ do\ \{$
 $\quad (events, s\ event') \leftarrow (etran\ s\ event\ event);$
 $\quad s\ core' \leftarrow foldl\ spmf\ key.\ poke\ (return\ spmf\ s\ core)\ events;$
 $\quad return\ spmf\ (s\ cnv, s\ event', s\ core')\ \}$
 $| cfunc\ adv\ ideal\ core' = (\lambda((s\ sim, -), s\ event\ core)\ q.$
 $\quad map\ spmf$

```

    (λ((out, s-sim'), s-event-core'). (out, (s-sim', ()), s-event-core'))
  (exec-gpv
    (†key.iface-adv ⊕O (λ(se, sc) isim. do {
      ((out, es), se') ← leak 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' = (λs 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-oracle (λs 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*=*leak* **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* \implies
 (*sim* |= *1_C*) \odot *lassocr_C* |= *1_C* |= *1_C* \triangleright *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* \triangleright *RES (fused-resource.fuse ideal-core-alt ideal-rest-alt)*
s (**is** ?*L* \implies ?*R*) **for** *sim s*
proof –
have *fact1*: *I-full*, *I-full* $\oplus_{\mathcal{I}}$ *I-full* \vdash_C *CNV sim-callee s* \surd **for** *s*
apply(*subst WT-converter-of-callee*)
apply (*rule WT-gpv-I-mono*)
apply (*rule WT-gpv-full*)
apply (*rule I-full-le-plus-I*)
apply(*rule order-refl*)
apply(*rule order-refl*)
by (*simp-all add:*)

have *fact2*: (*I-full* $\oplus_{\mathcal{I}}$ (*I-full* $\oplus_{\mathcal{I}}$ *I-full*)) $\oplus_{\mathcal{I}}$ (*I-full* $\oplus_{\mathcal{I}}$ *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'=(\cdot)]*)
apply(*subst eq-resource-on-UNIV-iff[symmetric]*)
apply (*subst eq-resource-on-trans*)
apply (*rule eq-I-attach-on'*)
defer
apply (*rule parallel-converter2-eq-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-refl)
    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 at-
tach-wiring-right-def
      rassoclw-def lassocrw-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 $real-s-core' :: (- \times 'grp\ cstate \times 'grp\ cstate) \times 'grp\ auth.state \times 'grp\ auth.state$

where

$real-s-core' \equiv ((((), CState-Void, CState-Void), (auth.State-Void, \{\}), (auth.State-Void, \{\})))$

definition $real-s-rest'$

where

$real-s-rest' \equiv basic-rest-sinit$

primcorec $real-core' :: ((unit \times -) \times -, -, -, -, -, -) core$

where

$cpoke\ real-core' = (\lambda(s-advusr, s-core)\ event.$

$\ map\ spmf\ (Pair\ s-advusr)\ (parallel-handler\ auth.poke\ auth.poke\ s-core\ event))$

$| cfunc-adv\ real-core' = \dagger(auth.iface-adv\ \ddagger_O\ auth.iface-adv)$

$| cfunc-usr\ real-core' = (\lambda((s-adv, s-usr), s-core)\ iusr.$

$\ let\ handle-req = lsumr \circ map-sum\ id\ (rsuml \circ map-sum\ swap-sum\ id \circ lsumr)$
 $\ \circ\ rsuml\ in$

$\ let\ handle-ret = lsumr \circ (map-sum\ id\ (rsuml \circ (map-sum\ swap-sum\ id \circ lsumr))) \circ rsuml \circ map-sum\ id\ swap-sum\ in$

$\ let\ handle-inp = map-sum\ id\ swap-sum \circ (lsumr \circ map-sum\ id\ (rsuml \circ map-sum\ swap-sum\ id \circ lsumr)) \circ rsuml\ in$

$\ let\ handle-out = apfst\ (lsumr \circ (map-sum\ id\ (rsuml \circ (map-sum\ swap-sum\ id \circ lsumr))) \circ rsuml)\ in$

$\ map\ spmf$

$\ (\lambda((ousr, s-usr'), s-core'). (ousr, (s-adv, s-usr'), s-core'))$

$\ (exec-gpv$

$\ (auth.iface-usr\ \ddagger_O\ auth.iface-usr)$

$\ (map-gpv'$

$\ handle-out\ handle-inp\ handle-ret$

$\ ((alice-callee\ \ddagger_I\ bob-callee)\ s-usr\ (handle-req\ iusr)))$

$\ s-core))$

definition $real-rest' :: ((unit \times unit) \times -, -, -, -, -, -) rest-scheme$

where

$real-rest' \equiv basic-rest$

12.6.1 The real attachment lemma

lemma $attach-real: real-resource = 1_C \mid_{=} rassocl_C \triangleright RES\ (fused-resource.fuse\ real-core'\ real-rest')$ $(real-s-core', real-s-rest')$

proof –

have $att-core: real-core' = attach-core\ 1_I$

```

      (attach-wiring parallel-wiringw
        (attach-wiring-right (parallel-wiringw ∘w (id, id) |w swapw) (alice-callee
‡I bob-callee)))
      (parallel-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-wiringw-def swap-lassocrw-def swapw-def lassocrw-def
rassoclw-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-rest' = attach-rest 1I 1I id (parallel-rest auth1-rest auth2-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:
    (I-full ⊕I I-full) ⊕I (I-full ⊕I I-full), (I-full ⊕I I-full) ⊕I (I-full ⊕I I-full)
‡C
    CNV (alice-callee ‡I bob-callee) (CState-Void, CState-Void) √
  apply(subst conv-callee-parallel)
  apply(rule WT-intro)

```

apply (rule *WT-converter-of-callee*[**where** $\mathcal{I}=\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}$ **and** $\mathcal{I}'=\mathcal{I}\text{-full}$
 $\oplus_{\mathcal{I}} \mathcal{I}\text{-full}$])
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** $\mathcal{I}=\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}$ **and** $\mathcal{I}'=\mathcal{I}\text{-full}$
 $\oplus_{\mathcal{I}} \mathcal{I}\text{-full}$])
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:
 $(\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}), (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})$
 \vdash_C
 CNV (*alice-callee* $\ddagger_{\mathcal{I}}$ *bob-callee*) (*CState-Void*, *CState-Void*) \odot *parallel-wiring*
 \odot ($1_C \models \text{swap}_C$) \surd
apply(rule *WT-intro*)
apply (rule *fact1*)
apply(rule *WT-intro*)+
done

have fact3:
 $(\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}), (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})$
 \vdash_C
 CNV (*alice-callee* $\ddagger_{\mathcal{I}}$ *bob-callee*) (*CState-Void*, *CState-Void*) \odot *parallel-wiring*
 \odot ($1_C \models \text{swap}_C$) \sim
 CNV (*attach-wiring-right* (*parallel-wiring* $_w$ \circ_w (*id*, *id*) $|_w$ *swap* $_w$) (*alice-callee*
 $\ddagger_{\mathcal{I}}$ *bob-callee*)) (*CState-Void*, *CState-Void*)
apply (rule *comp-converter-of-callee-wiring*)
apply(rule *wiring-intro*)+
apply(rule *fact1*)
done

have fact4:
 $(\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}), (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})$

```

⊢C
  parallel-wiring ⊙ CNV (alice-callee ‡I bob-callee) (CState-Void, CState-Void)
⊙ parallel-wiring ⊙ (1C |= swapC) ~
  CNV (attach-wiring parallel-wiringw (attach-wiring-right (parallel-wiringw ◦w
(id, id) |w swapw) (alice-callee ‡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-refl)
  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  $\mathcal{I} = ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})) \oplus_{\mathcal{I}} ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}))$ ])
  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-refl)
    apply (rule WT-intro)+
    apply (rule parallel-converter2-eq- $\mathcal{I}$ -cong)

```

```

    apply(rule eq- $\mathcal{I}$ -converter-refl)
    apply(rule WT-intro)+
    apply(rule parallel-converter2-eq- $\mathcal{I}$ -cong)
    prefer 2
    apply(rule eq- $\mathcal{I}$ -converter-refl)
    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  $\mathcal{I}=(\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (((\mathcal{I}\text{-full}
\oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full})) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}))$ ])
    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 ≡ (λstate query.
      let handle = λ((sk, (sc1, sc2)), sa1, sa2). ((sk, (sc2, sc1)), sa2, sa1) in
      let func = λh-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 = 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) ← sample-triple;

```

$$\begin{aligned} & \text{return-spmf } (gxy, \text{h-state } (gxy, gx, gy)) \} \\ & | \text{Some } (gxy, gx, gy) \Rightarrow \text{return-spmf } (gxy, \text{h-state } (gxy, gx, gy)) \\ & \text{else return-pmf None} \\ & | \text{basic-core-oracle-usr-base } ((\text{Some dhs, CState-Full } -, s\text{-cnv2}), s\text{-auth1, auth.State-Collected,} \\ & \text{parties2}) - = \\ & \quad (\text{case dhs of } (gxy, gx, gy) \Rightarrow \\ & \quad \text{return-spmf } (gxy, (\text{Some dhs, CState-Full } (0, \mathbf{1}), s\text{-cnv2}), s\text{-auth1, auth.State-Collected,} \\ & \text{parties2})) \\ & | \text{basic-core-oracle-usr-base } - - = \text{return-pmf None} \end{aligned}$$

definition *basic-core-oracle-usr* :: $(-, \text{key.iusr-alice} + \text{key.iusr-bob}, -)$ *oracle'*
where

$$\begin{aligned} \text{basic-core-oracle-usr} & \equiv (\lambda \text{state query.} \\ & \text{let handle} = \lambda((sk, (sc1, sc2)), sa1, sa2). ((sk, (sc2, sc1)), sa2, sa1) \text{ in} \\ & \text{let func} = \lambda h\text{-o h-s f s. map-spmf } (\text{map-prod } h\text{-o h-s}) (f (h\text{-s s}) ()) \text{ in} \\ & \text{let func-alc} = \text{func } (\text{Inl o key.Out-Alice}) \text{ id basic-core-oracle-usr-base in} \\ & \text{let func-bob} = \text{func } (\text{Inr o key.Out-Bob}) \text{ handle basic-core-oracle-usr-base in} \\ & \text{case-sum } (\lambda\text{-. func-alc state}) (\lambda\text{-. func-bob state}) \text{ query} \end{aligned}$$

primcorec *basic-core*

where

$$\begin{aligned} \text{cpoke basic-core} & = (\lambda(s\text{-other, s-core}) \text{ event.} \\ & \text{map-spmf } (\text{Pair } s\text{-other}) (\text{parallel-handler auth.poke auth.poke s-core event})) \\ & | \text{cfunc-adv basic-core} = (\lambda((s\text{-key, s-cnv}), s\text{-auth1, s-auth2}) \text{ iadv.} \\ & \quad \text{let handle} = (\lambda \text{sel s-init h-out h-state query.} \\ & \quad \text{map-spmf} \\ & \quad (\lambda(\text{out, (s-key}', s\text{-auth}')). (h\text{-out out, (s-key}', s\text{-cnv}), h\text{-state s-auth}' s\text{-auth1} \\ & \quad \text{s-auth2})) \\ & \quad (\text{basic-core-oracle-adv sel } (s\text{-key, s-init}) \text{ query})) \text{ in} \\ & \quad \text{case-sum } (\text{handle } (\text{Inl } ()) \text{ s-auth1 Inl } (\lambda x y z. (x, z))) (\text{handle } (\text{Inr } ()) \text{ s-auth2} \\ & \quad \text{Inr } (\lambda x y z. (y, x))) \text{ iadv}) \\ & | \text{cfunc-usr basic-core} = \\ & \quad (\text{let handle} = \text{map-sum } (\lambda\text{-. Out-Activation-Alice}) (\lambda\text{-. Out-Activation-Bob}) \text{ in} \\ & \quad \text{basic-core-oracle-usr } \oplus_O (\lambda s q. \text{map-spmf } (\text{Pair } (\text{handle } q)) (\text{basic-core-helper} \\ & \quad s q))) \end{aligned}$$

primcorec *lazy-core*

where

$$\begin{aligned} \text{cpoke lazy-core} & = (\lambda s. \text{case-sum } (\lambda q. \text{basic-core-helper } s q) (\text{cpoke basic-core } s)) \\ & | \text{cfunc-adv lazy-core} = \text{cfunc-adv basic-core} \\ & | \text{cfunc-usr lazy-core} = \text{basic-core-oracle-usr} \end{aligned}$$

definition *lazy-rest*

where

$$\text{lazy-rest} \equiv \text{ideal-rest}'$$

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 = (λx. case sc of ((sk, (sc1, sc2)), sa1, sa2) ⇒ ((x, (sc1, sc2)),
sa1, sa2)) in
  let func-look = (λsel 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 (λa 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 = (λh-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 (λa 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 G);
  y ← sample-uniform (order 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 G);

```

```

y ← sample-uniform (order  $\mathcal{G}$ );
z ← sample-uniform (order  $\mathcal{G}$ );
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 ≡ (λA x y z. bind-spmf (return-spmf (z, x, y)) A)

definition *DH-adversary*

where
DH-adversary D ≡ *DH-adversary-curry* (λxyz.
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 (λx. (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: 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 (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
  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  $s\text{-cnv1 } s\text{-cnv2 } s\text{-krn1 } s\text{-krn2 } s\text{-act1 } s\text{-act2 } s\text{-rest1 } s\text{-rest2}$ 
        apply (simp, case-tac (Inl ()), (None,  $s\text{-krn1}$ ,  $s\text{-act1}$ )) 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  $S\text{-ic-intros}$ )
      subgoal for  $s\text{-cnv1 } s\text{-cnv2 } s\text{-krn1 } s\text{-krn2 } x s\text{-act1 } s\text{-act2 } s\text{-rest1 } s\text{-rest2}$ 
        apply (simp, case-tac (Inl ()), (Some  $x$ ,  $s\text{-krn1}$ ,  $s\text{-act1}$ )) rule:
interceptor-base-look.cases)
        by (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro!:
 $S\text{-ic-intros}$ )
      done
    subgoal for  $q\text{-fedit}$ 
      apply (erule  $S\text{-ic-cases}$ )
      subgoal for  $s\text{-cnv1 } s\text{-cnv2 } s\text{-krn1 } s\text{-krn2 } s\text{-act1 } s\text{-act2 } s\text{-rest1 } s\text{-rest2}$ 
        by (simp, case-tac (Inl ()), (None,  $s\text{-krn1}$ ,  $s\text{-act1}$ ), Inr (Inr  $q\text{-fedit}$ ))
rule: basic-core-oracle-adv.cases)
        (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro!:
trace-eq-simcl.base  $S\text{-ic-intros}$ )
      subgoal for  $s\text{-cnv1 } s\text{-cnv2 } s\text{-krn1 } s\text{-krn2 } x s\text{-act1 } s\text{-act2 } s\text{-rest1 } s\text{-rest2}$ 
        by (simp, case-tac (Inl ()), (Some  $x$ ,  $s\text{-krn1}$ ,  $s\text{-act1}$ ), Inr (Inr  $q\text{-fedit}$ ))
rule: basic-core-oracle-adv.cases)
        (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro!:
trace-eq-simcl.base  $S\text{-ic-intros}$ )
      done
    done
  done
done
subgoal for  $q\text{-acore2}$ 
  apply (cases  $q\text{-acore2}$ )
  subgoal for  $q\text{-drop}$  by (erule  $S\text{-ic-cases}$ ) simp-all
  subgoal for  $q\text{-lfe}$ 
    apply (cases  $q\text{-lfe}$ )
    subgoal for  $q\text{-look}$ 
      apply (erule  $S\text{-ic-cases}$ )
      subgoal for  $s\text{-cnv1 } s\text{-cnv2 } s\text{-krn1 } s\text{-krn2 } s\text{-act1 } s\text{-act2 } s\text{-rest1 } s\text{-rest2}$ 
        apply (simp, case-tac (Inr ()), (None,  $s\text{-krn2}$ ,  $s\text{-act2}$ )) 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  $S\text{-ic-intros}$ )
      subgoal for  $s\text{-cnv1 } s\text{-cnv2 } s\text{-krn1 } s\text{-krn2 } x s\text{-act1 } s\text{-act2 } s\text{-rest1 } s\text{-rest2}$ 
        apply (simp, case-tac (Inr ()), (Some  $x$ ,  $s\text{-krn2}$ ,  $s\text{-act2}$ )) rule:
interceptor-base-look.cases)
        by (auto simp add: map-spmf-conv-bind-spmf[symmetric] intro!:
 $S\text{-ic-intros}$ )
      done
    subgoal for  $q\text{-fedit}$ 
      apply (erule  $S\text{-ic-cases}$ )
      subgoal for  $s\text{-cnv1 } s\text{-cnv2 } s\text{-krn1 } s\text{-krn2 } s\text{-act1 } s\text{-act2 } s\text{-rest1 } s\text{-rest2}$ 

```

```

      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= $\lambda(-, (-, x), -). x$ 
      and  $p=\lambda(a, (b, -), d). (a, b, d)$  and  $f=\lambda(a, b, d) c. (a, (b, c), d)$ ,
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= $\lambda(-, (-, x), -). x$ 
      and  $p=\lambda(a, (b, -), d). (a, b, d)$  and  $f=\lambda(a, b, d) c. (a, (b, c), d)$ ,
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= $\lambda(-, -, -, x). x$ 
      and p= $\lambda(a, b, c, -). (a, b, c)$  and f= $\lambda(a, b, c) d. (a, b, c, d)$ ,
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= $\lambda(-, -, -, x). x$ 
      and p= $\lambda(a, b, c, -). (a, b, c)$  and f= $\lambda(a, b, c) d. (a, b, c, d)$ ,
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= $\lambda(-, (-, x), -)$ . x
and p= $\lambda(a, (b, -), d)$ . (a, b, d) and f= $\lambda(a, b, d) c$ . (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= $\lambda(-, (-, x), -)$ . x
and p= $\lambda(a, (b, -), d)$ . (a, b, d) and f= $\lambda(a, b, d) c$ . (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= $\lambda(-, -, -, x)$ . x
and p= $\lambda(a, b, c, -)$ . (a, b, c) and f= $\lambda(a, b, c) d$ . (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= $\lambda(-, -, -, x). x$ 
      and p= $\lambda(a, b, c, -). (a, b, c)$  and f= $\lambda(a, b, c) d. (a, b, c, d)$ ,
simplified])
      by (clarsimp simp add: map-spmf-conv-bind-spmf[symmetric] intro!:
S-ic-intros)
      done
      done
      done
      done
      done
      done
      done

then show ?thesis by simp
qed

```

private abbreviation *dummy* $x \equiv \text{TRY } \text{map-spmf } \text{OK } x \text{ ELSE } \text{return-spmf } \text{Fault}$

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)  $A = \text{do } \{
  x \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});
  y \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});
  (DH\text{-adversary-curry } A) (\mathbf{g} [\wedge] x) (\mathbf{g} [\wedge] y) (\mathbf{g} [\wedge] (x * y))
\} \text{ for } A \text{ by } (\text{simp add: } DH0\text{-sample-def } DH\text{-adversary-curry-def } \text{nat-pow-pow})$ 
```

```

have fact2: bind-spmf  $DH1$ -sample  $A = \text{do } \{
  x \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});
  y \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});
  z \leftarrow \text{sample-uniform } (\text{order } \mathcal{G});
  (DH\text{-adversary-curry } A) (\mathbf{g} [\wedge] x) (\mathbf{g} [\wedge] y) (\mathbf{g} [\wedge] (z))
\} \text{ for } A \text{ by } (\text{simp add: } DH1\text{-sample-def } DH\text{-adversary-curry-def})$ 
```

```

{
  fix sample-triple :: ('grp  $\times$  'grp  $\times$  'grp) spmf
  assume *: lossless-spmf sample-triple
  define s-init where s-init  $\equiv$  (basic-core-sinit, basic-rest-sinit)
  have [unfolded s-init-def, simp]: dummy (dummy (return-spmf s-init)) = re-
turn-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  $\circ$  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  $\mathcal{I}=\mathcal{I}\text{-full}$  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* \equiv *sample-uniform* (order \mathcal{G})

private abbreviation *isample-key* \equiv *spmf-of-set* (carrier \mathcal{G})

private abbreviation *isample-pair-nn* \equiv *pair-spmf* *isample-nat* *isample-nat*

private abbreviation *isample-pair-nk* \equiv *pair-spmf* *isample-nat* *isample-key*

private inductive *S-il* :: (('grp *astate* \times *unit*) \times *estate* \times 'grp *key.state*) *spmf* \Rightarrow 'grp *bc-state* *spmf* \Rightarrow *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 \in s-act1
 | sil-2-0: S-il (map-spmf ($\lambda k.$ ((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 \in s-act1
 — ../(Auth1 =a)@(Auth2 =0) # look
 | sil-1'-0: S-il (map-spmf ($\lambda y.$ ((Some (g [\checkmark] x, g [\checkmark] y), ()), (False, (EState-Store,
 s-act1), EState-Void, s-act2), key.PState-Store, { $\}$)) isample-nat)
 (map-spmf ($\lambda yz.$ ((Some (g [\checkmark] snd yz, g [\checkmark] (x :: nat), g [\checkmark] fst yz),
 CState-Half 0, CState-Void), (auth.State-Store **1**, s-act1), auth.State-Void, s-act2))
 isample-pair-nn)
if auth.Alice \in s-act1
 | sil-2'-0: S-il (map-spmf ($\lambda yk.$ ((Some (g [\checkmark] x, g [\checkmark] fst yk), ()), (True,
 (EState-Collect, s-act1), EState-Void, s-act2), key.State-Store (snd yk), { $\}$)) isam-
 ple-pair-nk)
 (map-spmf ($\lambda yz.$ ((Some (g [\checkmark] snd yz, g [\checkmark] (x :: nat), g [\checkmark] fst yz), CState-Half
 0, CState-Void), (auth.State-Collect **1**, s-act1), auth.State-Void, s-act2)) isam-
 ple-pair-nn)
if auth.Alice \in 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 \in 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 \in s-act1 **and** auth.Alice \in s-act2
 | sil-2-1: S-il (map-spmf ($\lambda k.$ ((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 \in s-act1 **and** auth.Alice \in s-act2 **and** key.Alice \notin s-actk **and**
 auth.Bob \in s-act1 \longleftrightarrow key.Bob \in 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 ($\lambda xy.$ ((Some (g [\checkmark] (z :: nat), g [\checkmark] fst xy, g [\checkmark] 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 \in s-act1 **and** auth.Alice \in s-act2 **and** key.Alice \notin s-actk **and**
 auth.Bob \in s-act1 **and** key.Bob \in s-actk **and** k = g [\checkmark] z
 — ../(Auth1 =a)@(Auth2 =1) # look
 | sil-1c-1c: S-il (return-spmf ((Some (g [\checkmark] x, g [\checkmark] y), ()), (False, (EState-Store,

$s\text{-act1}$), $E\text{State-Store}$, $s\text{-act2}$), $key.P\text{State-Store}$, $\{\}$)
 $(\text{map-spmf } (\lambda z. ((\text{Some } (\mathbf{g} [\] z, \mathbf{g} [\] (x :: \text{nat}), \mathbf{g} [\] (y :: \text{nat})), C\text{State-Half } 0, C\text{State-Half } 0), (\text{auth.State-Store } \mathbf{1}, s\text{-act1}), \text{auth.State-Store } \mathbf{1}, s\text{-act2}))) \text{ isample-nat}$
if $\text{auth.Alice} \in s\text{-act1}$ **and** $\text{auth.Alice} \in s\text{-act2}$
 $| \text{sil-2c-1c: } S\text{-il } (\text{return-spmf } ((\text{Some } (\mathbf{g} [\] x, \mathbf{g} [\] y), ()), (\text{True}, (E\text{State-Collect}, s\text{-act1}), E\text{State-Store}, s\text{-act2}), \text{key.State-Store } k, s\text{-actk}))$
 $(\text{return-spmf } ((\text{Some } (\mathbf{g} [\] z, \mathbf{g} [\] (x :: \text{nat}), \mathbf{g} [\] (y :: \text{nat})), C\text{State-Half } 0, C\text{State-Half } 0), (\text{auth.State-Collect } \mathbf{1}, s\text{-act1}), \text{auth.State-Store } \mathbf{1}, s\text{-act2}))$
if $\text{auth.Alice} \in s\text{-act1}$ **and** $\text{auth.Alice} \in s\text{-act2}$ **and** $\text{key.Alice} \notin s\text{-actk}$ **and** $\text{auth.Bob} \in s\text{-act1} \longleftrightarrow \text{key.Bob} \in s\text{-actk}$ **and** $k = \mathbf{g} [\] z$ **and** $z \in \text{set-spmf isample-nat}$
 $| \text{sil-3c-1c: } S\text{-il } (\text{return-spmf } ((\text{Some } (\mathbf{g} [\] x, \mathbf{g} [\] y), ()), (\text{True}, (E\text{State-Collect}, s\text{-act1}), E\text{State-Store}, s\text{-act2}), \text{key.State-Store } k, s\text{-actk}))$
 $(\text{return-spmf } ((\text{Some } (\mathbf{g} [\] (z :: \text{nat}), \mathbf{g} [\] (x :: \text{nat}), \mathbf{g} [\] (y :: \text{nat})), C\text{State-Half } 0, C\text{State-Full } (0, \mathbf{1})), (\text{auth.State-Collected}, s\text{-act1}), \text{auth.State-Store } \mathbf{1}, s\text{-act2}))$
if $\text{auth.Alice} \in s\text{-act1}$ **and** $\text{auth.Alice} \in s\text{-act2}$ **and** $\text{key.Alice} \notin s\text{-actk}$ **and** $\text{auth.Bob} \in s\text{-act1}$ **and** $\text{key.Bob} \in s\text{-actk}$ **and** $k = \mathbf{g} [\] z$
 $— (\text{Auth1} = a) @ (\text{Auth2} = 2)$
 $| \text{sil-0-2: } S\text{-il } (\text{map-spmf } (\lambda k. ((\text{None}, ()), (\text{True}, (E\text{State-Void}, s\text{-act1}), E\text{State-Collect}, s\text{-act2}), \text{key.State-Store } k, \{\}))) \text{ isample-key}$
 $(\text{return-spmf } ((\text{None}, C\text{State-Void}, C\text{State-Half } 0), (\text{auth.State-Void}, s\text{-act1}), \text{auth.State-Collect } \mathbf{1}, s\text{-act2}))$
if $\text{auth.Alice} \in s\text{-act2}$
 $— .. / (\text{Auth1} = a) @ (\text{Auth2} = 2) \# \text{wl}$
 $| \text{sil-1-2: } S\text{-il } (\text{map-spmf } (\lambda k. ((\text{None}, ()), (\text{True}, (E\text{State-Store}, s\text{-act1}), E\text{State-Collect}, s\text{-act2}), \text{key.State-Store } k, s\text{-actk}))) \text{ isample-key}$
 $(\text{return-spmf } ((\text{None}, C\text{State-Half } 0, C\text{State-Half } 0), (\text{auth.State-Store } \mathbf{1}, s\text{-act1}), \text{auth.State-Collect } \mathbf{1}, s\text{-act2}))$
if $\text{auth.Alice} \in s\text{-act1}$ **and** $\text{auth.Alice} \in s\text{-act2}$ **and** $\text{auth.Bob} \in s\text{-act2} \longleftrightarrow \text{key.Alice} \in s\text{-actk}$ **and** $\text{key.Bob} \notin s\text{-actk}$
 $| \text{sil-2-2: } S\text{-il } (\text{map-spmf } (\lambda k. ((\text{None}, ()), (\text{True}, (E\text{State-Collect}, s\text{-act1}), E\text{State-Collect}, s\text{-act2}), \text{key.State-Store } k, s\text{-actk}))) \text{ isample-key}$
 $(\text{return-spmf } ((\text{None}, C\text{State-Half } 0, C\text{State-Half } 0), (\text{auth.State-Collect } \mathbf{1}, s\text{-act1}), \text{auth.State-Collect } \mathbf{1}, s\text{-act2}))$
if $\text{auth.Alice} \in s\text{-act1}$ **and** $\text{auth.Alice} \in s\text{-act2}$ **and** $\text{auth.Bob} \in s\text{-act2} \longleftrightarrow \text{key.Alice} \in s\text{-actk}$ **and** $\text{auth.Bob} \in s\text{-act1} \longleftrightarrow \text{key.Bob} \in s\text{-actk}$
 $| \text{sil-3-2: } S\text{-il } (\text{return-spmf } ((\text{None}, ()), (\text{True}, (E\text{State-Collect}, s\text{-act1}), E\text{State-Collect}, s\text{-act2}), \text{key.State-Store } k, s\text{-actk})))$
 $(\text{map-spmf } (\lambda xy. ((\text{Some } (\mathbf{g} [\] (z :: \text{nat}), \mathbf{g} [\] \text{fst } xy, \mathbf{g} [\] \text{snd } xy), C\text{State-Half } 0, C\text{State-Full } (0, \mathbf{1})), (\text{auth.State-Collected}, s\text{-act1}), \text{auth.State-Collect } \mathbf{1}, s\text{-act2}))) \text{ isample-pair-nn}$
if $\text{auth.Alice} \in s\text{-act1}$ **and** $\text{auth.Alice} \in s\text{-act2}$ **and** $\text{auth.Bob} \in s\text{-act2} \longleftrightarrow \text{key.Alice} \in s\text{-actk}$ **and** $\text{auth.Bob} \in s\text{-act1}$ **and** $\text{key.Bob} \in s\text{-actk}$ **and** $k = \mathbf{g} [\] z$
 $— .. / (\text{Auth1} = a) @ (\text{Auth2} = 2) \# \text{look}$
 $| \text{sil-1c-2c: } S\text{-il } (\text{return-spmf } ((\text{Some } (\mathbf{g} [\] x, \mathbf{g} [\] y), ()), (\text{True}, (E\text{State-Store}, s\text{-act1}), E\text{State-Collect}, s\text{-act2}), \text{key.State-Store } k, s\text{-actk})))$
 $(\text{return-spmf } ((\text{Some } (\mathbf{g} [\] z, \mathbf{g} [\] (x :: \text{nat}), \mathbf{g} [\] (y :: \text{nat})), C\text{State-Half } 0,$

CState-Half 0), (*auth.State-Store 1*, *s-act1*), *auth.State-Collect 1*, *s-act2*)
if *auth.Alice* \in *s-act1* **and** *auth.Alice* \in *s-act2* **and** *auth.Bob* \in *s-act2* \longleftrightarrow
key.Alice \in *s-actk* **and** *key.Bob* \notin *s-actk* **and** $k = \mathbf{g} [\uparrow] z$ **and** $z \in \text{set-spmf}$
isample-nat
| *sil-2c-2c*: *S-il* (*return-spmf* ((*Some* ($\mathbf{g} [\uparrow] x$, $\mathbf{g} [\uparrow] y$), ()), (*True*, (*EState-Collect*,
s-act1), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* ($\mathbf{g} [\uparrow] z$, $\mathbf{g} [\uparrow] (x :: \text{nat})$), $\mathbf{g} [\uparrow] (y :: \text{nat})$), *CState-Half 0*,
CState-Half 0), (*auth.State-Collect 1*, *s-act1*), *auth.State-Collect 1*, *s-act2*))
if *auth.Alice* \in *s-act1* **and** *auth.Alice* \in *s-act2* **and** *auth.Bob* \in *s-act2* \longleftrightarrow
key.Alice \in *s-actk* **and** *auth.Bob* \in *s-act1* \longleftrightarrow *key.Bob* \in *s-actk* **and** $k = \mathbf{g} [\uparrow] z$
and $z \in \text{set-spmf}$ *isample-nat*
| *sil-3c-2c*: *S-il* (*return-spmf* ((*Some* ($\mathbf{g} [\uparrow] x$, $\mathbf{g} [\uparrow] y$), ()), (*True*, (*EState-Collect*,
s-act1), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* ($\mathbf{g} [\uparrow] (z :: \text{nat})$), $\mathbf{g} [\uparrow] (x :: \text{nat})$), $\mathbf{g} [\uparrow] (y :: \text{nat})$),
CState-Half 0, *CState-Full (0, 1)*), (*auth.State-Collected*, *s-act1*), *auth.State-Collect*
1, *s-act2*))
if *auth.Alice* \in *s-act1* **and** *auth.Alice* \in *s-act2* **and** *auth.Bob* \in *s-act2* \longleftrightarrow
key.Alice \in *s-actk* **and** *auth.Bob* \in *s-act1* **and** *key.Bob* \in *s-actk* **and** $k = \mathbf{g} [\uparrow] z$
— (*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* ($\lambda xy. ((\text{Some } (\mathbf{g} [\uparrow] (z :: \text{nat})), \mathbf{g} [\uparrow] \text{fst } xy, \mathbf{g} [\uparrow] \text{snd } xy)$), *CState-Full*
(*0, 1*), *CState-Half 0*), (*auth.State-Store 1*, *s-act1*), *auth.State-Collected*, *s-act2*))
isample-pair-nn)
if *auth.Alice* \in *s-act1* **and** *auth.Alice* \in *s-act2* **and** *auth.Bob* \in *s-act2* **and**
key.Alice \in *s-actk* **and** *key.Bob* \notin *s-actk* **and** $k = \mathbf{g} [\uparrow] z$
| *sil-2-3*: *S-il* (*return-spmf* ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *ES-*
tate-Collect, *s-act2*), *key.State-Store k*, *s-actk*))
(*map-spmf* ($\lambda xy. ((\text{Some } (\mathbf{g} [\uparrow] (z :: \text{nat})), \mathbf{g} [\uparrow] \text{fst } xy, \mathbf{g} [\uparrow] \text{snd } xy)$), *CState-Full*
(*0, 1*), *CState-Half 0*), (*auth.State-Collect 1*, *s-act1*), *auth.State-Collected*, *s-act2*))
isample-pair-nn)
if *auth.Alice* \in *s-act1* **and** *auth.Alice* \in *s-act2* **and** *auth.Bob* \in *s-act2* **and**
key.Alice \in *s-actk* **and** *auth.Bob* \in *s-act1* \longleftrightarrow *key.Bob* \in *s-actk* **and** $k = \mathbf{g} [\uparrow] z$
| *sil-3-3*: *S-il* (*return-spmf* ((*None*, ()), (*True*, (*EState-Collect*, *s-act1*), *ES-*
tate-Collect, *s-act2*), *key.State-Store k*, *s-actk*))
(*map-spmf* ($\lambda xy. ((\text{Some } (\mathbf{g} [\uparrow] (z :: \text{nat})), \mathbf{g} [\uparrow] \text{fst } xy, \mathbf{g} [\uparrow] \text{snd } xy)$), *CState-Full*
(*0, 1*), *CState-Full (0, 1)*), (*auth.State-Collected*, *s-act1*), *auth.State-Collected*,
s-act2)) *isample-pair-nn*)
if *auth.Alice* \in *s-act1* **and** *auth.Alice* \in *s-act2* **and** *auth.Bob* \in *s-act2* **and**
key.Alice \in *s-actk* **and** *auth.Bob* \in *s-act1* **and** *key.Bob* \in *s-actk* **and** $k = \mathbf{g} [\uparrow] z$
— ../(*Auth1 = a*)@(*Auth2 = 3*) # *look*
| *sil-1c-3c*: *S-il* (*return-spmf* ((*Some* ($\mathbf{g} [\uparrow] x$, $\mathbf{g} [\uparrow] y$), ()), (*True*, (*EState-Store*,
s-act1), *EState-Collect*, *s-act2*), *key.State-Store k*, *s-actk*))
(*return-spmf* ((*Some* ($\mathbf{g} [\uparrow] (z :: \text{nat})$), $\mathbf{g} [\uparrow] (x :: \text{nat})$), $\mathbf{g} [\uparrow] (y :: \text{nat})$), *CState-Full*
(*0, 1*), *CState-Half 0*), (*auth.State-Store 1*, *s-act1*), *auth.State-Collected*, *s-act2*))
if *auth.Alice* \in *s-act1* **and** *auth.Alice* \in *s-act2* **and** *auth.Bob* \in *s-act2* **and**
key.Alice \in *s-actk* **and** *key.Bob* \notin *s-actk* **and** $k = \mathbf{g} [\uparrow] z$
| *sil-2c-3c*: *S-il* (*return-spmf* ((*Some* ($\mathbf{g} [\uparrow] x$, $\mathbf{g} [\uparrow] y$), ()), (*True*, (*EState-Collect*,

$s\text{-act1}$), $E\text{State-Collect}$, $s\text{-act2}$), $key.State\text{-Store } k$, $s\text{-actk}$)
 (return-spmf ((Some (g [∧] (z :: nat)), g [∧] (x :: nat), g [∧] (y :: nat)), $C\text{State-Full}$
 (0, 1), $C\text{State-Half } 0$), (auth.State-Collect 1, $s\text{-act1}$), auth.State-Collected, $s\text{-act2}$))
 if auth.Alice ∈ $s\text{-act1}$ and auth.Alice ∈ $s\text{-act2}$ and auth.Bob ∈ $s\text{-act2}$ and
 key.Alice ∈ $s\text{-actk}$ and auth.Bob ∈ $s\text{-act1} \longleftrightarrow key.Bob \in s\text{-actk}$ and $k = \mathbf{g} [\wedge] z$
 | sil-3c-3c: S-il (return-spmf ((Some (g [∧] x, g [∧] y), ()), (True, (EState-Collect,
 $s\text{-act1}$), $E\text{State-Collect}$, $s\text{-act2}$), key.State-Store k, $s\text{-actk}$))
 (return-spmf ((Some (g [∧] (z :: nat)), g [∧] (x :: nat), g [∧] (y :: nat)), $C\text{State-Full}$
 (0, 1), $C\text{State-Full } (0, 1)$), (auth.State-Collected, $s\text{-act1}$), auth.State-Collected,
 $s\text{-act2}$))
 if auth.Alice ∈ $s\text{-act1}$ and auth.Alice ∈ $s\text{-act2}$ and auth.Bob ∈ $s\text{-act2}$ and
 key.Alice ∈ $s\text{-actk}$ and auth.Bob ∈ $s\text{-act1}$ and key.Bob ∈ $s\text{-actk}$ and $k = \mathbf{g} [\wedge] z$
 — (Auth1 = a)@(Auth2 = 1')
 | sil-0-1': S-il (map-spmf (λx. ((Some (g [∧] x, g [∧] y), ()), (False, (EState-Void,
 $s\text{-act1}$), $E\text{State-Store}$, $s\text{-act2}$), key.PState-Store, { }))) isample-nat
 (map-spmf (λxz. ((Some (g [∧] snd xz, g [∧] fst xz, g [∧] (y :: nat)),
 $C\text{State-Void}$, $C\text{State-Half } 0$), (auth.State-Void, $s\text{-act1}$), auth.State-Store 1, $s\text{-act2}$))
 isample-pair-nn)
 if auth.Alice ∈ $s\text{-act2}$
 — (Auth1 = a)@(Auth2 = 2')
 | sil-0-2': S-il (map-spmf (λxk. ((Some (g [∧] fst xk, g [∧] y), ()), (True,
 (EState-Void, $s\text{-act1}$), $E\text{State-Collect}$, $s\text{-act2}$), key.State-Store (snd xk), { }))) isam-
 ple-pair-nk)
 (map-spmf (λxz. ((Some (g [∧] snd xz, g [∧] fst xz, g [∧] (y :: nat)), $C\text{State-Void}$,
 $C\text{State-Half } 0$), (auth.State-Void, $s\text{-act1}$), auth.State-Collect 1, $s\text{-act2}$)) isample-pair-nn)
 if auth.Alice ∈ $s\text{-act2}$

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 (λx. 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 leak-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
    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* \equiv *sample-uniform* (order \mathcal{G})

private abbreviation *rsample-pair-nn* \equiv *pair-spmf* *rsample-nat* *rsample-nat*

private inductive *S-rl* :: ((unit \times 'grp *cstate* \times 'grp *cstate*) \times 'grp *auth.state* \times 'grp *auth.state*) *spmf*

\Rightarrow (('grp *st-state* \times 'grp *cstate* \times 'grp *cstate*) \times 'grp *auth.state* \times 'grp *auth.state*) *spmf* \Rightarrow *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 (λx . ((((), *CState-Half* *x*, *CState-Void*), (*auth.State-Store* (\mathbf{g} [\uparrow] *x*), *s-act1*), *auth.State-Void*, *s-act2*))) *rsample-nat*)

(return-spmf ((None, *CState-Half* 0, *CState-Void*), (*auth.State-Store* $\mathbf{1}$, *s-act1*), *auth.State-Void*, *s-act2*)))

| *srl-2-0*: *S-rl* (map-spmf (λx . ((((), *CState-Half* *x*, *CState-Void*), (*auth.State-Collect* (\mathbf{g} [\uparrow] *x*), *s-act1*), *auth.State-Void*, *s-act2*))) *rsample-nat*)

(return-spmf ((None, *CState-Half* 0, *CState-Void*), (*auth.State-Collect* $\mathbf{1}$, *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* (\mathbf{g} [\uparrow] *x*), *s-act1*), *auth.State-Void*, *s-act2*)))

(map-spmf (λy . ((Some ((\mathbf{g} [\uparrow] *x*) [\uparrow] *y*, \mathbf{g} [\uparrow] *x*, \mathbf{g} [\uparrow] *y*), *CState-Half* 0, *CState-Void*), (*auth.State-Store* $\mathbf{1}$, *s-act1*), *auth.State-Void*, *s-act2*))) *rsample-nat*)

| *srl-2'-0*: *S-rl* (return-spmf ((((), *CState-Half* *x*, *CState-Void*), (*auth.State-Collect* (\mathbf{g} [\uparrow] *x*), *s-act1*), *auth.State-Void*, *s-act2*)))

(map-spmf (λy . ((Some ((\mathbf{g} [\uparrow] *x*) [\uparrow] *y*, \mathbf{g} [\uparrow] *x*, \mathbf{g} [\uparrow] *y*), *CState-Half* 0, *CState-Void*), (*auth.State-Collect* $\mathbf{1}$, *s-act1*), *auth.State-Void*, *s-act2*))) *rsample-nat*)

— (Auth1 =a)@(Auth2 =1)

| *srl-0-1*: *S-rl* (map-spmf (λy . ((((), *CState-Void*, *CState-Half* *y*), (*auth.State-Void*, *s-act1*), *auth.State-Store* (\mathbf{g} [\uparrow] *y*), *s-act2*))) *rsample-nat*)

(return-spmf ((None, *CState-Void*, *CState-Half* 0), (*auth.State-Void*, *s-act1*), *auth.State-Store* $\mathbf{1}$, *s-act2*)))

— ../(Auth1 =a)@(Auth2 =1) # wl

| *srl-1-1*: *S-rl* (map-spmf (λyx . ((((), *CState-Half* (*snd* *yx*), *CState-Half* (*fst* *yx*)),

$(\text{auth.State-Store } (\mathbf{g} \ [\] \ \text{snd } yx), \text{ s-act1}), \text{ auth.State-Store } (\mathbf{g} \ [\] \ \text{fst } yx), \text{ s-act2}))$
rsample-pair-nn
 $(\text{return-spmf } ((\text{None}, \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Store } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Store } \mathbf{1}, \text{ s-act2}))$
 $| \text{ srl-2-1: S-rl } (\text{map-spmf } (\lambda yx. (((), \text{CState-Half } (\text{snd } yx), \text{CState-Half } (\text{fst } yx)), (\text{auth.State-Collect } (\mathbf{g} \ [\] \ \text{snd } yx), \text{ s-act1}), \text{ auth.State-Store } (\mathbf{g} \ [\] \ \text{fst } yx), \text{ s-act2})))$
rsample-pair-nn
 $(\text{return-spmf } ((\text{None}, \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Collect } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Store } \mathbf{1}, \text{ s-act2}))$
 $— \text{../}(\text{Auth1} = \mathbf{a})@(\text{Auth2} = \mathbf{1}) \# \text{ look}$
 $| \text{ srl-1c-1c: S-rl } (\text{return-spmf } (((), \text{CState-Half } x, \text{CState-Half } y), (\text{auth.State-Store } (\mathbf{g} \ [\] \ x), \text{ s-act1}), \text{ auth.State-Store } (\mathbf{g} \ [\] \ y), \text{ s-act2}))$
 $(\text{return-spmf } ((\text{Some } ((\mathbf{g} \ [\] \ x) \ [\] \ y, \mathbf{g} \ [\] \ x, \mathbf{g} \ [\] \ y), \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Store } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Store } \mathbf{1}, \text{ s-act2}))$
 $| \text{ srl-2c-1c: S-rl } (\text{return-spmf } (((), \text{CState-Half } x, \text{CState-Half } y), (\text{auth.State-Collect } (\mathbf{g} \ [\] \ x), \text{ s-act1}), \text{ auth.State-Store } (\mathbf{g} \ [\] \ y), \text{ s-act2}))$
 $(\text{return-spmf } ((\text{Some } ((\mathbf{g} \ [\] \ x) \ [\] \ y, \mathbf{g} \ [\] \ x, \mathbf{g} \ [\] \ y), \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Collect } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Store } \mathbf{1}, \text{ s-act2}))$
 $| \text{ srl-3c-1c: S-rl } (\text{return-spmf } (((), \text{CState-Half } x, \text{CState-Full } (y, z)), (\text{auth.State-Collected}, \text{ s-act1}), \text{ auth.State-Store } (\mathbf{g} \ [\] \ y), \text{ s-act2}))$
 $(\text{return-spmf } ((\text{Some } (z, \mathbf{g} \ [\] \ x, \mathbf{g} \ [\] \ y), \text{CState-Half } 0, \text{CState-Full } (0, \mathbf{1})), (\text{auth.State-Collected}, \text{ s-act1}), \text{ auth.State-Store } \mathbf{1}, \text{ s-act2}))$
 $\text{if } z = (\mathbf{g} \ [\] \ x) \ [\] \ y$
 $— (\text{Auth1} = \mathbf{a})@(\text{Auth2} = \mathbf{2})$
 $| \text{ srl-0-2: S-rl } (\text{map-spmf } (\lambda y. (((), \text{CState-Void}, \text{CState-Half } y), (\text{auth.State-Void}, \text{ s-act1}), \text{ auth.State-Collect } (\mathbf{g} \ [\] \ y), \text{ s-act2}))) \text{ rsample-nat}$
 $(\text{return-spmf } ((\text{None}, \text{CState-Void}, \text{CState-Half } 0), (\text{auth.State-Void}, \text{ s-act1}), \text{ auth.State-Collect } \mathbf{1}, \text{ s-act2}))$
 $— \text{../}(\text{Auth1} = \mathbf{a})@(\text{Auth2} = \mathbf{2}) \# \text{ wl}$
 $| \text{ srl-1-2: S-rl } (\text{map-spmf } (\lambda yx. (((), \text{CState-Half } (\text{snd } yx), \text{CState-Half } (\text{fst } yx)), (\text{auth.State-Store } (\mathbf{g} \ [\] \ \text{snd } yx), \text{ s-act1}), \text{ auth.State-Collect } (\mathbf{g} \ [\] \ \text{fst } yx), \text{ s-act2})))$
rsample-pair-nn
 $(\text{return-spmf } ((\text{None}, \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Store } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Collect } \mathbf{1}, \text{ s-act2}))$
 $| \text{ srl-2-2: S-rl } (\text{map-spmf } (\lambda yx. (((), \text{CState-Half } (\text{snd } yx), \text{CState-Half } (\text{fst } yx)), (\text{auth.State-Collect } (\mathbf{g} \ [\] \ \text{snd } yx), \text{ s-act1}), \text{ auth.State-Collect } (\mathbf{g} \ [\] \ \text{fst } yx), \text{ s-act2})))$
rsample-pair-nn
 $(\text{return-spmf } ((\text{None}, \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Collect } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Collect } \mathbf{1}, \text{ s-act2}))$
 $— \text{../}(\text{Auth1} = \mathbf{a})@(\text{Auth2} = \mathbf{2}) \# \text{ look}$
 $| \text{ srl-1c-2c: S-rl } (\text{return-spmf } (((), \text{CState-Half } x, \text{CState-Half } y), (\text{auth.State-Store } (\mathbf{g} \ [\] \ x), \text{ s-act1}), \text{ auth.State-Collect } (\mathbf{g} \ [\] \ y), \text{ s-act2}))$
 $(\text{return-spmf } ((\text{Some } ((\mathbf{g} \ [\] \ x) \ [\] \ y, \mathbf{g} \ [\] \ x, \mathbf{g} \ [\] \ y), \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Store } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Collect } \mathbf{1}, \text{ s-act2}))$
 $| \text{ srl-2c-2c: S-rl } (\text{return-spmf } (((), \text{CState-Half } x, \text{CState-Half } y), (\text{auth.State-Collect } (\mathbf{g} \ [\] \ x), \text{ s-act1}), \text{ auth.State-Collect } (\mathbf{g} \ [\] \ y), \text{ s-act2}))$
 $(\text{return-spmf } ((\text{Some } ((\mathbf{g} \ [\] \ x) \ [\] \ y, \mathbf{g} \ [\] \ x, \mathbf{g} \ [\] \ y), \text{CState-Half } 0, \text{CState-Half } 0), (\text{auth.State-Collect } \mathbf{1}, \text{ s-act1}), \text{ auth.State-Collect } \mathbf{1}, \text{ s-act2}))$
 $| \text{ srl-3c-2c: S-rl } (\text{return-spmf } (((), \text{CState-Half } x, \text{CState-Full } (y, z)), (\text{auth.State-Collected},$

$s\text{-act1}$), $\text{auth.State-Collect } (\mathbf{g} [\uparrow] y)$, $s\text{-act2}$)
 $(\text{return-spmf } ((\text{Some } (z, \mathbf{g} [\uparrow] x, \mathbf{g} [\uparrow] y)), \text{CState-Half } 0, \text{CState-Full } (0, \mathbf{1})),$
 $(\text{auth.State-Collected}, s\text{-act1}), \text{auth.State-Collect } \mathbf{1}, s\text{-act2}))$
if $z = (\mathbf{g} [\uparrow] x) [\uparrow] y$
 $\text{— } (\text{Auth1} = \mathbf{a}) @ (\text{Auth2} = \mathbf{3})$
 $| \text{srl-1c-3c: } S\text{-rl } (\text{return-spmf } (((), \text{CState-Full } (x, z), \text{CState-Half } y), (\text{auth.State-Store}$
 $(\mathbf{g} [\uparrow] x), s\text{-act1}), \text{auth.State-Collected}, s\text{-act2}))$
 $(\text{return-spmf } ((\text{Some } (z, \mathbf{g} [\uparrow] x, \mathbf{g} [\uparrow] y)), \text{CState-Full } (0, \mathbf{1}), \text{CState-Half } 0),$
 $(\text{auth.State-Store } \mathbf{1}, s\text{-act1}), \text{auth.State-Collected}, s\text{-act2}))$
if $z = (\mathbf{g} [\uparrow] y) [\uparrow] x$
 $| \text{srl-2c-3c: } S\text{-rl } (\text{return-spmf } (((), \text{CState-Full } (x, z), \text{CState-Half } y), (\text{auth.State-Collect}$
 $(\mathbf{g} [\uparrow] x), s\text{-act1}), \text{auth.State-Collected}, s\text{-act2}))$
 $(\text{return-spmf } ((\text{Some } (z, \mathbf{g} [\uparrow] x, \mathbf{g} [\uparrow] y)), \text{CState-Full } (0, \mathbf{1}), \text{CState-Half } 0),$
 $(\text{auth.State-Collect } \mathbf{1}, s\text{-act1}), \text{auth.State-Collected}, s\text{-act2}))$
if $z = (\mathbf{g} [\uparrow] y) [\uparrow] x$
 $| \text{srl-3c-3c: } S\text{-rl } (\text{return-spmf } (((), \text{CState-Full } (x, z), \text{CState-Full } (y, z)), (\text{auth.State-Collected},$
 $s\text{-act1}), \text{auth.State-Collected}, s\text{-act2}))$
 $(\text{return-spmf } ((\text{Some } (z, \mathbf{g} [\uparrow] x, \mathbf{g} [\uparrow] y)), \text{CState-Full } (0, \mathbf{1}), \text{CState-Full } (0,$
 $\mathbf{1})), (\text{auth.State-Collected}, s\text{-act1}), \text{auth.State-Collected}, s\text{-act2}))$
if $z = (\mathbf{g} [\uparrow] y) [\uparrow] x$
 $\text{— } (\text{Auth1} = \mathbf{0}) @ (\text{Auth2} = \mathbf{1}')$
 $| \text{srl-0-1': } S\text{-rl } (\text{return-spmf } (((), \text{CState-Void}, \text{CState-Half } y), (\text{auth.State-Void},$
 $s\text{-act1}), \text{auth.State-Store } (\mathbf{g} [\uparrow] y), s\text{-act2}))$
 $(\text{map-spmf } (\lambda x. ((\text{Some } ((\mathbf{g} [\uparrow] x) [\uparrow] y, \mathbf{g} [\uparrow] x, \mathbf{g} [\uparrow] y)), \text{CState-Void},$
 $\text{CState-Half } 0), (\text{auth.State-Void}, s\text{-act1}), \text{auth.State-Store } \mathbf{1}, s\text{-act2})) \text{rsample-nat})$
 $\text{— } (\text{Auth1} = \mathbf{0}) @ (\text{Auth2} = \mathbf{2}')$
 $| \text{srl-0-2': } S\text{-rl } (\text{return-spmf } (((), \text{CState-Void}, \text{CState-Half } y), (\text{auth.State-Void},$
 $s\text{-act1}), \text{auth.State-Collect } (\mathbf{g} [\uparrow] y), s\text{-act2}))$
 $(\text{map-spmf } (\lambda x. ((\text{Some } ((\mathbf{g} [\uparrow] x) [\uparrow] y, \mathbf{g} [\uparrow] x, \mathbf{g} [\uparrow] y)), \text{CState-Void},$
 $\text{CState-Half } 0), (\text{auth.State-Void}, s\text{-act1}), \text{auth.State-Collect } \mathbf{1}, s\text{-act2})) \text{rsample-nat})$

private lemma *trac-eq-core-rl: trace-core-eq real-core' (basic-core DH0-sample)*
 $(\text{UNIV} <+> \text{UNIV}) ((\text{UNIV} <+> \text{UNIV} <+> \text{UNIV}) <+> \text{UNIV} <+>$
 $\text{UNIV} <+> \text{UNIV}) ((\text{UNIV} <+> \text{UNIV}) <+> \text{UNIV} <+> \text{UNIV})$
 $(\text{return-spmf } \text{real-s-core}') (\text{return-spmf } \text{basic-core-sinit})$

proof —

have *power-commute*: $(\mathbf{g} [\uparrow] x) [\uparrow] (y :: \text{nat}) = (\mathbf{g} [\uparrow] y) [\uparrow] (x :: \text{nat})$ **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
  subgoal AState-OK for sl sr q s1 s2 s1' s2' oa
    apply (cases q)

```

```

subgoal for  $q\text{-auth1}$ 
  apply (cases  $q\text{-auth1}$ )
  subgoal for  $q\text{-drop}$  by (erule  $S\text{-rl.cases}$ ; simp)
  subgoal for  $q\text{-lfe}$ 
    apply (cases  $q\text{-lfe}$ )
    subgoal for  $q\text{-look}$ 
      proof (erule  $S\text{-rl.cases}$ , goal-cases)
        case ( $2\ s\text{-act1}\ s\text{-act2}$ ) — Corresponds to  $srl\text{-1-0}$ 
        then show ?case
          apply(cases  $s1'$ )
          apply (clarsimp simp add:  $DH0\text{-sample-def}$ )
          apply(simp add:  $bind\text{-bind-conv-pair-spmf}$ )
          apply(simp add:  $map\text{-spmf-conv-bind-spmf}[symmetric]$ )
          apply (subst  $cond\text{-spmf-fst-pair-spmf1}[unfolding\ map\text{-prod-def}\ split\text{-def}]$ )
          apply(simp)
          apply(subst ( $1\ 2$ )  $cond\text{-spmf-fst-map-Pair1}$ ; simp add:  $inj\text{-on-def}$ )
          by(subst ( $1\ 2\ 3\ 4$ )  $inv\text{-into-f-f}$ ; simp add:  $inj\text{-on-def}\ trace\text{-eq-simcl.base}$ )
S-rl.intros)
        next
          case ( $4\ x\ s\text{-act1}\ s\text{-act2}$ ) — Corresponds to  $srl\text{-1'-0}$ 
          then show ?case
            by(auto simp add:  $DH0\text{-sample-def}\ map\text{-spmf-conv-bind-spmf}[symmetric]$ )
intro!:  $trace\text{-eq-simcl.base}\ S\text{-rl.intros})
          next
            case ( $7\ s\text{-act1}\ s\text{-act2}$ ) — Corresponds to  $srl\text{-1-1}$ 
            then show ?case
              apply(clarsimp simp add:  $DH0\text{-sample-def}\ pair\text{-spmf-alt-def}$ )
              apply(subst  $bind\text{-commute-spmf}$ )
              apply(simp add:  $bind\text{-bind-conv-pair-spmf}$ )
              apply(simp add:  $map\text{-spmf-conv-bind-spmf}[symmetric]$ )
              apply (subst ( $1\ 2$ )  $cond\text{-spmf-fst-pair-spmf1}[unfolding\ map\text{-prod-def}\ split\text{-def}]$ )
              apply(simp)
              apply(subst ( $1\ 2$ )  $cond\text{-spmf-fst-map-Pair1}$ ; simp add:  $inj\text{-on-def}$ )
              by (subst ( $1\ 2\ 3\ 4$ )  $inv\text{-into-f-f}$ ; simp add:  $inj\text{-on-def}\ trace\text{-eq-simcl-map}$ )
S-rl.intros)
            next
              case ( $13\ s\text{-act1}\ s\text{-act2}$ ) — Corresponds to  $srl\text{-1-2}$ 
              then show ?case
                apply(clarsimp simp add:  $DH0\text{-sample-def}\ pair\text{-spmf-alt-def}$ )
                apply(subst  $bind\text{-commute-spmf}$ )
                apply(simp add:  $bind\text{-bind-conv-pair-spmf}$ )
                apply(simp add:  $map\text{-spmf-conv-bind-spmf}[symmetric]$ )
                apply (subst ( $1\ 2$ )  $cond\text{-spmf-fst-pair-spmf1}[unfolding\ map\text{-prod-def}\ split\text{-def}]$ )
                apply(simp)
                apply(subst ( $1\ 2$ )  $cond\text{-spmf-fst-map-Pair1}$ ; simp add:  $inj\text{-on-def}$ )
                by(subst ( $1\ 2\ 3\ 4$ )  $inv\text{-into-f-f}$ ; simp add:  $inj\text{-on-def}\ trace\text{-eq-simcl-map}$ )
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)
        done
      done
    done
  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: $UNIV \vdash_R 1_C \models \text{rassocl}_C \triangleright RES$ (fused-resource.fuse real-core' real-rest') (real-s-core', real-s-rest')

$\approx RES$ (fused-resource.fuse (lazy-core DH0-sample) lazy-rest) (basic-core-sinit, basic-rest-sinit)

proof –

have fact1: $UNIV \vdash_R 1_C \models \text{rassocl}_C \triangleright RES$ (fused-resource.fuse (basic-core DH0-sample) basic-rest) (basic-core-sinit, basic-rest-sinit) \sim

RES (fused-resource.fuse (lazy-core DH0-sample) lazy-rest) (basic-core-sinit, basic-rest-sinit)

proof –

have [simp]: $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \vdash_{\text{res}} RES$ (fused-resource.fuse (basic-core DH0-sample) basic-rest) (basic-core-sinit, basic-rest-sinit) \checkmark **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** $\mathcal{I}=\mathcal{I}\text{-full}$]
wiring-rassocl[of $\mathcal{I}\text{-full}$ $\mathcal{I}\text{-full}$ $\mathcal{I}\text{-full}$])+

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*: $UNIV \vdash_R 1_C \models \text{rassoCl}_C \triangleright RES (\text{fused-resource.fuse } \text{real-core}' \text{ real-rest}') (real-s-core', \text{real-s-rest}') \approx$
 $1_C \models \text{rassoCl}_C \triangleright RES (\text{fused-resource.fuse } (\text{basic-core } DH0\text{-sample}) \text{ basic-rest})$
(basic-core-sinit, basic-rest-sinit)
(is - \vdash_R - $\triangleright RES ?L ?s-l \approx$ - $\triangleright RES ?R ?s-r$) 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*]: $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full} \vdash_c \text{callee-of-rest } \text{basic-rest } s \checkmark$ **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*]: $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \vdash_c \text{callee-of-core } (\text{basic-core } DH0\text{-sample})$
s \checkmark 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*]: $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \vdash_c \text{callee-of-core } \text{real-core}' s \checkmark$ **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]*: $((UNIV <+> UNIV) <+> UNIV <+> UNIV) \vdash_C$
 $?L(?s-l) \approx ?R(?s-r)$
by (*rule fuse-trace-eq[where IE= \mathcal{I} -full and ICA= \mathcal{I} -full and ICU= \mathcal{I} -full*
**and IRA= \mathcal{I} -full and 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 I= \mathcal{I} -full and 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*: $\text{connect } D (\text{obsf-resource } \text{real-resource}) = \text{connect } D (\text{obsf-resource } (RES (\text{fused-resource.fuse } (\text{lazy-core } DH0\text{-sample}) \text{ lazy-rest}) (\text{basic-core-sinit, basic-rest-sinit})))$

proof -

have [*simp*]: $\mathcal{I}\text{-full} \vdash_{\text{res}} \text{real-resource} \checkmark$
proof -
have [*simp*]: $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \vdash_{\text{res}} RES (\text{fused-resource.fuse } \text{real-core}' \text{ real-rest}') (real-s-core', \text{real-s-rest}') \checkmark$
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  $\mathcal{I}' = \mathcal{I}\text{-full} \oplus_{\mathcal{I}} ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} \mathcal{I}\text{-full})$ ])
  apply (rule WT-converter-mono[of  $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}))$ 
 $\mathcal{I}\text{-full} \oplus_{\mathcal{I}} ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} \mathcal{I}\text{-full})$ ])
  apply (rule WT-converter-parallel-converter2)
  apply (rule WT-intro)+
  by (simp-all add:  $\mathcal{I}\text{-full-le-plus-}\mathcal{I}$ )
  qed

  show ?thesis
  using trace-eq-obsf-resourceI[OF trace-eq-fuse-rl, folded attach-real]
  by (rule connect-cong-trace[where  $A = \text{UNIV}$  and  $\mathcal{I} = \mathcal{I}\text{-full}$ ])
  (auto intro!: WT-obsf-resource[where  $\mathcal{I} = \mathcal{I}\text{-full}$ , simplified exception- $\mathcal{I}\text{-full}$ ])
  qed

end

end

end

end

```

12.10 Concrete security

```
context diffie-hellman begin
```

context
fixes
 $auth1\text{-rest} :: ('auth1\text{-s-rest}, auth.event, 'auth1\text{-iadv-rest}, 'auth1\text{-iusr-rest}, 'auth1\text{-oadv-rest}, 'auth1\text{-ousr-rest}) rest\text{-wstate}$ **and**
 $auth2\text{-rest} :: ('auth2\text{-s-rest}, auth.event, 'auth2\text{-iadv-rest}, 'auth2\text{-iusr-rest}, 'auth2\text{-oadv-rest}, 'auth2\text{-ousr-rest}) rest\text{-wstate}$ **and**
 $\mathcal{I}\text{-adv-rest1}$ **and** $\mathcal{I}\text{-adv-rest2}$ **and** $\mathcal{I}\text{-usr-rest1}$ **and** $\mathcal{I}\text{-usr-rest2}$ **and** $I\text{-auth1-rest}$
and $I\text{-auth2-rest}$
assumes
 $WT\text{-auth1-rest}$ [$WT\text{-intro}$]: $WT\text{-rest}$ $\mathcal{I}\text{-adv-rest1}$ $\mathcal{I}\text{-usr-rest1}$ $I\text{-auth1-rest}$ $auth1\text{-rest}$
and
 $WT\text{-auth2-rest}$ [$WT\text{-intro}$]: $WT\text{-rest}$ $\mathcal{I}\text{-adv-rest2}$ $\mathcal{I}\text{-usr-rest2}$ $I\text{-auth2-rest}$ $auth2\text{-rest}$
begin

theorem *secure*:

defines $\mathcal{I}\text{-real} \equiv ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (auth.Inp\text{-Fedit} \text{ ' } (carrier \mathcal{G})) UNIV))) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (auth.Inp\text{-Fedit} \text{ ' } (carrier \mathcal{G})) UNIV))) \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-rest1} \oplus_{\mathcal{I}} \mathcal{I}\text{-adv-rest2})$
and $\mathcal{I}\text{-common} \equiv (\mathcal{I}\text{-uniform} UNIV (key.Out\text{-Alice} \text{ ' } carrier \mathcal{G}) \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} UNIV (key.Out\text{-Bob} \text{ ' } carrier \mathcal{G})) \oplus_{\mathcal{I}} ((\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full}) \oplus_{\mathcal{I}} (\mathcal{I}\text{-usr-rest1} \oplus_{\mathcal{I}} \mathcal{I}\text{-usr-rest2}))$
and $\mathcal{I}\text{-ideal} \equiv \mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-adv-rest1} \oplus_{\mathcal{I}} \mathcal{I}\text{-adv-rest2}))$
shows *constructive-security-obsf*
 $(real\text{-resource} \text{ TYPE}(-) \text{ TYPE}(-) auth1\text{-rest} auth2\text{-rest})$
 $(key.resource (ideal\text{-rest} auth1\text{-rest} auth2\text{-rest}))$
 $(let \text{ sim} = CNV \text{ sim-callee} \text{ None} \text{ in } ((\text{sim} \models 1_C) \odot \text{lassocr}_C))$
 $\mathcal{I}\text{-real} \mathcal{I}\text{-ideal} \mathcal{I}\text{-common} \mathcal{A}$
 $(ddh.advantage \mathcal{G} (DH\text{-adversary} \text{ TYPE}(-) \text{ TYPE}(-) auth1\text{-rest} auth2\text{-rest} \mathcal{A}))$

proof

let $?sim = (let \text{ sim} = CNV \text{ sim-callee} \text{ None} \text{ in } ((\text{sim} \models 1_C) \odot \text{lassocr}_C))$

have $*[WT\text{-intro}]$: $(\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (auth.Inp\text{-Fedit} \text{ ' } carrier \mathcal{G}) UNIV)) \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} (\mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} (auth.Inp\text{-Fedit} \text{ ' } carrier \mathcal{G}) UNIV)), \mathcal{I}\text{-full} \oplus_{\mathcal{I}} \mathcal{I}\text{-full} \vdash_C CNV \text{ sim-callee} s \checkmark$ **for** s
apply $(rule \text{ WT-converter-of-callee}, \text{simp-all})$
apply $(rename\text{-tac} \text{ s q r s'}, case\text{-tac} (s, q) rule: \text{sim-callee.cases})$
by $(auto \text{ split: if-splits option.splits})$

show $\mathcal{I}\text{-real} \oplus_{\mathcal{I}} \mathcal{I}\text{-common} \vdash_{res} real\text{-resource} \text{ TYPE}(-) \text{ TYPE}(-) auth1\text{-rest} auth2\text{-rest} \checkmark$

proof –

have [$WT\text{-intro}$]: $\mathcal{I}\text{-uniform} UNIV (key.Out\text{-Alice} \text{ ' } carrier \mathcal{G}) \oplus_{\mathcal{I}} \mathcal{I}\text{-full}, \mathcal{I}\text{-uniform} (auth.Inp\text{-Send} \text{ ' } carrier \mathcal{G}) UNIV \oplus_{\mathcal{I}} \mathcal{I}\text{-uniform} UNIV (auth.Out\text{-Recv} \text{ ' } carrier \mathcal{G}) \vdash_C CNV \text{ alice-callee} C\text{State-Void} \checkmark$
apply $(rule \text{ WT-converter-of-callee-invar}[\text{where } I = pred\text{-cstate} (\lambda x. x \in carrier \mathcal{G})])$
subgoal for $s \ q$ **by** $(cases (s, q) rule: \text{alice-callee.cases}) (auto \text{ simp add:})$

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* ‘ *car-*
rier \mathcal{G}) UNIV \vdash_C CNV *bob-callee CState-Void* \checkmark
apply (*rule WT-converter-of-callee-invar*[**where** $I = \text{pred-cstate } (\lambda x. x \in$
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)*
 \checkmark
unfolding \mathcal{I} -ideal-def \mathcal{I} -common-def *key.resource-def*
apply(*rule callee-invariant-on. WT-resource-of-oracle*[**where** $I = \lambda((\text{kernel}, -), -, -,$
 $s12). \text{key.set-s-kernel kernel} \subseteq \text{carrier } \mathcal{G} \wedge \text{pred-prod } I\text{-auth1-rest } I\text{-auth2-rest } s12$];
(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*
 $WT\text{-auth2-rest}$
 $WT\text{-restD-rfunc-usr}[OF WT\text{-auth1-rest}] WT\text{-restD-rfunc-usr}[OF WT\text{-auth2-rest}]$
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*
 $WT\text{-auth2-rest}$
 $WT\text{-restD-rfunc-usr}[OF WT\text{-auth1-rest}] WT\text{-restD-rfunc-usr}[OF WT\text{-auth2-rest}]$
split: if-split-asm)
apply(*case-tac xa*)

```

    apply auto
  done
done
done

show  $\mathcal{I}$ -real,  $\mathcal{I}$ -ideal  $\vdash_C$  ?sim  $\surd$ 
  unfolding  $\mathcal{I}$ -real-def  $\mathcal{I}$ -ideal-def Let-def
  by(rule WT-intro)+

show pfinite-converter  $\mathcal{I}$ -real  $\mathcal{I}$ -ideal ?sim
proof -
  have [pfinite-intro]:pfinite-converter (( $\mathcal{I}$ -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) (CNV sim-callee s) for s
  apply(rule raw-converter-invariant.pfinite-converter-of-callee[where  $I=\lambda\cdot$ .
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  $\mathcal{I}$ -real-def  $\mathcal{I}$ -ideal-def Let-def
  by (rule pfinite-intro | rule WT-intro)+
qed

show  $0 \leq$  ddh.advantage  $\mathcal{G}$  (diffie-hellman.DH-adversary  $\mathcal{G}$  auth1-rest auth2-rest
 $\mathcal{A}$ )
  by(simp add: ddh.advantage-def)

assume WT [WT-intro]: exception- $\mathcal{I}$  ( $\mathcal{I}$ -real  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common)  $\vdash_g$   $\mathcal{A}$   $\surd$ 
show advantage  $\mathcal{A}$  (obsf-resource (?sim  $\models 1_C \triangleright$  key.resource (ideal-rest auth1-rest
auth2-rest))) (obsf-resource (real-resource TYPE(-) TYPE(-) auth1-rest auth2-rest))
 $\leq$  ddh.advantage  $\mathcal{G}$  (diffie-hellman.DH-adversary  $\mathcal{G}$  auth1-rest auth2-rest  $\mathcal{A}$ )
proof -
  have id-split[unfolded Let-def]: connect  $\mathcal{A}$  (obsf-resource (?sim  $\models 1_C \triangleright$  key.resource
(ideal-rest auth1-rest auth2-rest))) =
    connect  $\mathcal{A}$  (obsf-resource (?sim  $\models (1_C \models 1_C) \triangleright$  key.resource (ideal-rest
auth1-rest auth2-rest))) (is connect - ?L = connect - ?R)
  proof -
    note [unfolded  $\mathcal{I}$ -ideal-def, WT-intro] =  $\langle \mathcal{I}$ -real,  $\mathcal{I}$ -ideal  $\vdash_C$  ?sim  $\surd \rangle$ 
    note [unfolded  $\mathcal{I}$ -ideal-def, WT-intro] =  $\langle \mathcal{I}$ -ideal  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common  $\vdash$  res key.resource
(ideal-rest auth1-rest auth2-rest)  $\surd \rangle$ 

```

```

have [WT-intro]: WT-rest ( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$  ( $\mathcal{I}$ -adv-rest1  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -adv-rest2)) ( $\mathcal{I}$ -full  $\oplus_{\mathcal{I}}$ 
( $\mathcal{I}$ -usr-rest1  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -usr-rest2)) ( $\lambda(-, s12).$  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- $\mathcal{I}$  (exception- $\mathcal{I}$  ( $\mathcal{I}$ -real  $\oplus_{\mathcal{I}}$   $\mathcal{I}$ -common))  $\vdash_R$  ?L  $\sim$  ?R
apply (rule obsf-resource-eq- $\mathcal{I}$ -cong)
apply (rule eq- $\mathcal{I}$ -attach-on')
apply (rule WT-intro | simp)+
apply(rule parallel-converter2-eq- $\mathcal{I}$ -cong)
apply(rule eq- $\mathcal{I}$ -converter-refl)
apply (rule  $\langle \mathcal{I}$ -real,  $\mathcal{I}$ -ideal  $\vdash_C$  ?sim  $\surd$   $\rangle$ [unfolded assms Let-def])
apply (rule eq- $\mathcal{I}$ -converter-sym)
apply (rule parallel-converter2-id-id)
by (auto simp add:  $\mathcal{I}$ -real-def  $\mathcal{I}$ -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  $\mathcal{G} ::$  security  $\Rightarrow$  'grp cyclic-group
  assumes diffie-hellman [locale-witness]:  $\bigwedge \eta.$  diffie-hellman ( $\mathcal{G}$   $\eta$ )
begin

```

sublocale *diffie-hellman* \mathcal{G} η **for** η ..

definition *real-resource'* **where** *real-resource'* *rest1* *rest2* $\eta = \text{real-resource } \text{TYPE}(-)$
 $\text{TYPE}(-) \eta$ (*rest1* η) (*rest2* η)

definition *ideal-resource'* **where** *ideal-resource'* *rest1* *rest2* $\eta = \text{key.resource } \eta$
(*ideal-rest* (*rest1* η) (*rest2* η))

definition *sim'* **where** *sim'* $\eta = (\text{let } \text{sim} = \text{CNV } (\text{sim-callee } \eta) \text{ None in } ((\text{sim} \models$
 $1_C) \odot \text{lassocr}_C))$

context

fixes

auth1-rest :: $\text{nat} \Rightarrow ('auth1-s-rest, \text{auth.event}, 'auth1-iadv-rest, 'auth1-iusr-rest,$
'*auth1-oadv-rest*, '*auth1-ousr-rest*) *rest-wstate* **and**

auth2-rest :: $\text{nat} \Rightarrow ('auth2-s-rest, \text{auth.event}, 'auth2-iadv-rest, 'auth2-iusr-rest,$
'*auth2-oadv-rest*, '*auth2-ousr-rest*) *rest-wstate* **and**

I-adv-rest1 **and** *I-adv-rest2* **and** *I-usr-rest1* **and** *I-usr-rest2* **and** *I-auth1-rest*
and *I-auth2-rest*

assumes

WT-auth1-rest: $\bigwedge \eta. \text{WT-rest } (\text{I-adv-rest1 } \eta) (\text{I-usr-rest1 } \eta) (\text{I-auth1-rest } \eta)$
(*auth1-rest* η) **and**

WT-auth2-rest: $\bigwedge \eta. \text{WT-rest } (\text{I-adv-rest2 } \eta) (\text{I-usr-rest2 } \eta) (\text{I-auth2-rest } \eta)$
(*auth2-rest* η)

begin

theorem *secure*:

defines *I-real* $\equiv \lambda \eta. ((\text{I-full} \oplus_{\mathcal{I}} (\text{I-full} \oplus_{\mathcal{I}} \text{I-uniform } (\text{auth.Inp-Fedit } '(\text{carrier } (\mathcal{G} \eta)))$
 $\text{UNIV})) \oplus_{\mathcal{I}} (\text{I-full} \oplus_{\mathcal{I}} (\text{I-full} \oplus_{\mathcal{I}} \text{I-uniform } (\text{auth.Inp-Fedit } '(\text{carrier } (\mathcal{G} \eta)))$
 $\text{UNIV}))) \oplus_{\mathcal{I}} (\text{I-adv-rest1 } \eta \oplus_{\mathcal{I}} \text{I-adv-rest2 } \eta)$

and *I-common* $\equiv \lambda \eta. (\text{I-uniform } \text{UNIV } (\text{key.Out-Alice } '(\text{carrier } (\mathcal{G} \eta)))$
 $\oplus_{\mathcal{I}} \text{I-uniform } \text{UNIV } (\text{key.Out-Bob } '(\text{carrier } (\mathcal{G} \eta))) \oplus_{\mathcal{I}} ((\text{I-full} \oplus_{\mathcal{I}} \text{I-full}) \oplus_{\mathcal{I}}$
 $(\text{I-usr-rest1 } \eta \oplus_{\mathcal{I}} \text{I-usr-rest2 } \eta))$

and *I-ideal* $\equiv \lambda \eta. \text{I-full} \oplus_{\mathcal{I}} (\text{I-full} \oplus_{\mathcal{I}} (\text{I-adv-rest1 } \eta \oplus_{\mathcal{I}} \text{I-adv-rest2 } \eta))$

assumes *DDH*: *negligible* ($\lambda \eta. \text{ddh.advantage } (\mathcal{G} \eta) (\text{DH-adversary } \text{TYPE}(-) \text{TYPE}(-)$
 η (*auth1-rest* η) (*auth2-rest* η) ($\mathcal{A} \eta$))

shows *constructive-security-obsf'* (*real-resource'* *auth1-rest* *auth2-rest*) (*ideal-resource'*
auth1-rest *auth2-rest*) *sim'* *I-real* *I-ideal* *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) (*I-real* η) (*I-ideal* η)

(*I-common* η)

($\mathcal{A} \eta$) (*ddh.advantage* ($\mathcal{G} \eta$) (*DH-adversary* $\text{TYPE}(-) \text{TYPE}(-) \eta$ (*auth1-rest*
 η) (*auth2-rest* η) ($\mathcal{A} \eta$))) **for** η

unfolding *real-resource'-def* *ideal-resource'-def* *sim'-def* *I-real-def* *I-common-def*
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 I-adv-rest1 I-usr-rest1 I-auth1-rest auth1-rest*

and *WT-auth2-rest [WT-intro]: WT-rest I-adv-rest2 I-usr-rest2 I-auth2-rest auth2-rest*

shows *WT-rest (I-full $\oplus_{\mathcal{I}}$ (I-adv-rest1 $\oplus_{\mathcal{I}}$ I-adv-rest2)) ((I-full $\oplus_{\mathcal{I}}$ I-full) $\oplus_{\mathcal{I}}$ (I-usr-rest1 $\oplus_{\mathcal{I}}$ I-usr-rest2))*

($\lambda(-, s). \text{pred-prod } I\text{-auth1-rest } I\text{-auth2-rest } s$) (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 I-adv-resta I-usr-resta I-auth-rest auth-rest*

and *WT-rest I-adv-rest1 I-usr-rest1 I-auth1-rest auth1-rest*

and *WT-rest I-adv-rest2 I-usr-rest2 I-auth2-rest auth2-rest*

shows

constructive-security-obsf

```

(1C |= wiring-c1r22-c1r22 (CNV otp.enc-callee ()) (CNV otp.dec-callee ())) |=
1C ▷
  fused-wiring ▷ diffie-hellman.real-resource 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))
  ((1C ⊙
    (parallel-wiring ⊙ ((let sim = CNV dh.sim-callee None in (sim |= 1C) ⊙
lassocr_C) |= 1C) ⊙ parallel-wiring) ⊙
    1C) ⊙
    (otp.sim |= 1C))
  (((I-full ⊕I (I-full ⊕I I-uniform (otp.sec.Inp-Fedit ‘carrier G) UNIV)) ⊕I
    (I-full ⊕I (I-full ⊕I I-uniform (otp.sec.Inp-Fedit ‘carrier G) UNIV)))
⊕I
  (I-full ⊕I (I-full ⊕I I-uniform (otp.sec.Inp-Fedit ‘carrier L) UNIV))) ⊕I
  ((I-adv-rest1 ⊕I I-adv-rest2) ⊕I I-adv-resta))
  ((I-full ⊕I (I-full ⊕I I-uniform (otp.sec.Inp-Fedit ‘carrier L) UNIV)) ⊕I
  ((I-full ⊕I (I-adv-rest1 ⊕I I-adv-rest2)) ⊕I I-adv-resta))
  ((I-uniform (otp.sec.Inp-Send ‘carrier L) UNIV ⊕I I-uniform UNIV
(otp.sec.Out-Recv ‘carrier L)) ⊕I
  (((I-full ⊕I I-full) ⊕I (I-usr-rest1 ⊕I I-usr-rest2)) ⊕I I-usr-resta))
  A (0 + (ddh.advantage G
    (diffie-hellman.DH-adversary G auth1-rest auth2-rest
    (absorb
      (absorb A
        (obsf-converter (1C |= wiring-c1r22-c1r22 (CNV otp.enc-callee
    (CNV otp.dec-callee ())) |= 1C)))
      (obsf-converter
        (fused-wiring ⊙ (1C |∞ converter-of-resource (1C |= 1C ▷
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=1C and w-adv-ideal-inv=1C])
  apply(rule parallel-constructive-security-obsf-fuse)
  apply(fold carrier-G-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.