CryptHOL

Andreas Lochbihler

March 19, 2025

Abstract

CryptHOL provides a framework for formalising cryptographic arguments in Isabelle/HOL. It shallowly embeds a probabilistic functional programming language in higher order logic. The language features monadic sequencing, recursion, random sampling, failures and failure handling, and black-box access to oracles. Oracles are probabilistic functions which maintain hidden state between different invocations. All operators are defined in the new semantic domain of generative probabilistic values, a codatatype. We derive proof rules for the operators and establish a connection with the theory of relational parametricity. Thus, the resuting proofs are trustworthy and comprehensible, and the framework is extensible and widely applicable.

The framework is used in the accompanying AFP entry "Game-based Cryptography in HOL". There, we show-case our framework by formalizing different game-based proofs from the literature. This formalisation continues the work described in the author's ESOP 2016 paper [1].

A tutorial in the AFP entry *Game-based cryptography* explains how CryptHOL can be used to formalize game-based cryptography proofs.

Contents

1 Miscellaneous library additions

theory *Misc-CryptHOL* **imports**
Probabilistic-While.While-SPMF
HOL−Library.Rewrite
HOL−Library.Simps-Case-Conv
HOL−Library.Type-Length
HOL−Eisbach.Eisbach
Coinductive.TLList
Monad-Normalisation.Monad-Normalisation
Monomorphic-Monad.Monomorphic-Monad
Applicative-Lifting.Applicative
begin

hide-const (**open**) *Henstock-Kurzweil-Integration.negligible*

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

1.1 HOL

lemma *asm-rl-conv*: $(PROP\ P \Longrightarrow PROP\ P) \equiv Trueprop\ True$
by(*rule equal-intr-rule*) *iprover+*

named-theorems *if-distribs Distributivity theorems for If*

lemma *if-mono-cong*: $[\![b \Longrightarrow x \le x'; \neg\ b \Longrightarrow y \le y'\ ]\!] \Longrightarrow If\ b\ x\ y \le If\ b\ x'\ y'$
by *simp*

lemma *if-cong-then*: $[\![\ b = b';\ b' \Longrightarrow t = t';\ e = e'\ ]\!] \Longrightarrow If\ b\ t\ e = If\ b'\ t'\ e'$
by *simp*

lemma *if-False-eq*: $[\![\ b \Longrightarrow False;\ e = e'\ ]\!] \Longrightarrow If\ b\ t\ e = e'$
by *auto*

lemma *imp-OO-imp* [*simp*]: $(\longrightarrow)\ OO\ (\longrightarrow) = (\longrightarrow)$
by *auto*

lemma *inj-on-fun-updD*: $[\![\ inj\text{-}on\ (f(x := y))\ A;\ x \notin A\ ]\!] \Longrightarrow inj\text{-}on\ f\ A$
by(*auto simp add*: *inj-on-def split*: *if-split-asm*)

lemma *disjoint-notin1*: $[\![\ A \cap B = \{\};\ x \in B\ ]\!] \Longrightarrow x \notin A$ **by** *auto*

lemma *Least-le-Least*:
fixes $x :: 'a :: wellorder$
assumes $Q\ x$
and Q: $\bigwedge x.\ Q\ x \Longrightarrow \exists\, y{\le}x.\ P\ y$
shows $Least\ P \le Least\ Q$
by (*metis assms order-trans wellorder-Least-lemma*)

lemma *is-empty-image* [*simp*]: $Set.is\text{-}empty\ (f\ `\ A) = Set.is\text{-}empty\ A$

by(*auto simp add*: *Set.is-empty-def*)

1.2 Relations

inductive *Imagep* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ *'b* ⇒ *bool*
 for *R P*
where *ImagepI*: ⟦ *P x*; *R x y* ⟧ ⟹ *Imagep R P y*

lemma *r-r-into-tranclp*: ⟦ *r x y*; *r y z* ⟧ ⟹ *r*^++ *x z*
by(*rule tranclp.trancl-into-trancl*)(*rule tranclp.r-into-trancl*)

lemma *transp-tranclp-id*:
 assumes *transp R*
 shows *tranclp R* = *R*
proof(*intro ext iffI*)
 fix *x y*
 assume *R*^++ *x y*
 thus *R x y* **by** *induction*(*blast dest*: *transpD*[*OF assms*])+
qed *simp*

lemma *transp-inv-image*: *transp r* ⟹ *transp* (λ*x y*. *r* (*f x*) (*f y*))
using *trans-inv-image*[**where** *r*={(*x*, *y*). *r x y*} **and** *f* = *f*]
by(*simp add*: *transp-trans inv-image-def*)

lemma *Domainp-conversep*: *Domainp* R^{-1-1} = *Rangep R*
by(*auto*)

lemma *bi-unique-rel-set-bij-betw*:
 assumes *unique*: *bi-unique R*
 and *rel*: *rel-set R A B*
 shows ∃*f*. *bij-betw f A B* ∧ (∀ *x*∈*A*. *R x* (*f x*))
proof −
 from *assms* **obtain** *f* **where** *f*: ⋀*x*. *x* ∈ *A* ⟹ *R x* (*f x*) **and** *B*: ⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B*
  apply(*atomize-elim*)
  apply(*fold all-conj-distrib*)
  apply(*subst choice-iff*[*symmetric*])
  apply(*auto dest*: *rel-setD1*)
  done
 have *inj-on f A* **by**(*rule inj-onI*)(*auto dest*!: *f dest*: *bi-uniqueDl*[*OF unique*])
 moreover have *f* ‘ *A* = *B* **using** *rel*
  by(*auto 4 3 intro*: *B dest*: *rel-setD2 f bi-uniqueDr*[*OF unique*])
 ultimately have *bij-betw f A B* **unfolding** *bij-betw-def* **..**
 thus *?thesis* **using** *f* **by** *blast*
qed

definition *restrict-relp* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'b* ⇒ *bool*) ⇒ *'a* ⇒ *'b* ⇒ *bool*
 (‹- ↿ (- ⊗ -)› [*53*, *54*, *54*] *53*)

where *restrict-relp* $R\ P\ Q = (\lambda x\ y.\ R\ x\ y \wedge P\ x \wedge Q\ y)$

lemma *restrict-relp-apply* [*simp*]: $(R \upharpoonright P \otimes Q)\ x\ y \longleftrightarrow R\ x\ y \wedge P\ x \wedge Q\ y$
by(*simp add*: *restrict-relp-def*)

lemma *restrict-relpI* [*intro?*]: $[\![\ R\ x\ y;\ P\ x;\ Q\ y\ ]\!] \Longrightarrow (R \upharpoonright P \otimes Q)\ x\ y$
by(*simp add*: *restrict-relp-def*)

lemma *restrict-relpE* [*elim?*, *cases pred*]:
assumes $(R \upharpoonright P \otimes Q)\ x\ y$
obtains (*restrict-relp*) $R\ x\ y\ P\ x\ Q\ y$
using *assms* **by**(*simp add*: *restrict-relp-def*)

lemma *conversep-restrict-relp* [*simp*]: $(R \upharpoonright P \otimes Q)^{-1-1} = R^{-1-1} \upharpoonright Q \otimes P$
by(*auto simp add*: *fun-eq-iff*)

lemma *restrict-relp-restrict-relp* [*simp*]: $R \upharpoonright P \otimes Q \upharpoonright P' \otimes Q' = R \upharpoonright$ *inf* $P\ P' \otimes$ *inf* $Q\ Q'$
by(*auto simp add*: *fun-eq-iff*)

lemma *restrict-relp-cong*:
$[\![\ P = P';\ Q = Q';\ \bigwedge x\ y.\ [\![\ P\ x;\ Q\ y\ ]\!] \Longrightarrow R\ x\ y = R'\ x\ y\ ]\!] \Longrightarrow R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$
by(*auto simp add*: *fun-eq-iff*)

lemma *restrict-relp-cong-simp*:
$[\![\ P = P';\ Q = Q';\ \bigwedge x\ y.\ P\ x$ =*simp*=> $Q\ y$ =*simp*=> $R\ x\ y = R'\ x\ y\ ]\!] \Longrightarrow R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$
by(*rule restrict-relp-cong*; *simp add*: *simp-implies-def*)

lemma *restrict-relp-parametric* [*transfer-rule*]:
includes *lifting-syntax* **shows**
((A ===> B ===> (=)) ===> (A ===> (=)) ===> (B ===> (=)) ===> A ===> B ===> (=)) *restrict-relp restrict-relp*
unfolding *restrict-relp-def*[*abs-def*] **by** *transfer-prover*

lemma *restrict-relp-mono*: $[\![\ R \le R';\ P \le P';\ Q \le Q'\ ]\!] \Longrightarrow R \upharpoonright P \otimes Q \le R' \upharpoonright P' \otimes Q'$
by(*simp add*: *le-fun-def*)

lemma *restrict-relp-mono'*:
$[\![\ (R \upharpoonright P \otimes Q)\ x\ y;\ [\![\ R\ x\ y;\ P\ x;\ Q\ y\ ]\!] \Longrightarrow R'\ x\ y$ &&& $P'\ x$ &&& $Q'\ y\ ]\!]$
$\Longrightarrow (R' \upharpoonright P' \otimes Q')\ x\ y$
by(*auto dest*: *conjunctionD1 conjunctionD2*)

lemma *restrict-relp-DomainpD*: *Domainp* $(R \upharpoonright P \otimes Q)\ x \Longrightarrow$ *Domainp* $R\ x \wedge P\ x$
by(*auto simp add*: *Domainp.simps*)

lemma *restrict-relp-True*: *R* ↿ (λ-. *True*) ⊗ (λ-. *True*) = *R*
by(*simp add*: *fun-eq-iff*)

lemma *restrict-relp-False1*: *R* ↿ (λ-. *False*) ⊗ *Q* = *bot*
by(*simp add*: *fun-eq-iff*)

lemma *restrict-relp-False2*: *R* ↿ *P* ⊗ (λ-. *False*) = *bot*
by(*simp add*: *fun-eq-iff*)

definition *rel-prod2* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a* ⇒ (*'c* × *'b*) ⇒ *bool*
where *rel-prod2 R a* = (λ(*c*, *b*). *R a b*)

lemma *rel-prod2-simps* [*simp*]: *rel-prod2 R a* (*c*, *b*) ⟷ *R a b*
by(*simp add*: *rel-prod2-def*)

lemma *restrict-rel-prod*:
rel-prod (*R* ↿ *I1* ⊗ *I2*) (*S* ↿ *I1'* ⊗ *I2'*) = *rel-prod R S* ↿ *pred-prod I1 I1'* ⊗ *pred-prod I2 I2'*
by(*auto simp add*: *fun-eq-iff*)

lemma *restrict-rel-prod1*:
rel-prod (*R* ↿ *I1* ⊗ *I2*) *S* = *rel-prod R S* ↿ *pred-prod I1* (λ-. *True*) ⊗ *pred-prod I2* (λ-. *True*)
by(*simp add*: *restrict-rel-prod*[*symmetric*] *restrict-relp-True*)

lemma *restrict-rel-prod2*:
rel-prod R (*S* ↿ *I1* ⊗ *I2*) = *rel-prod R S* ↿ *pred-prod* (λ-. *True*) *I1* ⊗ *pred-prod* (λ-. *True*) *I2*
by(*simp add*: *restrict-rel-prod*[*symmetric*] *restrict-relp-True*)

consts *relcompp-witness* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'b* ⇒ *'c* ⇒ *bool*) ⇒ *'a* × *'c* ⇒ *'b*
specification (*relcompp-witness*)
relcompp-witness1: (*A OO B*) (*fst xy*) (*snd xy*) ⟹ *A* (*fst xy*) (*relcompp-witness A B xy*)
relcompp-witness2: (*A OO B*) (*fst xy*) (*snd xy*) ⟹ *B* (*relcompp-witness A B xy*) (*snd xy*)
apply(*fold all-conj-distrib*)
apply(*rule choice allI*)+
by(*auto intro*: *choice allI*)

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

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

lemma *relcompp-witness-eq* [*simp*]: *relcompp-witness* (=) (=) (*x*, *x*) = *x*
using *relcompp-witness*(*1*)[*of* (=) (=) *x x*] **by**(*simp add*: *eq-OO*)

1.3 Pairs

lemma *split-apfst* [*simp*]: *case-prod h* (*apfst f xy*) = *case-prod* (*h* ∘ *f*) *xy*
by(*cases xy*) *simp*

definition *corec-prod* :: ('*s* ⇒ '*a*) ⇒ ('*s* ⇒ '*b*) ⇒ '*s* ⇒ '*a* × '*b*
where *corec-prod f g* = (λ*s*. (*f s*, *g s*))

lemma *corec-prod-apply*: *corec-prod f g s* = (*f s*, *g s*)
by(*simp add*: *corec-prod-def*)

lemma *corec-prod-sel* [*simp*]:
  shows *fst-corec-prod*: *fst* (*corec-prod f g s*) = *f s*
  and *snd-corec-prod*: *snd* (*corec-prod f g s*) = *g s*
by(*simp-all add*: *corec-prod-apply*)

lemma *apfst-corec-prod* [*simp*]: *apfst h* (*corec-prod f g s*) = *corec-prod* (*h* ∘ *f*) *g s*
by(*simp add*: *corec-prod-apply*)

lemma *apsnd-corec-prod* [*simp*]: *apsnd h* (*corec-prod f g s*) = *corec-prod f* (*h* ∘ *g*) *s*
by(*simp add*: *corec-prod-apply*)

lemma *map-corec-prod* [*simp*]: *map-prod f g* (*corec-prod h k s*) = *corec-prod* (*f* ∘ *h*) (*g* ∘ *k*) *s*
by(*simp add*: *corec-prod-apply*)

lemma *split-corec-prod* [*simp*]: *case-prod h* (*corec-prod f g s*) = *h* (*f s*) (*g s*)
by(*simp add*: *corec-prod-apply*)

lemma *Pair-fst-Unity*: (*fst x*, ()) = *x*
  by(*cases x*) *simp*

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

lemma *rprodl-simps* [*simp*]: *rprodl* ((*a*, *b*), *c*) = (*a*, (*b*, *c*))
  by(*simp add*: *rprodl-def*)

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

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

lemma *lprodr-simps* [*simp*]: *lprodr* (*a*, *b*, *c*) = ((*a*, *b*), *c*)
  by(*simp add*: *lprodr-def*)

lemma *lprodr-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**

(*rel-prod A* (*rel-prod B C*) ===> *rel-prod* (*rel-prod A B*) *C*) *lprodr lprodr*
unfolding *lprodr-def* **by** *transfer-prover*

lemma *lprodr-inverse* [*simp*]: *rprodl* (*lprodr x*) = *x*
by(*cases x*) *auto*

lemma *rprodl-inverse* [*simp*]: *lprodr* (*rprodl x*) = *x*
by(*cases x*) *auto*

lemma *pred-prod-mono′* [*mono*]:
pred-prod A B xy ⟶ *pred-prod A′ B′ xy*
if ⋀*x*. *A x* ⟶ *A′ x* ⋀*y*. *B y* ⟶ *B′ y*
using *that* **by**(*cases xy*) *auto*

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

1.4 Sums

lemma *islE*:
assumes *isl x*
obtains *l* **where** *x* = *Inl l*
using *assms* **by**(*cases x*) *auto*

lemma *Inl-in-Plus* [*simp*]: *Inl x* ∈ *A* <+> *B* ⟷ *x* ∈ *A*
by *auto*

lemma *Inr-in-Plus* [*simp*]: *Inr x* ∈ *A* <+> *B* ⟷ *x* ∈ *B*
by *auto*

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

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

lemma *inj-on-map-sum* [*simp*]:
⟦ *inj-on f A*; *inj-on g B* ⟧ ⟹ *inj-on* (*map-sum f g*) (*A* <+> *B*)
proof(*rule inj-onI*, *goal-cases*)
case (*1 x y*)
then show *?case* **by**(*cases x*; *cases y*; *auto simp add*: *inj-on-def*)
qed

lemma *inv-into-map-sum*:
inv-into (*A* <+> *B*) (*map-sum f g*) *x* = *map-sum* (*inv-into A f*) (*inv-into B g*) *x*
if *x* ∈ *f* ‘ *A* <+> *g* ‘ *B inj-on f A inj-on g B*
using *that* **by**(*cases rule*: *PlusE*[*consumes 1*])(*auto simp add*: *inv-into-f-eq f-inv-into-f*)

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

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

lemma *rsuml-lsumr* [*simp*]: *rsuml* (*lsumr* *x*) = *x*
by(*cases x rule*: *lsumr.cases*) *simp-all*

lemma *lsumr-rsuml* [*simp*]: *lsumr* (*rsuml* *x*) = *x*
by(*cases x rule*: *rsuml.cases*) *simp-all*

1.5 Option

declare *is-none-bind* [*simp*]

lemma *case-option-collapse*: *case-option* *x* (λ-. *x*) *y* = *x*
by(*simp split*: *option.split*)

lemma *indicator-single-Some*: *indicator* {*Some x*} (*Some y*) = *indicator* {*x*} *y*
by(*simp split*: *split-indicator*)

1.5.1 Predicator and relator

lemma *option-pred-mono-strong*:
⟦ *pred-option P x*; ⋀*a*. ⟦ *a* ∈ *set-option x*; *P a* ⟧ ⟹ *P′ a* ⟧ ⟹ *pred-option P′ x*
by(*fact option.pred-mono-strong*)

lemma *option-pred-map* [*simp*]: *pred-option P* (*map-option f x*) = *pred-option* (*P* ∘ *f*) *x*
by(*fact option.pred-map*)

lemma *option-pred-o-map* [*simp*]: *pred-option P* ∘ *map-option f* = *pred-option* (*P* ∘ *f*)
by(*simp add*: *fun-eq-iff*)

lemma *option-pred-bind* [*simp*]: *pred-option P* (*Option.bind x f*) = *pred-option* (*pred-option P* ∘ *f*) *x*
by(*simp add*: *pred-option-def*)

lemma *pred-option-conj* [*simp*]:
pred-option (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *pred-option P x* ∧ *pred-option Q x*)
by(*auto simp add*: *pred-option-def*)

lemma *pred-option-top* [*simp*]:
pred-option (λ-. *True*) = (λ-. *True*)

by(*fact option.pred-True*)

lemma *rel-option-restrict-relpI* [*intro?*]:
⟦ *rel-option R x y*; *pred-option P x*; *pred-option Q y* ⟧ ⟹ *rel-option* ($R \upharpoonright P \otimes Q$) *x y*
by(*erule option.rel-mono-strong*) *simp*

lemma *rel-option-restrict-relpE* [*elim?*]:
assumes *rel-option* ($R \upharpoonright P \otimes Q$) *x y*
obtains *rel-option R x y pred-option P x pred-option Q y*
proof
show *rel-option R x y* **using** *assms* **by**(*auto elim*!: *option.rel-mono-strong*)
have *pred-option* (*Domainp* ($R \upharpoonright P \otimes Q$)) *x* **using** *assms* **by**(*fold option.Domainp-rel*) *blast*
then show *pred-option P x* **by**(*rule option-pred-mono-strong*)(*blast dest*!: *restrict-relp-DomainpD*)
have *pred-option* (*Domainp* $(R \upharpoonright P \otimes Q)^{-1-1}$) *y* **using** *assms*
by(*fold option.Domainp-rel*)(*auto simp only*: *option.rel-conversep Domainp-conversep*)
then show *pred-option Q y* **by**(*rule option-pred-mono-strong*)(*auto dest*!: *restrict-relp-DomainpD*)
qed

lemma *rel-option-restrict-relp-iff*:
rel-option ($R \upharpoonright P \otimes Q$) *x y* ⟷ *rel-option R x y* ∧ *pred-option P x* ∧ *pred-option Q y*
by(*blast intro*: *rel-option-restrict-relpI elim*: *rel-option-restrict-relpE*)

lemma *option-rel-map-restrict-relp*:
shows *option-rel-map-restrict-relp1*:
rel-option ($R \upharpoonright P \otimes Q$) (*map-option f x*) = *rel-option* ($R \circ f \upharpoonright P \circ f \otimes Q$) *x*
and *option-rel-map-restrict-relp2*:
rel-option ($R \upharpoonright P \otimes Q$) *x* (*map-option g y*) = *rel-option* (($\lambda x.\ R\ x \circ g$) $\upharpoonright P \otimes Q \circ g$) *x y*
by(*simp-all add*: *option.rel-map restrict-relp-def fun-eq-iff*)

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

lemma *rel-witness-option*:
shows *set-rel-witness-option*: ⟦ *rel-option A x y*; (*a*, *b*) ∈ *set-option* (*rel-witness-option* (*x*, *y*)) ⟧ ⟹ *A a b*
and *map1-rel-witness-option*: *rel-option A x y* ⟹ *map-option fst* (*rel-witness-option* (*x*, *y*)) = *x*
and *map2-rel-witness-option*: *rel-option A x y* ⟹ *map-option snd* (*rel-witness-option* (*x*, *y*)) = *y*
by(*cases* (*x*, *y*) *rule*: *rel-witness-option.cases*; *simp*; *fail*)+

lemma *rel-witness-option1*:
  assumes *rel-option A x y*
  shows *rel-option* (λa (a', b). $a = a' \wedge A\ a'\ b$) *x* (*rel-witness-option* (x, y))
  using *map1-rel-witness-option*[*OF assms*, *symmetric*]
  unfolding *option.rel-eq*[*symmetric*] *option.rel-map*
  by(*rule option.rel-mono-strong*)(*auto intro*: *set-rel-witness-option*[*OF assms*])

lemma *rel-witness-option2*:
  assumes *rel-option A x y*
  shows *rel-option* (λ(a, b') b. $b = b' \wedge A\ a\ b'$) (*rel-witness-option* (x, y)) *y*
  using *map2-rel-witness-option*[*OF assms*]
  unfolding *option.rel-eq*[*symmetric*] *option.rel-map*
  by(*rule option.rel-mono-strong*)(*auto intro*: *set-rel-witness-option*[*OF assms*])

1.5.2 Orders on option

abbreviation *le-option* :: *'a option* ⇒ *'a option* ⇒ *bool*
where *le-option* ≡ *ord-option* (=)

lemma *le-option-bind-mono*:
  ⟦ *le-option x y*; ⋀*a*. *a* ∈ *set-option x* ⟹ *le-option* (*f a*) (*g a*) ⟧
  ⟹ *le-option* (*Option.bind x f*) (*Option.bind y g*)
by(*cases x*) *simp-all*

lemma *le-option-refl* [*simp*]: *le-option x x*
by(*cases x*) *simp-all*

lemma *le-option-conv-option-ord*: *le-option* = *option-ord*
by(*auto simp add*: *fun-eq-iff flat-ord-def elim*: *ord-option.cases*)

definition *pcr-Some* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a* ⇒ *'b option* ⇒ *bool*
where *pcr-Some R x y* ⟷ (∃ *z*. *y* = *Some z* ∧ *R x z*)

lemma *pcr-Some-simps* [*simp*]: *pcr-Some R x* (*Some y*) ⟷ *R x y*
by(*simp add*: *pcr-Some-def*)

lemma *pcr-SomeE* [*cases pred*]:
  assumes *pcr-Some R x y*
  obtains (*pcr-Some*) *z* **where** *y* = *Some z R x z*
using *assms* **by**(*auto simp add*: *pcr-Some-def*)

1.5.3 Filter for option

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

lemma *set-filter-option* [*simp*]: *set-option* (*filter-option* P x) = {y ∈ *set-option* x. P y}
by(*cases* x) *auto*

lemma *filter-map-option*: *filter-option* P (*map-option* f x) = *map-option* f (*filter-option* ($P \circ f$) x)
by(*cases* x) *simp-all*

lemma *is-none-filter-option* [*simp*]: *Option.is-none* (*filter-option* P x) ⟷ *Option.is-none* x ∨ ¬ P (*the* x)
by(*cases* x) *simp-all*

lemma *filter-option-eq-Some-iff* [*simp*]: *filter-option* P x = *Some* y ⟷ x = *Some* y ∧ P y
by(*cases* x) *auto*

lemma *Some-eq-filter-option-iff* [*simp*]: *Some* y = *filter-option* P x ⟷ x = *Some* y ∧ P y
by(*cases* x) *auto*

lemma *filter-conv-bind-option*: *filter-option* P x = *Option.bind* x (λy. *if* P y *then Some* y *else None*)
by(*cases* x) *simp-all*

1.5.4 Assert for option

primrec *assert-option* :: *bool* ⇒ *unit option* **where**
assert-option True = *Some* ()
| *assert-option False* = *None*

lemma *set-assert-option-conv*: *set-option* (*assert-option* b) = (*if* b *then* {()} *else* {})
by(*simp*)

lemma *in-set-assert-option* [*simp*]: x ∈ *set-option* (*assert-option* b) ⟷ b
by(*cases* b) *simp-all*

1.5.5 Join on options

definition *join-option* :: *'a option option* ⇒ *'a option*
where *join-option* x = (*case* x *of Some* y ⇒ y | *None* ⇒ *None*)

simps-of-case *join-simps* [*simp*, *code*]: *join-option-def*

lemma *set-join-option* [*simp*]: *set-option* (*join-option* x) = ⋃ (*set-option* ‘ *set-option* x)
by(*cases* x)(*simp-all*)

lemma *in-set-join-option*: x ∈ *set-option* (*join-option* (*Some* (*Some* x)))
by *simp*

lemma *map-join-option*: *map-option f* (*join-option x*) = *join-option* (*map-option* (*map-option f*) *x*)
by(*cases x*) *simp-all*

lemma *bind-conv-join-option*: *Option.bind x f* = *join-option* (*map-option f x*)
by(*cases x*) *simp-all*

lemma *join-conv-bind-option*: *join-option x* = *Option.bind x id*
by(*cases x*) *simp-all*

lemma *join-option-parametric* [*transfer-rule*]:
  includes *lifting-syntax* **shows**
  (*rel-option* (*rel-option R*) ===> *rel-option R*) *join-option join-option*
unfolding *join-conv-bind-option*[*abs-def*] **by** *transfer-prover*

lemma *join-option-eq-Some* [*simp*]: *join-option x* = *Some y* ⟷ *x* = *Some* (*Some y*)
by(*cases x*) *simp-all*

lemma *Some-eq-join-option* [*simp*]: *Some y* = *join-option x* ⟷ *x* = *Some* (*Some y*)
by(*cases x*) *auto*

lemma *join-option-eq-None*: *join-option x* = *None* ⟷ *x* = *None* ∨ *x* = *Some None*
by(*cases x*) *simp-all*

lemma *None-eq-join-option*: *None* = *join-option x* ⟷ *x* = *None* ∨ *x* = *Some None*
by(*cases x*) *auto*

1.5.6 Zip on options

function *zip-option* :: *'a option* ⇒ *'b option* ⇒ (*'a* × *'b*) *option*
where
  zip-option (*Some x*) (*Some y*) = *Some* (*x*, *y*)
| *zip-option* - *None* = *None*
| *zip-option None* - = *None*
by *pat-completeness auto*
termination by *lexicographic-order*

lemma *zip-option-eq-Some-iff* [*iff*]:
  zip-option x y = *Some* (*a*, *b*) ⟷ *x* = *Some a* ∧ *y* = *Some b*
by(*cases* (*x*, *y*) *rule*: *zip-option.cases*) *simp-all*

lemma *set-zip-option* [*simp*]:
  set-option (*zip-option x y*) = *set-option x* × *set-option y*
by *auto*

lemma *zip-map-option1*: *zip-option* (*map-option f x*) *y* = *map-option* (*apfst f*) (*zip-option x y*)
by(*cases* (*x*, *y*) *rule*: *zip-option.cases*) *simp-all*

lemma *zip-map-option2*: *zip-option x* (*map-option g y*) = *map-option* (*apsnd g*) (*zip-option x y*)
by(*cases* (*x*, *y*) *rule*: *zip-option.cases*) *simp-all*

lemma *map-zip-option*:
map-option (*map-prod f g*) (*zip-option x y*) = *zip-option* (*map-option f x*) (*map-option g y*)
by(*simp add*: *zip-map-option1 zip-map-option2 option.map-comp apfst-def apsnd-def o-def prod.map-comp*)

lemma *zip-conv-bind-option*:
zip-option x y = *Option.bind x* (λ*x*. *Option.bind y* (λ*y*. *Some* (*x*, *y*)))
by(*cases* (*x*, *y*) *rule*: *zip-option.cases*) *simp-all*

lemma *zip-option-parametric* [*transfer-rule*]:
includes *lifting-syntax* **shows**
(*rel-option R* ===> *rel-option Q* ===> *rel-option* (*rel-prod R Q*)) *zip-option zip-option*
unfolding *zip-conv-bind-option*[*abs-def*] **by** *transfer-prover*

lemma *rel-option-eqI* [*simp*]: *rel-option* (=) *x x*
by(*simp add*: *option.rel-eq*)

1.5.7 Binary supremum on *'a option*

primrec *sup-option* :: *'a option* ⇒ *'a option* ⇒ *'a option*
where
sup-option x None = *x*
| *sup-option x* (*Some y*) = (*Some y*)

lemma *sup-option-idem* [*simp*]: *sup-option x x* = *x*
by(*cases x*) *simp-all*

lemma *sup-option-assoc*: *sup-option* (*sup-option x y*) *z* = *sup-option x* (*sup-option y z*)
by(*cases z*) *simp-all*

lemma *sup-option-left-idem*: *sup-option x* (*sup-option x y*) = *sup-option x y*
by(*rewrite sup-option-assoc*[*symmetric*])(*simp*)

lemmas *sup-option-ai* = *sup-option-assoc sup-option-left-idem*

lemma *sup-option-None* [*simp*]: *sup-option None y* = *y*
by(*cases y*) *simp-all*

1.5.8 Restriction on $'a$ *option*

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

lemma *set-enforce-option* [*simp*]: *set-option* (*enforce-option P x*) = $\{a \in$ *set-option* *x*. *P a*$\}$
by(*cases x*) *auto*

lemma *enforce-map-option*: *enforce-option P* (*map-option f x*) = *map-option f* (*enforce-option* (*P* $\circ$ *f*) *x*)
by(*cases x*) *auto*

lemma *enforce-bind-option* [*simp*]:
enforce-option P (*Option.bind x f*) = *Option.bind x* (*enforce-option P* $\circ$ *f*)
by(*cases x*) *auto*

lemma *enforce-option-alt-def*:
enforce-option P x = *Option.bind x* (λa. *Option.bind* (*assert-option* (*P a*)) (λ- :: *unit*. *Some a*))
by(*cases x*) *simp-all*

lemma *enforce-option-eq-None-iff* [*simp*]:
enforce-option P x = *None* $\longleftrightarrow$ ($\forall$ *a*. *x* = *Some a* $\longrightarrow$ $\neg$ *P a*)
by(*cases x*) *auto*

lemma *enforce-option-eq-Some-iff* [*simp*]:
enforce-option P x = *Some y* $\longleftrightarrow$ *x* = *Some y* $\wedge$ *P y*
by(*cases x*) *auto*

lemma *Some-eq-enforce-option-iff* [*simp*]:
Some y = *enforce-option P x* $\longleftrightarrow$ *x* = *Some y* $\wedge$ *P y*
by(*cases x*) *auto*

lemma *enforce-option-top* [*simp*]: *enforce-option* $\top$ = *id*
by(*rule ext*; *rename-tac x*; *case-tac x*; *simp*)

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

lemma *enforce-option-bot* [*simp*]: *enforce-option* $\bot$ = (λ-. *None*)
by(*simp add*: *fun-eq-iff*)

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

lemma *enforce-pred-id-option*: *pred-option P x* $\Longrightarrow$ *enforce-option P x* = *x*
by(*cases x*) *auto*

1.5.9 Maps

lemma *map-add-apply*: $(m1 ++ m2)\ x = sup\text{-}option\ (m1\ x)\ (m2\ x)$
by(*simp add*: *map-add-def split*: *option.split*)

lemma *map-le-map-upd2*: $[\![\ f \subseteq_m g;\ \bigwedge y'.\ f\ x = Some\ y' \Longrightarrow y' = y\ ]\!] \Longrightarrow f \subseteq_m g(x \mapsto y)$
by(*cases* $x \in dom\ f$)(*auto simp add*: *map-le-def Ball-def*)

lemma *eq-None-iff-not-dom*: $f\ x = None \longleftrightarrow x \notin dom\ f$
by *auto*

lemma *card-ran-le-dom*: $finite\ (dom\ m) \Longrightarrow card\ (ran\ m) \leq card\ (dom\ m)$
by(*simp add*: *ran-alt-def card-image-le*)

lemma *dom-subset-ran-iff*:
  assumes *finite* (*ran m*)
  shows $dom\ m \subseteq ran\ m \longleftrightarrow dom\ m = ran\ m$
proof
  assume *le*: $dom\ m \subseteq ran\ m$
  then have $card\ (dom\ m) \leq card\ (ran\ m)$ **by**(*simp add*: *card-mono assms*)
  moreover have $card\ (ran\ m) \leq card\ (dom\ m)$ **by**(*simp add*: *finite-subset*[*OF le assms*] *card-ran-le-dom*)
  ultimately show $dom\ m = ran\ m$ **using** *card-subset-eq*[*OF assms le*] **by** *simp*
qed *simp*

We need a polymorphic constant for the empty map such that *transfer-prover* can use a custom transfer rule for *Map.empty*

definition *Map-empty* **where** [*simp*]: $Map\text{-}empty \equiv Map.empty$

lemma *map-le-Some1D*: $[\![\ m \subseteq_m m';\ m\ x = Some\ y\ ]\!] \Longrightarrow m'\ x = Some\ y$
by(*auto simp add*: *map-le-def Ball-def*)

lemma *map-le-fun-upd2*: $[\![\ f \subseteq_m g;\ x \notin dom\ f\ ]\!] \Longrightarrow f \subseteq_m g(x := y)$
by(*auto simp add*: *map-le-def*)

lemma *map-eqI*: $\forall x \in dom\ m \cup dom\ m'.\ m\ x = m'\ x \Longrightarrow m = m'$
by(*auto simp add*: *fun-eq-iff domIff intro*: *option.expand*)

1.6 Countable

lemma *countable-lfp*:
  assumes *step*: $\bigwedge Y.\ countable\ Y \Longrightarrow countable\ (F\ Y)$
  and *cont*: *Order-Continuity.sup-continuous F*
  shows *countable* (*lfp F*)
by(*subst sup-continuous-lfp*[*OF cont*])(*simp add*: *countable-funpow*[*OF step*])

lemma *countable-lfp-apply*:
  assumes *step*: $\bigwedge Y\ x.\ (\bigwedge x.\ countable\ (Y\ x)) \Longrightarrow countable\ (F\ Y\ x)$
  and *cont*: *Order-Continuity.sup-continuous F*

shows *countable* (*lfp F x*)
proof –
{ fix *n*
have $\bigwedge x$. *countable* ((*F* ^^ *n*) *bot x*)
by(*induct n*)(*auto intro*: *step*) **}**
thus *?thesis* **using** *cont* **by**(*simp add*: *sup-continuous-lfp*)
qed

1.7 Extended naturals

lemma *idiff-enat-eq-enat-iff*: $x - \textit{enat } n = \textit{enat } m \longleftrightarrow (\exists k.\ x = \textit{enat } k \wedge k - n = m)$
by (*cases x*) *simp-all*

lemma *eSuc-SUP*: $A \neq \{\} \Longrightarrow \textit{eSuc } (\bigsqcup (f \text{ ` } A)) = (\bigsqcup x \in A.\ \textit{eSuc } (f\ x))$
by (*subst eSuc-Sup*) (*simp-all add*: *image-comp*)

lemma *ereal-of-enat-1*: *ereal-of-enat 1* = *ereal 1*
by (*simp add*: *one-enat-def*)

lemma *ennreal-real-conv-ennreal-of-enat*: *ennreal* (*real n*) = *ennreal-of-enat n*
by (*simp add*: *ennreal-of-nat-eq-real-of-nat*)

lemma *enat-add-sub-same2*: $b \neq \infty \Longrightarrow a + b - b = (a :: \textit{enat})$
by (*cases a*; *cases b*) *simp-all*

lemma *enat-sub-add*: $y \leq x \Longrightarrow x - y + z = x + z - (y :: \textit{enat})$
by (*cases x*; *cases y*; *cases z*) *simp-all*

lemma *SUP-enat-eq-0-iff* [*simp*]: $\bigsqcup (f \text{ ` } A) = (0 :: \textit{enat}) \longleftrightarrow (\forall x \in A.\ f\ x = 0)$
by (*simp add*: *bot-enat-def* [*symmetric*])

lemma *SUP-enat-add-left*:
assumes $I \neq \{\}$
shows (*SUP i∈I. f i + c :: enat*) = (*SUP i∈I. f i*) + *c* (**is** *?lhs* = *?rhs*)
proof(*cases c*, *rule antisym*)
case (*enat n*)
show *?lhs* ≤ *?rhs* **by**(*auto 4 3 intro*: *SUP-upper intro*: *SUP-least*)
have (*SUP i∈I. f i*) ≤ *?lhs* − *c* **using** *enat*
by(*auto simp add*: *enat-add-sub-same2 intro*!: *SUP-least order-trans*[*OF - SUP-upper*[*THEN enat-minus-mono1*]])
note *add-right-mono*[*OF this*, *of c*]
also have . . . + *c* ≤ *?lhs* **using** *assms*
by(*subst enat-sub-add*)(*auto intro*: *SUP-upper2 simp add*: *enat-add-sub-same2 enat*)
finally show *?rhs* ≤ *?lhs* **.**
qed(*simp add*: *assms SUP-constant*)

lemma *SUP-enat-add-right*:

assumes $I \neq \{\}$
shows $(SUP\ i{\in}I.\ c + f\ i :: enat) = c + (SUP\ i{\in}I.\ f\ i)$
using *SUP-enat-add-left*[*OF assms*, *of f c*]
by(*simp add*: *add.commute*)

lemma *iadd-SUP-le-iff*: $n + (SUP\ x{\in}A.\ f\ x :: enat) \leq y \longleftrightarrow (if\ A = \{\}\ then\ n \leq y\ else\ \forall x{\in}A.\ n + f\ x \leq y)$
by(*simp add*: *bot-enat-def SUP-enat-add-right*[*symmetric*] *SUP-le-iff*)

lemma *SUP-iadd-le-iff*: $(SUP\ x{\in}A.\ f\ x :: enat) + n \leq y \longleftrightarrow (if\ A = \{\}\ then\ n \leq y\ else\ \forall x{\in}A.\ f\ x + n \leq y)$
using *iadd-SUP-le-iff*[*of n f A y*] **by**(*simp add*: *add.commute*)

1.8 Extended non-negative reals

lemma (**in** *finite-measure*) *nn-integral-indicator-neq-infty*:
$f -` A \in sets\ M \Longrightarrow (\int^+ x.\ indicator\ A\ (f\ x)\ \partial M) \neq \infty$
unfolding *ennreal-indicator*[*symmetric*]
apply(*rule integrableD*)
apply(*rule integrable-const-bound*[**where** *B=1*])
apply(*simp-all add*: *indicator-vimage*[*symmetric*])
done

lemma (**in** *finite-measure*) *nn-integral-indicator-neq-top*:
$f -` A \in sets\ M \Longrightarrow (\int^+ x.\ indicator\ A\ (f\ x)\ \partial M) \neq \top$
by(*drule nn-integral-indicator-neq-infty*) *simp*

lemma *nn-integral-indicator-map*:
assumes [*measurable*]: $f \in measurable\ M\ N\ \{x{\in}space\ N.\ P\ x\} \in sets\ N$
shows $(\int^+ x.\ indicator\ \{x{\in}space\ N.\ P\ x\}\ (f\ x)\ \partial M) = emeasure\ M\ \{x{\in}space\ M.\ P\ (f\ x)\}$
using *assms*(*1*)[*THEN measurable-space*]
by (*subst nn-integral-indicator*[*symmetric*])
(*auto intro*!: *nn-integral-cong split*: *split-indicator simp del*: *nn-integral-indicator*)

1.9 BNF material

lemma *transp-rel-fun*: $[\![\ is\text{-}equality\ Q;\ transp\ R\ ]\!] \Longrightarrow transp\ (rel\text{-}fun\ Q\ R)$
by(*rule transpI*)(*auto dest*: *transpD rel-funD simp add*: *is-equality-def*)

lemma *rel-fun-inf*: $inf\ (rel\text{-}fun\ Q\ R)\ (rel\text{-}fun\ Q\ R') = rel\text{-}fun\ Q\ (inf\ R\ R')$
by(*rule antisym*)(*auto elim*: *rel-fun-mono dest*: *rel-funD*)

lemma *reflp-fun1*: **includes** *lifting-syntax* **shows** $[\![\ is\text{-}equality\ A;\ reflp\ B\ ]\!] \Longrightarrow reflp\ (A ===> B)$
by(*simp add*: *reflp-def rel-fun-def is-equality-def*)

lemma *type-copy-id'*: *type-definition* $(\lambda x.\ x)\ (\lambda x.\ x)$ *UNIV*
by *unfold-locales simp-all*

```
lemma type-copy-id: type-definition id id UNIV
by(simp add: id-def type-copy-id′)

lemma GrpE [cases pred]:
  assumes BNF-Def.Grp A f x y
  obtains (Grp) y = f x x ∈ A
using assms
by(simp add: Grp-def)

lemma rel-fun-Grp-copy-Abs:
  includes lifting-syntax
  assumes type-definition Rep Abs A
  shows rel-fun (BNF-Def.Grp A Abs) (BNF-Def.Grp B g) = BNF-Def.Grp {f.
f ‘ A ⊆ B} (Rep −−−> g)
proof −
  interpret type-definition Rep Abs A by fact
  show ?thesis
    by(auto simp add: rel-fun-def Grp-def fun-eq-iff Abs-inverse Rep-inverse intro!:
Rep)
qed

lemma rel-set-Grp:
  rel-set (BNF-Def.Grp A f) = BNF-Def.Grp {B. B ⊆ A} (image f)
by(auto simp add: rel-set-def BNF-Def.Grp-def fun-eq-iff)

lemma rel-set-comp-Grp:
  rel-set R = (BNF-Def.Grp {x. x ⊆ {(x, y). R x y}} ((‘) fst))⁻¹⁻¹ OO BNF-Def.Grp
{x. x ⊆ {(x, y). R x y}} ((‘) snd)
apply(auto 4 4 del: ext intro!: ext simp add: BNF-Def.Grp-def intro!: rel-setI intro:
rev-bexI)
apply(simp add: relcompp-apply)
subgoal for A B
  apply(rule exI[where x=A × B ∩ {(x, y). R x y}])
  apply(auto 4 3 dest: rel-setD1 rel-setD2 intro: rev-image-eqI)
  done
done

lemma Domainp-Grp: Domainp (BNF-Def.Grp A f) = (λx. x ∈ A)
by(auto simp add: fun-eq-iff Grp-def)

lemma pred-prod-conj [simp]:
  shows pred-prod-conj1: ⋀P Q R. pred-prod (λx. P x ∧ Q x) R = (λx. pred-prod
P R x ∧ pred-prod Q R x)
  and pred-prod-conj2: ⋀P Q R. pred-prod P (λx. Q x ∧ R x) = (λx. pred-prod P
Q x ∧ pred-prod P R x)
by(auto simp add: pred-prod.simps)

lemma pred-sum-conj [simp]:
  shows pred-sum-conj1: ⋀P Q R. pred-sum (λx. P x ∧ Q x) R = (λx. pred-sum
```

P R x ∧ *pred-sum Q R x*)
and *pred-sum-conj2*: $\bigwedge$*P Q R*. *pred-sum P* (λ*x*. *Q x* ∧ *R x*) = (λ*x*. *pred-sum P Q x* ∧ *pred-sum P R x*)
by(*auto simp add*: *pred-sum.simps fun-eq-iff*)

lemma *pred-list-conj* [*simp*]: *list-all* (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *list-all P x* ∧ *list-all Q x*)
by(*auto simp add*: *list-all-def*)

lemma *pred-prod-top* [*simp*]:
pred-prod (λ-. *True*) (λ-. *True*) = (λ-. *True*)
by(*simp add*: *pred-prod.simps fun-eq-iff*)

lemma *rel-fun-conversep*: **includes** *lifting-syntax* **shows**
(*A*^−−*1* ===> *B*^−−*1*) = (*A* ===> *B*)^−−*1*
by(*auto simp add*: *rel-fun-def fun-eq-iff*)

lemma *left-unique-Grp* [*iff*]:
left-unique (*BNF-Def.Grp A f*) ⟷ *inj-on f A*
unfolding *Grp-def left-unique-def* **by**(*auto simp add*: *inj-on-def*)

lemma *right-unique-Grp* [*simp*, *intro*!]: *right-unique* (*BNF-Def.Grp A f*)
by(*simp add*: *Grp-def right-unique-def*)

lemma *bi-unique-Grp* [*iff*]:
bi-unique (*BNF-Def.Grp A f*) ⟷ *inj-on f A*
by(*simp add*: *bi-unique-alt-def*)

lemma *left-total-Grp* [*iff*]:
left-total (*BNF-Def.Grp A f*) ⟷ *A* = *UNIV*
by(*auto simp add*: *left-total-def Grp-def*)

lemma *right-total-Grp* [*iff*]:
right-total (*BNF-Def.Grp A f*) ⟷ *f* ' *A* = *UNIV*
by(*auto simp add*: *right-total-def BNF-Def.Grp-def image-def*)

lemma *bi-total-Grp* [*iff*]:
bi-total (*BNF-Def.Grp A f*) ⟷ *A* = *UNIV* ∧ *surj f*
by(*auto simp add*: *bi-total-alt-def*)

lemma *left-unique-vimage2p* [*simp*]:
⟦ *left-unique P*; *inj f* ⟧ ⟹ *left-unique* (*BNF-Def.vimage2p f g P*)
unfolding *vimage2p-Grp* **by**(*intro left-unique-OO*) *simp-all*

lemma *right-unique-vimage2p* [*simp*]:
⟦ *right-unique P*; *inj g* ⟧ ⟹ *right-unique* (*BNF-Def.vimage2p f g P*)
unfolding *vimage2p-Grp* **by**(*intro right-unique-OO*) *simp-all*

lemma *bi-unique-vimage2p* [*simp*]:

$\llbracket$ *bi-unique P*; *inj f*; *inj g* $\rrbracket \Longrightarrow$ *bi-unique* (*BNF-Def.vimage2p f g P*)
unfolding *bi-unique-alt-def* **by** *simp*

lemma *left-total-vimage2p* [*simp*]:
$\llbracket$ *left-total P*; *surj g* $\rrbracket \Longrightarrow$ *left-total* (*BNF-Def.vimage2p f g P*)
unfolding *vimage2p-Grp* **by**(*intro left-total-OO*) *simp-all*

lemma *right-total-vimage2p* [*simp*]:
$\llbracket$ *right-total P*; *surj f* $\rrbracket \Longrightarrow$ *right-total* (*BNF-Def.vimage2p f g P*)
unfolding *vimage2p-Grp* **by**(*intro right-total-OO*) *simp-all*

lemma *bi-total-vimage2p* [*simp*]:
$\llbracket$ *bi-total P*; *surj f*; *surj g* $\rrbracket \Longrightarrow$ *bi-total* (*BNF-Def.vimage2p f g P*)
unfolding *bi-total-alt-def* **by** *simp*

lemma *vimage2p-eq* [*simp*]:
inj f $\Longrightarrow$ *BNF-Def.vimage2p f f* (=) = (=)
by(*auto simp add*: *vimage2p-def fun-eq-iff inj-on-def*)

lemma *vimage2p-conversep*: *BNF-Def.vimage2p f g R^−−1* = (*BNF-Def.vimage2p g f R*)*^−−1*
by(*simp add*: *vimage2p-def fun-eq-iff*)

lemma *rel-fun-refl*: $\llbracket$ *A* ≤ (=); (=) ≤ *B* $\rrbracket \Longrightarrow$ (=) ≤ *rel-fun A B*
by(*subst fun.rel-eq*[*symmetric*])(*rule fun-mono*)

lemma *rel-fun-mono-strong*:
$\llbracket$ *rel-fun A B f g*; *A′* ≤ *A*; $\bigwedge$*x y*. $\llbracket$ *x* ∈ *f* ‘ {*x*. *Domainp A′ x*}; *y* ∈ *g* ‘ {*x*. *Rangep A′ x*}; *B x y* $\rrbracket \Longrightarrow$ *B′ x y* $\rrbracket \Longrightarrow$ *rel-fun A′ B′ f g*
by(*auto simp add*: *rel-fun-def*) *fastforce*

lemma *rel-fun-refl-strong*:
assumes *A* ≤ (=) $\bigwedge$*x*. *x* ∈ *f* ‘ {*x*. *Domainp A x*} $\Longrightarrow$ *B x x*
shows *rel-fun A B f f*
proof −
have *rel-fun* (=) (=) *f f* **by**(*simp add*: *rel-fun-eq*)
then show *?thesis* **using** *assms*(*1*)
by(*rule rel-fun-mono-strong*) (*auto intro*: *assms*(*2*))
qed

lemma *Grp-iff*: *BNF-Def.Grp B g x y* ⟷ *y* = *g x* ∧ *x* ∈ *B* **by**(*simp add*: *Grp-def*)

lemma *Rangep-Grp*: *Rangep* (*BNF-Def.Grp A f*) = (λ*x*. *x* ∈ *f* ‘ *A*)
by(*auto simp add*: *fun-eq-iff Grp-iff*)

lemma *rel-fun-Grp*:
rel-fun (*BNF-Def.Grp UNIV h*)$^{-1-1}$ (*BNF-Def.Grp A g*) = *BNF-Def.Grp* {*f*. *f* ‘ *range h* ⊆ *A*} (*map-fun h g*)
by(*auto simp add*: *rel-fun-def fun-eq-iff Grp-iff*)

1.10 Transfer and lifting material

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

lemma *monotone-parametric* [*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-total A*
  shows ((*A* ===> *A* ===> (=)) ===> (*B* ===> *B* ===> (=)) ===> (*A* ===> *B*) ===> (=)) *monotone monotone*
unfolding *monotone-def*[*abs-def*] **by** *transfer-prover*

lemma *fun-ord-parametric* [*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-total C*
  shows ((*A* ===> *B* ===> (=)) ===> (*C* ===> *A*) ===> (*C* ===> *B*) ===> (=)) *fun-ord fun-ord*
unfolding *fun-ord-def*[*abs-def*] **by** *transfer-prover*

lemma *Plus-parametric* [*transfer-rule*]:
  (*rel-set A* ===> *rel-set B* ===> *rel-set* (*rel-sum A B*)) (<+>) (<+>)
unfolding *Plus-def*[*abs-def*] **by** *transfer-prover*

lemma *pred-fun-parametric* [*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-total A*
  shows ((*A* ===> (=)) ===> (*B* ===> (=)) ===> (*A* ===> *B*) ===> (=)) *pred-fun pred-fun*
unfolding *pred-fun-def* **by**(*transfer-prover*)

lemma *rel-fun-eq-OO*: ((=) ===> *A*) *OO* ((=) ===> *B*) = ((=) ===> *A OO B*)
by(*clarsimp simp add*: *rel-fun-def fun-eq-iff relcompp.simps*) *metis*

end

lemma *Quotient-set-rel-eq*:
  includes *lifting-syntax*
  assumes *Quotient R Abs Rep T*
  shows (*rel-set T* ===> *rel-set T* ===> (=)) (*rel-set R*) (=)
proof(*rule rel-funI iffI*)+
  fix *A B C D*
  assume *AB*: *rel-set T A B* **and** *CD*: *rel-set T C D*
  have ∗: ⋀*x y*. *R x y* = (*T x* (*Abs x*) ∧ *T y* (*Abs y*) ∧ *Abs x* = *Abs y*)
    ⋀*a b*. *T a b* ⟹ *Abs a* = *b*
    using *assms* **unfolding** *Quotient-alt-def* **by** *simp-all*

  { assume [*simp*]: *B* = *D*
    thus *rel-set R A C*
      by(*auto 4 4 intro*!: *rel-setI dest*: *rel-setD1*[*OF AB*, *simplified*] *rel-setD2*[*OF AB*, *simplified*] *rel-setD2*[*OF CD*] *rel-setD1*[*OF CD*] *simp add*: ∗ *elim*!: *rev-bexI*)
  next
    assume *AC*: *rel-set R A C*
    show *B* = *D*

apply *safe*
apply(*drule rel-setD2*[*OF AB*], *erule bexE*)
apply(*drule rel-setD1*[*OF AC*], *erule bexE*)
apply(*drule rel-setD1*[*OF CD*], *erule bexE*)
apply(*simp add*: ∗)
apply(*drule rel-setD2*[*OF CD*], *erule bexE*)
apply(*drule rel-setD2*[*OF AC*], *erule bexE*)
apply(*drule rel-setD1*[*OF AB*], *erule bexE*)
apply(*simp add*: ∗)
done
}
qed

lemma *Domainp-eq*: *Domainp* (=) = (λ-. *True*)
by(*simp add*: *Domainp.simps fun-eq-iff*)

lemma *rel-fun-eq-onpI*: *eq-onp* (*pred-fun P Q*) *f g* ⟹ *rel-fun* (*eq-onp P*) (*eq-onp Q*) *f g*
by(*auto simp add*: *eq-onp-def rel-fun-def*)

lemma *bi-unique-eq-onp*: *bi-unique* (*eq-onp P*)
by(*simp add*: *bi-unique-def eq-onp-def*)

lemma *rel-fun-eq-conversep*: **includes** *lifting-syntax* **shows** (A^{-1-1} ===> (=)) = (*A* ===> (=))$^{-1-1}$
by(*auto simp add*: *fun-eq-iff rel-fun-def*)

lemma *rel-fun-comp*:
⋀*f g h*. *rel-fun A B* (*f* ∘ *g*) *h* = *rel-fun A* (λ*x*. *B* (*f x*)) *g h*
⋀*f g h*. *rel-fun A B f* (*g* ∘ *h*) = *rel-fun A* (λ*x y*. *B x* (*g y*)) *f h*
by(*auto simp add*: *rel-fun-def*)

lemma *rel-fun-map-fun1*: *rel-fun* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *A f g* ⟹ *rel-fun* (=) *A* (*map-fun h id f*) *g*
by(*auto simp add*: *rel-fun-def Grp-def*)

lemma *map-fun2-id*: *map-fun f g x* = *g* ∘ *map-fun f id x*
by(*simp add*: *map-fun-def o-assoc*)

lemma *map-fun-id2-in*: *map-fun g h f* = *map-fun g id* (*h* ∘ *f*)
by(*simp add*: *map-fun-def*)

lemma *Domainp-rel-fun-le*: *Domainp* (*rel-fun A B*) ≤ *pred-fun* (*Domainp A*) (*Domainp B*)
by(*auto dest*: *rel-funD*)

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

lemma
  assumes *fg*: *rel-fun* (*A OO A′*) *B f g*
    and *A*: *left-unique A right-total A*
    and *A′*: *right-unique A′ left-total A′*
  shows *rel-witness-fun1*: *rel-fun A* (λx (x', y). $x = x' \wedge B\ x'\ y$) *f* (*rel-witness-fun A A′* (*f*, *g*))
    and *rel-witness-fun2*: *rel-fun A′* (λ(x, y') y. $y = y' \wedge B\ x\ y'$) (*rel-witness-fun A A′* (*f*, *g*)) *g*
proof (*goal-cases*)
  case *1*
  have $A\ x\ y \Longrightarrow f\ x = f$ (*THE a. A a y*) $\wedge$ *B* (*f* (*THE a. A a y*)) (*g* (*The* (*A′ y*))) **for** *x y*
    by(*rule left-totalE*[*OF A′*(*2*)]; *erule meta-allE*[*of - y*]; *erule exE*; *frule* (*1*) *fg*[*THEN rel-funD*, *OF relcomppI*])
      (*auto intro*!: *arg-cong*[**where** *f*=*f*] *arg-cong*[**where** *f*=*g*] *rel-funI the-equality the-equality*[*symmetric*] *dest*: *left-uniqueD*[*OF A*(*1*)] *right-uniqueD*[*OF A′*(*1*)] *elim*!: *arg-cong2*[**where** *f*=*B*, *THEN iffD2*, *rotated −1*])

  with *1* **show** *?case* **by**(*clarsimp simp add*: *rel-fun-def rel-witness-fun-def*)
next
  case *2*
  have $A'\ x\ y \Longrightarrow g\ y = g$ (*The* (*A′ x*)) $\wedge$ *B* (*f* (*THE a. A a x*)) (*g* (*The* (*A′ x*))) **for** *x y*
    by(*rule right-totalE*[*OF A*(*2*), *of x*]; *frule* (*1*) *fg*[*THEN rel-funD*, *OF relcomppI*])
      (*auto intro*!: *arg-cong*[**where** *f*=*f*] *arg-cong*[**where** *f*=*g*] *rel-funI the-equality the-equality*[*symmetric*] *dest*: *left-uniqueD*[*OF A*(*1*)] *right-uniqueD*[*OF A′*(*1*)] *elim*!: *arg-cong2*[**where** *f*=*B*, *THEN iffD2*, *rotated −1*])

  with *2* **show** *?case* **by**(*clarsimp simp add*: *rel-fun-def rel-witness-fun-def*)
qed

lemma *rel-witness-fun-eq* [*simp*]: *rel-witness-fun* (=) (=) (*f*, *g*) = (λx. (*f x*, *g x*))
  by(*simp add*: *rel-witness-fun-def*)

1.11 Arithmetic

lemma *abs-diff-triangle-ineq2*: $|a - b$ :: - :: *ordered-ab-group-add-abs*$| \le |a - c| + |c - b|$
by(*rule order-trans*[*OF - abs-diff-triangle-ineq*]) *simp*

lemma (**in** *ordered-ab-semigroup-add*) *add-left-mono-trans*:
  $[\![\ x \le a + b;\ b \le c\ ]\!] \Longrightarrow x \le a + c$
by(*erule order-trans*)(*rule add-left-mono*)

lemma *of-nat-le-one-cancel-iff* [*simp*]:
  fixes *n* :: *nat* **shows** *real* $n \le 1 \longleftrightarrow n \le 1$
by *linarith*

lemma (**in** *linordered-semidom*) *mult-right-le*: $c \le 1 \Longrightarrow 0 \le a \Longrightarrow c * a \le a$
by(*subst mult.commute*)(*rule mult-left-le*)

1.12 Chain-complete partial orders and *partial-function*

lemma *fun-ordD*: *fun-ord ord f g* $\Longrightarrow$ *ord* (*f x*) (*g x*)
by(*simp add*: *fun-ord-def*)

lemma *parallel-fixp-induct-strong*:
  assumes *ccpo1*: *class.ccpo luba orda* (*mk-less orda*)
  and *ccpo2*: *class.ccpo lubb ordb* (*mk-less ordb*)
  and *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*λx. P* (*fst x*) (*snd x*))
  and *f*: *monotone orda orda f*
  and *g*: *monotone ordb ordb g*
  and *bot*: *P* (*luba* {}) (*lubb* {})
  and *step*: ⋀*x y*. ⟦ *orda x* (*ccpo.fixp luba orda f*); *ordb y* (*ccpo.fixp lubb ordb g*); *P x y* ⟧ $\Longrightarrow$ *P* (*f x*) (*g y*)
  shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
proof −
  let *?P=λx y. orda x* (*ccpo.fixp luba orda f*) ∧ *ordb y* (*ccpo.fixp lubb ordb g*) ∧ *P x y*
  show *?thesis* **using** *ccpo1 ccpo2 - f g*
  proof(*rule parallel-fixp-induct*[**where** *P=?P*, *THEN conjunct2*, *THEN conjunct2*])
    note [*cont-intro*] =
      admissible-leI[*OF ccpo1*] *ccpo.mcont-const*[*OF ccpo1*]
      admissible-leI[*OF ccpo2*] *ccpo.mcont-const*[*OF ccpo2*]
    show *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*λxy. ?P* (*fst xy*) (*snd xy*))
      using *adm* **by** *simp*
    show *?P* (*luba* {}) (*lubb* {}) **using** *bot* **by**(*auto intro*: *ccpo.ccpo-Sup-least ccpo1 ccpo2 chain-empty*)
    show *?P* (*f x*) (*g y*) **if** *?P x y* **for** *x y* **using** *that*
      apply(*subst ccpo.fixp-unfold*[*OF ccpo1 f*])
      apply(*subst ccpo.fixp-unfold*[*OF ccpo2 g*])
      apply(*auto intro*: *step monotoneD*[*OF f*] *monotoneD*[*OF g*])
      done
  qed
qed

lemma *parallel-fixp-induct-strong-uc*:
  assumes *a*: *partial-function-definitions orda luba*
  and *b*: *partial-function-definitions ordb lubb*
  and *F*: ⋀*x. monotone* (*fun-ord orda*) *orda* (*λf. U1* (*F* (*C1 f*)) *x*)
  and *G*: ⋀*y. monotone* (*fun-ord ordb*) *ordb* (*λg. U2* (*G* (*C2 g*)) *y*)
  and *eq1*: *f* ≡ *C1* (*ccpo.fixp* (*fun-lub luba*) (*fun-ord orda*) (*λf. U1* (*F* (*C1 f*))))
  and *eq2*: *g* ≡ *C2* (*ccpo.fixp* (*fun-lub lubb*) (*fun-ord ordb*) (*λg. U2* (*G* (*C2 g*))))
  and *inverse*: ⋀*f. U1* (*C1 f*) = *f*
  and *inverse2*: ⋀*g. U2* (*C2 g*) = *g*

and *adm*: *ccpo.admissible* (*prod-lub* (*fun-lub luba*) (*fun-lub lubb*)) (*rel-prod* (*fun-ord orda*) (*fun-ord ordb*)) (λx. *P* (*fst x*) (*snd x*))
and *bot*: *P* (λ-. *luba* {}) (λ-. *lubb* {})
and *step*: $\bigwedge f'\ g'$. ⟦ $\bigwedge x$. *orda* (*U1 f' x*) (*U1 f x*); $\bigwedge y$. *ordb* (*U2 g' y*) (*U2 g y*); *P* (*U1 f'*) (*U2 g'*) ⟧ $\Longrightarrow$ *P* (*U1* (*F f'*)) (*U2* (*G g'*))
shows *P* (*U1 f*) (*U2 g*)
apply(*unfold eq1 eq2 inverse inverse2*)
apply(*rule parallel-fixp-induct-strong*[*OF partial-function-definitions.ccpo*[*OF a*] *partial-function-definitions.ccpo*[*OF b*] *adm*])
using *F* **apply**(*simp add*: *monotone-def fun-ord-def*)
using *G* **apply**(*simp add*: *monotone-def fun-ord-def*)
apply(*simp add*: *fun-lub-def bot*)
apply(*rule step*; *simp add*: *inverse inverse2 eq1 eq2 fun-ordD*)
done

lemmas *parallel-fixp-induct-strong-1-1* = *parallel-fixp-induct-strong-uc*[
of - - - - λx. *x* - λx. *x* λx. *x* - λx. *x*,
OF - - - - - - *refl refl*]

lemmas *parallel-fixp-induct-strong-2-2* = *parallel-fixp-induct-strong-uc*[
of - - - - *case-prod* - *curry case-prod* - *curry*,
where *P*=$\lambda f\ g$. *P* (*curry f*) (*curry g*),
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - - *refl refl*,
split-format (*complete*), *unfolded prod.case*]
for *P*

lemma *fixp-induct-option'*: — Stronger induction rule
fixes *F* :: *'c* ⇒ *'c* **and**
U :: *'c* ⇒ *'b* ⇒ *'a option* **and**
C :: (*'b* ⇒ *'a option*) ⇒ *'c* **and**
P :: *'b* ⇒ *'a* ⇒ *bool*
assumes *mono*: $\bigwedge x$. *mono-option* (λf. *U* (*F* (*C f*)) *x*)
assumes *eq*: *f* ≡ *C* (*ccpo.fixp* (*fun-lub* (*flat-lub None*)) (*fun-ord option-ord*) (λf. *U* (*F* (*C f*))))
assumes *inverse2*: $\bigwedge f$. *U* (*C f*) = *f*
assumes *step*: $\bigwedge g\ x\ y$. ⟦ $\bigwedge x\ y$. *U g x* = *Some y* $\Longrightarrow$ *P x y*; *U* (*F g*) *x* = *Some y*; $\bigwedge x$. *option-ord* (*U g x*) (*U f x*) ⟧ $\Longrightarrow$ *P x y*
assumes *defined*: *U f x* = *Some y*
shows *P x y*
using *step defined option.fixp-strong-induct-uc*[*of U F C*, *OF mono eq inverse2 option-admissible*, *of P*]
unfolding *fun-lub-def flat-lub-def fun-ord-def*
by(*simp* (*no-asm-use*)) *blast*

declaration ‹*Partial-Function.init option'* @{*term option.fixp-fun*}
@{*term option.mono-body*} @{*thm option.fixp-rule-uc*} @{*thm option.fixp-induct-uc*}
(*SOME* @{*thm fixp-induct-option'*})›

```
lemma bot-fun-least [simp]: (λ-. bot :: 'a :: order-bot) ≤ x
by(fold bot-fun-def) simp

lemma fun-ord-conv-rel-fun: fun-ord = rel-fun (=)
by(simp add: fun-ord-def fun-eq-iff rel-fun-def)

inductive finite-chains :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  for ord
where finite-chainsI: (⋀Y. Complete-Partial-Order.chain ord Y ⟹ finite Y)
⟹ finite-chains ord

lemma finite-chainsD: ⟦ finite-chains ord; Complete-Partial-Order.chain ord Y ⟧
⟹ finite Y
by(rule finite-chains.cases)

lemma finite-chains-flat-ord [simp, intro!]: finite-chains (flat-ord x)
proof
  fix Y
  assume chain: Complete-Partial-Order.chain (flat-ord x) Y
  show finite Y
  proof(cases ∃ y ∈ Y. y ≠ x)
    case True
    then obtain y where y: y ∈ Y and yx: y ≠ x by blast
    hence Y ⊆ {x, y} by(auto dest: chainD[OF chain] simp add: flat-ord-def)
    thus ?thesis by(rule finite-subset) simp
  next
    case False
    hence Y ⊆ {x} by auto
    thus ?thesis by(rule finite-subset) simp
  qed
qed

lemma mcont-finite-chains:
  assumes finite: finite-chains ord
  and mono: monotone ord ord' f
  and ccpo: class.ccpo lub ord (mk-less ord)
  and ccpo': class.ccpo lub' ord' (mk-less ord')
  shows mcont lub ord lub' ord' f
proof(intro mcontI contI)
  fix Y
  assume chain: Complete-Partial-Order.chain ord Y and Y: Y ≠ {}
  from finite chain have fin: finite Y by(rule finite-chainsD)
  from ccpo chain fin Y have lub: lub Y ∈ Y by(rule ccpo.in-chain-finite)

  interpret ccpo': ccpo lub' ord' mk-less ord' by(rule ccpo')

  have chain': Complete-Partial-Order.chain ord' (f ` Y) using chain
    by(rule chain-imageI)(rule monotoneD[OF mono])
```

have *ord′* (*f* (*lub Y*)) (*lub′* (*f* ‘ *Y*)) **using** *chain′*
by(*rule ccpo′.ccpo-Sup-upper*)(*simp add*: *lub*)
moreover
have *ord′* (*lub′* (*f* ‘ *Y*)) (*f* (*lub Y*)) **using** *chain′*
by(*rule ccpo′.ccpo-Sup-least*)(*blast intro*: *monotoneD*[*OF mono*] *ccpo.ccpo-Sup-upper*[*OF ccpo chain*])
ultimately show *f* (*lub Y*) = *lub′* (*f* ‘ *Y*) **by**(*rule ccpo′.order.antisym*)
qed(*fact mono*)

lemma *rel-fun-curry*: **includes** *lifting-syntax* **shows**
(*A* ===> *B* ===> *C*) *f g* ⟷ (*rel-prod A B* ===> *C*) (*case-prod f*) (*case-prod g*)
by(*auto simp add*: *rel-fun-def*)

lemma (**in** *ccpo*) *Sup-image-mono*:
assumes *ccpo*: *class.ccpo luba orda lessa*
and *mono*: *monotone orda* (≤) *f*
and *chain*: *Complete-Partial-Order.chain orda A*
and *A* ≠ {}
shows *Sup* (*f* ‘ *A*) ≤ (*f* (*luba A*))
proof(*rule ccpo-Sup-least*)
from *chain* **show** *Complete-Partial-Order.chain* (≤) (*f* ‘ *A*)
by(*rule chain-imageI*)(*rule monotoneD*[*OF mono*])
fix *x*
assume *x* ∈ *f* ‘ *A*
then obtain *y* **where** *x* = *f y y* ∈ *A* **by** *blast*
from ‹*y* ∈ *A*› **have** *orda y* (*luba A*) **by**(*rule ccpo.ccpo-Sup-upper*[*OF ccpo chain*])
hence *f y* ≤ *f* (*luba A*) **by**(*rule monotoneD*[*OF mono*])
thus *x* ≤ *f* (*luba A*) **using** ‹*x* = *f y*› **by** *simp*
qed

lemma (**in** *ccpo*) *admissible-le-mono*:
assumes *monotone* (≤) (≤) *f*
shows *ccpo.admissible Sup* (≤) (λ*x*. *x* ≤ *f x*)
proof(*rule ccpo.admissibleI*)
fix *Y*
assume *chain*: *Complete-Partial-Order.chain* (≤) *Y*
and *Y*: *Y* ≠ {}
and *le* [*rule-format*]: ∀ *x*∈*Y*. *x* ≤ *f x*
have ⨆ *Y* ≤ ⨆ (*f* ‘ *Y*) **using** *chain*
by(*rule ccpo-Sup-least*)(*rule order-trans*[*OF le*]; *blast intro*!: *ccpo-Sup-upper chain-imageI*[*OF chain*] *intro*: *monotoneD*[*OF assms*])
also have ... ≤ *f* (⨆ *Y*)
by(*rule Sup-image-mono*[*OF - assms chain Y*, **where** *lessa*=(<)]) *unfold-locales*
finally show ⨆ *Y* ≤ ... .
qed

lemma (**in** *ccpo*) *fixp-induct-strong2*:
assumes *adm*: *ccpo.admissible Sup* (≤) *P*

and *mono*: *monotone* $(\leq)$ $(\leq)$ *f*
and *bot*: *P* $(\bigsqcup\{\})$
and *step*: $\bigwedge x.$ ⟦ $x \leq$ *ccpo-class.fixp f*; $x \leq f\ x$; $P\ x$ ⟧ $\Longrightarrow$ $P\ (f\ x)$
shows *P* (*ccpo-class.fixp f*)
proof(*rule fixp-strong-induct*[**where** $P=\lambda x.\ x \leq f\ x \wedge P\ x$, *THEN conjunct2*])
show *ccpo.admissible Sup* $(\leq)$ $(\lambda x.\ x \leq f\ x \wedge P\ x)$
using *admissible-le-mono adm* **by**(*rule admissible-conj*)(*rule mono*)
next
show $\bigsqcup\{\} \leq f\ (\bigsqcup\{\}) \wedge P\ (\bigsqcup\{\})$
by(*auto simp add*: *bot chain-empty intro*: *ccpo-Sup-least*)
next
fix *x*
assume $x \leq$ *ccpo-class.fixp f* $x \leq f\ x \wedge P\ x$
thus $f\ x \leq f\ (f\ x) \wedge P\ (f\ x)$
by(*auto dest*: *monotoneD*[*OF mono*] *intro*: *step*)
qed(*rule mono*)

context *partial-function-definitions* **begin**

lemma *fixp-induct-strong2-uc*:
fixes $F :: 'c \Rightarrow 'c$
and $U :: 'c \Rightarrow 'b \Rightarrow 'a$
and $C :: ('b \Rightarrow 'a) \Rightarrow 'c$
and $P :: ('b \Rightarrow 'a) \Rightarrow bool$
assumes *mono*: $\bigwedge x.$ *mono-body* $(\lambda f.\ U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C$ (*fixp-fun* $(\lambda f.\ U\ (F\ (C\ f))))$
and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
and *adm*: *ccpo.admissible lub-fun le-fun P*
and *bot*: P (λ-. *lub* $\{\}$)
and *step*: $\bigwedge f'.$ ⟦ *le-fun* $(U\ f')\ (U\ f)$; *le-fun* $(U\ f')\ (U\ (F\ f'))$; $P\ (U\ f')$ ⟧ $\Longrightarrow$
$P\ (U\ (F\ f'))$
shows $P\ (U\ f)$
unfolding *eq inverse*
apply (*rule ccpo.fixp-induct-strong2*[*OF ccpo adm*])
apply (*insert mono*, *auto simp*: *monotone-def fun-ord-def bot fun-lub-def*)[*2*]
apply (*rule-tac* $f'5=C\ x$ **in** *step*)
apply (*simp-all add*: *inverse eq*)
done

end

lemmas *parallel-fixp-induct-2-4* = *parallel-fixp-induct-uc*[
of - - - - *case-prod* - *curry* $\lambda f.$ *case-prod* (*case-prod* (*case-prod f*)) - $\lambda f.$ *curry*
(*curry* (*curry f*)),
where $P=\lambda f\ g.\ P$ (*curry f*) (*curry* (*curry* (*curry g*))),
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - - *refl refl*]
for *P*

```
lemma (in ccpo) fixp-greatest:
  assumes f: monotone (≤) (≤) f
    and ge: ⋀y. f y ≤ y ⟹ x ≤ y
  shows x ≤ ccpo.fixp Sup (≤) f
  by(rule ge)(simp add: fixp-unfold[OF f, symmetric])

lemma fixp-rolling:
  assumes class.ccpo lub1 leq1 (mk-less leq1)
    and class.ccpo lub2 leq2 (mk-less leq2)
    and f: monotone leq1 leq2 f
    and g: monotone leq2 leq1 g
  shows ccpo.fixp lub1 leq1 (λx. g (f x)) = g (ccpo.fixp lub2 leq2 (λx. f (g x)))
proof –
  interpret c1: ccpo lub1 leq1 mk-less leq1 by fact
  interpret c2: ccpo lub2 leq2 mk-less leq2 by fact
  show ?thesis
  proof(rule c1.order.antisym)
    have fg: monotone leq2 leq2 (λx. f (g x)) using f g by(rule monotone2mono-
tone) simp-all
    have gf: monotone leq1 leq1 (λx. g (f x)) using g f by(rule monotone2mono-
tone) simp-all
    show leq1 (c1.fixp (λx. g (f x))) (g (c2.fixp (λx. f (g x)))) using gf
      by(rule c1.fixp-lowerbound)(subst (2) c2.fixp-unfold[OF fg], simp)
    show leq1 (g (c2.fixp (λx. f (g x)))) (c1.fixp (λx. g (f x))) using gf
    proof(rule c1.fixp-greatest)
      fix u
      assume u: leq1 (g (f u)) u
      have leq1 (g (c2.fixp (λx. f (g x)))) (g (f u))
        by(intro monotoneD[OF g] c2.fixp-lowerbound[OF fg] monotoneD[OF f u])
      then show leq1 (g (c2.fixp (λx. f (g x)))) u using u by(rule c1.order-trans)
    qed
  qed
qed

lemma fixp-lfp-parametric-eq:
  includes lifting-syntax
  assumes f: ⋀x. lfp.mono-body (λf. F f x)
  and g: ⋀x. lfp.mono-body (λf. G f x)
  and param: ((A ===> (=)) ===> A ===> (=)) F G
  shows (A ===> (=)) (lfp.fixp-fun F) (lfp.fixp-fun G)
using f g
proof(rule parallel-fixp-induct-1-1[OF complete-lattice-partial-function-definitions
complete-lattice-partial-function-definitions - - reflexive reflexive, where P=(A ===>
(=))])
 show ccpo.admissible (prod-lub lfp.lub-fun lfp.lub-fun) (rel-prod lfp.le-fun lfp.le-fun)
(λx. (A ===> (=)) (fst x) (snd x))
    unfolding rel-fun-def by simp
  show (A ===> (=)) (λ-. ⨆{}) (λ-. ⨆{}) by auto
  show (A ===> (=)) (F f) (G g) if (A ===> (=)) f g for f g
```

using *that* **by**(*rule rel-funD*[*OF param*])
qed

lemma *mono2mono-map-option*[*THEN option.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-map-option*: *monotone option-ord option-ord* (*map-option f*)
by(*rule monotoneI*)(*auto simp add*: *flat-ord-def*)

lemma *mcont2mcont-map-option*[*THEN option.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-map-option*: *mcont* (*flat-lub None*) *option-ord* (*flat-lub None*) *option-ord* (*map-option f*)
by(*rule mcont-finite-chains*[*OF - - flat-interpretation*[*THEN ccpo*] *flat-interpretation*[*THEN ccpo*]]) *simp-all*

lemma *mono2mono-set-option* [*THEN lfp.mono2mono*]:
shows *monotone-set-option*: *monotone option-ord* (⊆) *set-option*
by(*auto intro*!: *monotoneI simp add*: *option-ord-Some1-iff*)

lemma *mcont2mcont-set-option* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-set-option*: *mcont* (*flat-lub None*) *option-ord Union* (⊆) *set-option*
by(*rule mcont-finite-chains*)(*simp-all add*: *monotone-set-option ccpo option.partial-function-definitions-axioms*

lemma *eadd-gfp-partial-function-mono* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord* (≥)) (≥) *f*; *monotone* (*fun-ord* (≥)) (≥) *g* ⟧
⟹ *monotone* (*fun-ord* (≥)) (≥) (λ*x*. *f x* + *g x* :: *enat*)
by(*rule mono2mono-gfp-eadd*)

lemma *map-option-mono* [*partial-function-mono*]:
mono-option B ⟹ *mono-option* (λ*f*. *map-option g* (*B f*))
unfolding *map-conv-bind-option* **by**(*rule bind-mono*) *simp-all*

1.13 Folding over finite sets

lemma (**in** *comp-fun-commute*) *fold-invariant-remove* [*consumes 1*, *case-names start step*]:
assumes *fin*: *finite A*
and *start*: *I A s*
and *step*: ⋀*x s A'*. ⟦ *x* ∈ *A'*; *I A' s*; *A'* ⊆ *A* ⟧ ⟹ *I* (*A'* − {*x*}) (*f x s*)
shows *I* {} (*Finite-Set.fold f s A*)
proof −
define *A'* **where** *A'* == *A*
with *fin start* **have** *finite A' A'* ⊆ *A I A' s* **by** *simp-all*
thus *I* {} (*Finite-Set.fold f s A'*)
proof(*induction arbitrary*: *s*)
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A'*)
let *?A'* = *insert x A'*
have *x* ∈ *?A' I ?A' s ?A'* ⊆ *A* **using** *insert* **by** *auto*
hence *I* (*?A'* − {*x*}) (*f x s*) **by**(*rule step*)

with *insert* **have** $A' \subseteq A$ I A' $(f\ x\ s)$ **by** *auto*
hence I {} (*Finite-Set.fold* f $(f\ x\ s)$ A') **by**(*rule insert.IH*)
thus *?case* **using** *insert* **by**(*simp add*: *fold-insert2 del*: *fold-insert*)
qed
qed

lemma (**in** *comp-fun-commute*) *fold-invariant-insert* [*consumes 1*, *case-names start step*]:
assumes *fin*: *finite A*
and *start*: I {} s
and *step*: $\bigwedge x\ s\ A'.$ ⟦ $I\ A'\ s$; $x \notin A'$; $x \in A$; $A' \subseteq A$ ⟧ $\Longrightarrow I$ (*insert* x A') $(f\ x\ s)$
shows I A (*Finite-Set.fold* f s A)
using *fin start*
proof(*rule fold-invariant-remove*[**where** $I{=}\lambda A'.\ I\ (A - A')$ **and** $A{=}A$ **and** $s{=}s$, *simplified*])
fix $x\ s\ A'$
assume $*$: $x \in A'$ $I\ (A - A')\ s$ $A' \subseteq A$
hence $x \notin A - A'$ $x \in A$ $A - A' \subseteq A$ **by** *auto*
with ‹$I\ (A - A')\ s$› **have** I (*insert* x $(A - A')$) $(f\ x\ s)$ **by**(*rule step*)
also have *insert* x $(A - A') = A - (A' - \{x\})$ **using** $*$ **by** *auto*
finally show I ... $(f\ x\ s)$ **.**
qed

lemma (**in** *comp-fun-idem*) *fold-set-union*:
assumes *finite A finite B*
shows *Finite-Set.fold* f z $(A \cup B)$ = *Finite-Set.fold* f (*Finite-Set.fold* f z A) B
using *assms*(*2*,*1*) **by** *induction simp-all*

1.14 Parametrisation of transfer rules

attribute-setup *transfer-parametric* = ‹
```
  Attrib.thm >> (fn parametricity =>
    Thm.rule-attribute [] (fn context => fn transfer-rule =>
      let
        val ctxt = Context.proof-of context;
        val thm' = Lifting-Term.parametrize-transfer-rule ctxt transfer-rule
      in Lifting-Def.generate-parametric-transfer-rule ctxt thm' parametricity
      end
    handle Lifting-Term.MERGE-TRANSFER-REL msg => error (Pretty.string-of
msg)
      ))
```
› *combine transfer rule with parametricity theorem*

1.15 Lists

lemma *nth-eq-tlI*: $xs\ !\ n = z \Longrightarrow (x\ \#\ xs)\ !\ Suc\ n = z$
by *simp*

lemma *list-all2-append'*:

length us = *length vs* ⟹ *list-all2 P* (*xs* @ *us*) (*ys* @ *vs*) ⟷ *list-all2 P xs ys* ∧ *list-all2 P us vs*
by(*auto simp add*: *list-all2-append1 list-all2-append2 dest*: *list-all2-lengthD*)

definition *disjointp* :: (*'a* ⇒ *bool*) *list* ⇒ *bool*
where *disjointp xs* = *disjoint-family-on* (λ*n*. {*x*. (*xs* ! *n*) *x*}) {*0*..<*length xs*}

lemma *disjointpD*:
⟦ *disjointp xs*; (*xs* ! *n*) *x*; (*xs* ! *m*) *x*; *n* < *length xs*; *m* < *length xs* ⟧ ⟹ *n* = *m*
by(*auto 4 3 simp add*: *disjointp-def disjoint-family-on-def*)

lemma *disjointpD'*:
⟦ *disjointp xs*; *P x*; *Q x*; *xs* ! *n* = *P*; *xs* ! *m* = *Q*; *n* < *length xs*; *m* < *length xs* ⟧ ⟹ *n* = *m*
by(*auto 4 3 simp add*: *disjointp-def disjoint-family-on-def*)

lemma *wf-strict-prefix*: *wfP strict-prefix*
proof −
from *wf* **have** *wf* (*inv-image* {(*x*, *y*). *x* < *y*} *length*) **by**(*rule wf-inv-image*)
moreover have {(*x*, *y*). *strict-prefix x y*} ⊆ *inv-image* {(*x*, *y*). *x* < *y*} *length* **by**(*auto intro*: *prefix-length-less*)
ultimately show *?thesis* **unfolding** *wfp-def* **by**(*rule wf-subset*)
qed

lemma *strict-prefix-setD*:
strict-prefix xs ys ⟹ *set xs* ⊆ *set ys*
by(*auto simp add*: *strict-prefix-def prefix-def*)

1.15.1 List of a given length

inductive-set *nlists* :: *'a set* ⇒ *nat* ⇒ *'a list set* **for** *A n*
where *nlists*: ⟦ *set xs* ⊆ *A*; *length xs* = *n* ⟧ ⟹ *xs* ∈ *nlists A n*
hide-fact (**open**) *nlists*

lemma *nlists-alt-def*: *nlists A n* = {*xs*. *set xs* ⊆ *A* ∧ *length xs* = *n*}
by(*auto simp add*: *nlists.simps*)

lemma *nlists-empty*: *nlists* {} *n* = (*if n* = *0 then* {[]} *else* {})
by(*auto simp add*: *nlists-alt-def*)

lemma *nlists-empty-gt0* [*simp*]: *n* > *0* ⟹ *nlists* {} *n* = {}
by(*simp add*: *nlists-empty*)

lemma *nlists-0* [*simp*]: *nlists A 0* = {[]}
by(*auto simp add*: *nlists-alt-def*)

lemma *Cons-in-nlists-Suc* [*simp*]: *x* # *xs* ∈ *nlists A* (*Suc n*) ⟷ *x* ∈ *A* ∧ *xs* ∈ *nlists A n*
by(*simp add*: *nlists-alt-def*)

lemma *Nil-in-nlists* [*simp*]: [] ∈ *nlists A n* ⟷ *n* = *0*
by(*auto simp add*: *nlists-alt-def*)

lemma *Cons-in-nlists-iff*: *x* # *xs* ∈ *nlists A n* ⟷ (∃ *n*′. *n* = *Suc n*′ ∧ *x* ∈ *A* ∧ *xs* ∈ *nlists A n*′)
by(*cases n*) *simp-all*

lemma *in-nlists-Suc-iff*: *xs* ∈ *nlists A* (*Suc n*) ⟷ (∃ *x xs*′. *xs* = *x* # *xs*′ ∧ *x* ∈ *A* ∧ *xs*′ ∈ *nlists A n*)
by(*cases xs*) *simp-all*

lemma *nlists-Suc*: *nlists A* (*Suc n*) = (⋃*x*∈*A*. (#) *x* ‘ *nlists A n*)
by(*auto 4 3 simp add*: *in-nlists-Suc-iff intro*: *rev-image-eqI*)

lemma *replicate-in-nlists* [*simp*, *intro*]: *x* ∈ *A* ⟹ *replicate n x* ∈ *nlists A n*
by(*simp add*: *nlists-alt-def set-replicate-conv-if*)

lemma *nlists-eq-empty-iff* [*simp*]: *nlists A n* = {} ⟷ *n* > *0* ∧ *A* = {}
using *replicate-in-nlists* **by**(*cases n*)(*auto*)

lemma *finite-nlists* [*simp*]: *finite A* ⟹ *finite* (*nlists A n*)
by(*induction n*)(*simp-all add*: *nlists-Suc*)

lemma *finite-nlistsD*:
  assumes *finite* (*nlists A n*)
  shows *finite A* ∨ *n* = *0*
proof(*rule disjCI*)
  assume *n* ≠ *0*
  then obtain *n*′ **where** *n*: *n* = *Suc n*′ **by**(*cases n*)*auto*
  then have *A* = *hd* ‘ *nlists A n* **by**(*auto 4 4 simp add*: *nlists-Suc intro*: *rev-image-eqI rev-bexI*)
  also have *finite* … **using** *assms* **..**
  finally show *finite A* **.**
qed

lemma *finite-nlists-iff*: *finite* (*nlists A n*) ⟷ *finite A* ∨ *n* = *0*
by(*auto dest*: *finite-nlistsD*)

lemma *card-nlists*: *card* (*nlists A n*) = *card A* ^ *n*
proof(*induction n*)
  case (*Suc n*)
  have *card* (⋃*x*∈*A*. (#) *x* ‘ *nlists A n*) = *card A* ∗ *card* (*nlists A n*)
  proof(*cases finite A*)
    case *True*
      then show *?thesis* **by**(*subst card-UN-disjoint*)(*auto simp add*: *card-image inj-on-def*)
  next
    case *False*

hence ¬ *finite* (⋃*x*∈*A*. (#) *x* ‘ *nlists A n*)
unfolding *nlists-Suc*[*symmetric*] **by**(*auto dest*: *finite-nlistsD*)
then show *?thesis* **using** *False* **by** *simp*
qed
then show *?case* **using** *Suc.IH* **by**(*simp add*: *nlists-Suc*)
qed *simp*

lemma *in-nlists-UNIV*: *xs* ∈ *nlists UNIV n* ⟷ *length xs* = *n*
by(*simp add*: *nlists-alt-def*)

1.15.2 The type of lists of a given length

typedef (**overloaded**) (′*a*, ′*b* :: *len0*) *nlist* = *nlists* (*UNIV* :: ′*a set*) (*LENGTH*(′*b*))
proof
show *replicate LENGTH*(′*b*) *undefined* ∈ *?nlist* **by** *simp*
qed

setup-lifting *type-definition-nlist*

1.16 Streams and infinite lists

primrec *sprefix* :: ′*a list* ⇒ ′*a stream* ⇒ *bool* **where**
sprefix-Nil: *sprefix* [] *ys* = *True*
| *sprefix-Cons*: *sprefix* (*x* # *xs*) *ys* ⟷ *x* = *shd ys* ∧ *sprefix xs* (*stl ys*)

lemma *sprefix-append*: *sprefix* (*xs* @ *ys*) *zs* ⟷ *sprefix xs zs* ∧ *sprefix ys* (*sdrop* (*length xs*) *zs*)
by(*induct xs arbitrary*: *zs*) *simp-all*

lemma *sprefix-stake-same* [*simp*]: *sprefix* (*stake n xs*) *xs*
by(*induct n arbitrary*: *xs*) *simp-all*

lemma *sprefix-same-imp-eq*:
assumes *sprefix xs ys sprefix xs*′ *ys*
and *length xs* = *length xs*′
shows *xs* = *xs*′
using *assms*(*3*,*1*,*2*) **by**(*induct arbitrary*: *ys rule*: *list-induct2*) *auto*

lemma *sprefix-shift-same* [*simp*]:
sprefix xs (*xs* @− *ys*)
by(*induct xs*) *simp-all*

lemma *sprefix-shift* [*simp*]:
length xs ≤ *length ys* ⟹ *sprefix xs* (*ys* @− *zs*) ⟷ *prefix xs ys*
by(*induct xs arbitrary*: *ys*)(*simp*, *case-tac ys*, *auto*)

lemma *prefixeq-stake2* [*simp*]: *prefix xs* (*stake n ys*) ⟷ *length xs* ≤ *n* ∧ *sprefix xs ys*
proof(*induct xs arbitrary*: *n ys*)
case (*Cons x xs*)

thus *?case* **by**(*cases ys n rule*: *stream.exhaust*[*case-product nat.exhaust*]) *auto*
qed *simp*

lemma *tlength-eq-infinity-iff*: *tlength xs* = ∞ ⟷ ¬ *tfinite xs*
including *tllist.lifting* **by** *transfer*(*simp add*: *llength-eq-infty-conv-lfinite*)

1.17 Monomorphic monads

context includes *lifting-syntax* **begin**
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path monad*)›

definition *bind-option* :: ′*m fail* ⇒ ′*a option* ⇒ (′*a* ⇒ ′*m*) ⇒ ′*m*
where *bind-option fail x f* = (*case x of None* ⇒ *fail* | *Some x*′ ⇒ *f x*′) **for** *fail*

simps-of-case *bind-option-simps* [*simp*]: *bind-option-def*

lemma *bind-option-parametric* [*transfer-rule*]:
(*M* ===> *rel-option B* ===> (*B* ===> *M*) ===> *M*) *bind-option bind-option*
unfolding *bind-option-def* **by** *transfer-prover*

lemma *bind-option-K*:
⋀*monad*. (*x* = *None* ⟹ *m* = *fail*) ⟹ *bind-option fail x* (λ-. *m*) = *m*
by(*cases x*) *simp-all*

end

lemma *bind-option-option* [*simp*]: *monad.bind-option None* = *Option.bind*
by(*simp add*: *monad.bind-option-def fun-eq-iff split*: *option.split*)

context *monad-fail-hom* **begin**

lemma *hom-bind-option*: *h* (*monad.bind-option fail1 x f*) = *monad.bind-option fail2 x* (*h* ∘ *f*)
by(*cases x*)(*simp-all*)

end

lemma *bind-option-set* [*simp*]: *monad.bind-option fail-set* = (λ*x f*. ⋃ (*f* ‘ *set-option x*))
by(*simp add*: *monad.bind-option-def fun-eq-iff split*: *option.split*)

lemma *run-bind-option-stateT* [*simp*]:
⋀*more*. *run-state* (*monad.bind-option* (*fail-state fail*) *x f*) *s* =
monad.bind-option fail x (λ*y*. *run-state* (*f y*) *s*)
by(*cases x*) *simp-all*

lemma *run-bind-option-envT* [*simp*]:
⋀*more*. *run-env* (*monad.bind-option* (*fail-env fail*) *x f*) *s* =

monad.bind-option fail x (λy. *run-env* (*f y*) *s*)
by(*cases x*) *simp-all*

1.18 Measures

declare *sets-restrict-space-count-space* [*measurable-cong*]

lemma (**in** *sigma-algebra*) *sets-Collect-countable-Ex1*:
($\bigwedge i$ :: $'i$ :: *countable*. $\{x \in \Omega.\ P\ i\ x\} \in M$) $\Longrightarrow$ $\{x \in \Omega.\ \exists!i.\ P\ i\ x\} \in M$
using *sets-Collect-countable-Ex1'*[*of UNIV* :: *'i set*] **by** *simp*

lemma *pred-countable-Ex1* [*measurable*]:
($\bigwedge i$ :: - :: *countable*. *Measurable.pred M* (λx. *P i x*))
$\Longrightarrow$ *Measurable.pred M* (λx. $\exists!i$. *P i x*)
unfolding *pred-def* **by**(*rule sets.sets-Collect-countable-Ex1*)

lemma *measurable-snd-count-space* [*measurable*]:
$A \subseteq B \Longrightarrow$ *snd* $\in$ *measurable* (*M1* $\bigotimes_M$ *count-space A*) (*count-space B*)
by(*auto simp add*: *measurable-def space-pair-measure snd-vimage-eq-Times Times-Int-Times*)

lemma *integrable-scale-measure* [*simp*]:
$[\![$ *integrable M f*; $r < \top$ $]\!] \Longrightarrow$ *integrable* (*scale-measure r M*) *f*
for *f* :: $'a \Rightarrow$ $'b$::{*banach*, *second-countable-topology*}
by(*auto simp add*: *integrable-iff-bounded nn-integral-scale-measure ennreal-mult-less-top*)

lemma *integral-scale-measure*:
assumes *integrable M f r* $< \top$
shows *integral*^{L} (*scale-measure r M*) *f* = *enn2real r* $*$ *integral*^{L} *M f*
using *assms*
apply(*subst* (*1 2*) *real-lebesgue-integral-def*)
apply(*simp-all add*: *nn-integral-scale-measure ennreal-enn2real-if*)
by(*auto simp add*: *ennreal-mult-less-top ennreal-less-top-iff ennreal-mult-eq-top-iff enn2real-mult right-diff-distrib elim*!: *integrableE*)

1.19 Sequence space

lemma (**in** *sequence-space*) *nn-integral-split*:
assumes *f*[*measurable*]: *f* $\in$ *borel-measurable S*
shows ($\int^+\omega$. *f* ω ∂S) = ($\int^+\omega$. ($\int^+\omega'$. *f* (*comb-seq i* ω ω') ∂S) ∂S)
by (*subst PiM-comb-seq*[*symmetric*, **where** *i*=*i*])
(*simp add*: *nn-integral-distr P.nn-integral-fst*[*symmetric*])

lemma (**in** *sequence-space*) *prob-Collect-split*:
assumes *f*[*measurable*]: $\{x\in$*space S*. *P x*$\} \in$ *sets S*
shows $\mathcal{P}$(*x in S*. *P x*) = ($\int^+x$. $\mathcal{P}$(*x'* *in S*. *P* (*comb-seq i x x'*)) ∂S)
proof −
have $\mathcal{P}$(*x in S*. *P x*) = ($\int^+x$. ($\int^+x'$. *indicator* $\{x\in$*space S*. *P x*$\}$ (*comb-seq i x x'*) ∂S) ∂S)
using *nn-integral-split*[*of indicator* $\{x\in$*space S*. *P x*$\}$] **by** (*auto simp*: *emeasure-eq-measure*)

also have ... $= (\int^{+}x.\ \mathcal{P}(x'\ in\ S.\ P\ (comb\text{-}seq\ i\ x\ x'))\ \partial S)$
 by (*intro nn-integral-cong*) (*auto simp*: *emeasure-eq-measure nn-integral-indicator-map*)
finally show *?thesis* **.**
qed

1.20 Probability mass functions

lemma *measure-map-pmf-conv-distr*:
measure-pmf (*map-pmf f p*) = *distr* (*measure-pmf p*) (*count-space UNIV*) *f*
by(*fact map-pmf-rep-eq*)

abbreviation *coin-pmf* :: *bool pmf* **where** *coin-pmf* ≡ *pmf-of-set UNIV*

The rule *rel-pmf-bindI* is not complete as a program logic.

notepad begin
define *x* **where** *x* = *pmf-of-set* {*True*, *False*}
define *y* **where** *y* = *pmf-of-set* {*True*, *False*}
define *f* **where** *f x* = *pmf-of-set* {*True*, *False*} **for** *x* :: *bool*
define *g* :: *bool* ⇒ *bool pmf* **where** *g* = *return-pmf*
define *P* :: *bool* ⇒ *bool* ⇒ *bool* **where** *P* = (=)
have *rel-pmf P* (*bind-pmf x f*) (*bind-pmf y g*)
by(*simp add*: *P-def f-def*[*abs-def*] *g-def y-def bind-return-pmf'* *pmf.rel-eq*)
have ¬ *R x y* **if** ⋀*x y*. *R x y* ⟹ *rel-pmf P* (*f x*) (*g y*) **for** *R x y*
— Only the empty relation satisfies *rel-pmf-bindI*'s second premise.
proof
assume *R x y*
hence *rel-pmf P* (*f x*) (*g y*) **by**(*rule that*)
thus *False* **by**(*auto simp add*: *P-def f-def g-def rel-pmf-return-pmf2*)
qed
define *R* **where** *R x y* = *False* **for** *x y* :: *bool*
have ¬ *rel-pmf R x y* **by**(*simp add*: *R-def*[*abs-def*])
end

lemma *pred-rel-pmf*:
⟦ *pred-pmf P p*; *rel-pmf R p q* ⟧ ⟹ *pred-pmf* (*Imagep R P*) *q*
unfolding *pred-pmf-def*
apply(*rule ballI*)
apply(*unfold rel-pmf.simps*)
apply(*erule exE conjE*)+
apply *hypsubst*
apply(*unfold pmf.set-map*)
apply(*erule imageE*, *hypsubst*)
apply(*drule bspec*)
apply(*erule rev-image-eqI*)
apply(*rule refl*)
apply(*erule Imagep.intros*)
apply(*erule allE*)+
apply(*erule mp*)
apply(*unfold prod.collapse*)

apply *assumption*
done

lemma *pmf-rel-mono′*: ⟦ *rel-pmf P x y*; $P \leq Q$ ⟧ $\Longrightarrow$ *rel-pmf Q x y*
by(*drule pmf.rel-mono*) (*auto*)

lemma *rel-pmf-eqI* [*simp*]: *rel-pmf* (=) *x x*
by(*simp add*: *pmf.rel-eq*)

lemma *rel-pmf-bind-reflI*:
  ($\bigwedge$*x*. $x \in$ *set-pmf p* $\Longrightarrow$ *rel-pmf R* (*f x*) (*g x*))
  $\Longrightarrow$ *rel-pmf R* (*bind-pmf p f*) (*bind-pmf p g*)
by(*rule rel-pmf-bindI*[**where** *R*=λ*x y*. $x = y \wedge x \in$ *set-pmf p*])(*auto intro*: *rel-pmf-reflI*)

lemma *pmf-pred-mono-strong*:
  ⟦ *pred-pmf P p*; $\bigwedge$*a*. ⟦ $a \in$ *set-pmf p*; *P a* ⟧ $\Longrightarrow$ *P′ a* ⟧ $\Longrightarrow$ *pred-pmf P′ p*
by(*simp add*: *pred-pmf-def*)

lemma *rel-pmf-restrict-relpI* [*intro?*]:
  ⟦ *rel-pmf R x y*; *pred-pmf P x*; *pred-pmf Q y* ⟧ $\Longrightarrow$ *rel-pmf* ($R \upharpoonright P \otimes Q$) *x y*
by(*erule pmf.rel-mono-strong*)(*simp add*: *pred-pmf-def*)

lemma *rel-pmf-restrict-relpE* [*elim?*]:
  assumes *rel-pmf* ($R \upharpoonright P \otimes Q$) *x y*
  obtains *rel-pmf R x y pred-pmf P x pred-pmf Q y*
proof
  show *rel-pmf R x y* **using** *assms* **by**(*auto elim*!: *pmf.rel-mono-strong*)
  have *pred-pmf* (*Domainp* ($R \upharpoonright P \otimes Q$)) *x* **using** *assms* **by**(*fold pmf.Domainp-rel*)
blast
  then show *pred-pmf P x* **by**(*rule pmf-pred-mono-strong*)(*blast dest*!: *restrict-relp-DomainpD*)
  have *pred-pmf* (*Domainp* $(R \upharpoonright P \otimes Q)^{-1-1}$) *y* **using** *assms*
   by(*fold pmf.Domainp-rel*)(*auto simp only*: *pmf.rel-conversep Domainp-conversep*)
  then show *pred-pmf Q y* **by**(*rule pmf-pred-mono-strong*)(*auto dest*!: *restrict-relp-DomainpD*)
qed

lemma *rel-pmf-restrict-relp-iff*:
  rel-pmf ($R \upharpoonright P \otimes Q$) *x y* $\longleftrightarrow$ *rel-pmf R x y* $\wedge$ *pred-pmf P x* $\wedge$ *pred-pmf Q y*
by(*blast intro*: *rel-pmf-restrict-relpI elim*: *rel-pmf-restrict-relpE*)

lemma *rel-pmf-OO-trans* [*trans*]:
  ⟦ *rel-pmf R p q*; *rel-pmf S q r* ⟧ $\Longrightarrow$ *rel-pmf* (*R OO S*) *p r*
unfolding *pmf.rel-compp* **by** *blast*

lemma *pmf-pred-map* [*simp*]: *pred-pmf P* (*map-pmf f p*) = *pred-pmf* ($P \circ f$) *p*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-bind* [*simp*]: *pred-pmf P* (*bind-pmf p f*) = *pred-pmf* (*pred-pmf P*
$\circ$ *f*) *p*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-return* [*simp*]: *pred-pmf P* (*return-pmf x*) = *P x*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-of-set* [*simp*]: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *pred-pmf P* (*pmf-of-set A*) = *Ball A P*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-of-multiset* [*simp*]: *M* ≠ {#} ⟹ *pred-pmf P* (*pmf-of-multiset M*) = *Ball* (*set-mset M*) *P*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-cond* [*simp*]:
set-pmf p ∩ *A* ≠ {} ⟹ *pred-pmf P* (*cond-pmf p A*) = *pred-pmf* (λ*x*. *x* ∈ *A* ⟶ *P x*) *p*
by(*auto simp add*: *pred-pmf-def*)

lemma *pred-pmf-pair* [*simp*]:
pred-pmf P (*pair-pmf p q*) = *pred-pmf* (λ*x*. *pred-pmf* (*P* ∘ *Pair x*) *q*) *p*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-join* [*simp*]: *pred-pmf P* (*join-pmf p*) = *pred-pmf* (*pred-pmf P*) *p*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-bernoulli* [*simp*]: ⟦ *0* < *p*; *p* < *1* ⟧ ⟹ *pred-pmf P* (*bernoulli-pmf p*) = *All P*
by(*simp add*: *pred-pmf-def*)

lemma *pred-pmf-geometric* [*simp*]: ⟦ *0* < *p*; *p* < *1* ⟧ ⟹ *pred-pmf P* (*geometric-pmf p*) = *All P*
by(*simp add*: *pred-pmf-def set-pmf-geometric*)

lemma *pred-pmf-poisson* [*simp*]: *0* < *rate* ⟹ *pred-pmf P* (*poisson-pmf rate*) = *All P*
by(*simp add*: *pred-pmf-def*)

lemma *pmf-rel-map-restrict-relp*:
shows *pmf-rel-map-restrict-relp1*: *rel-pmf* (*R* ↿ *P* ⊗ *Q*) (*map-pmf f p*) = *rel-pmf* (*R* ∘ *f* ↿ *P* ∘ *f* ⊗ *Q*) *p*
and *pmf-rel-map-restrict-relp2*: *rel-pmf* (*R* ↿ *P* ⊗ *Q*) *p* (*map-pmf g q*) = *rel-pmf* ((λ*x*. *R x* ∘ *g*) ↿ *P* ⊗ *Q* ∘ *g*) *p q*
by(*simp-all add*: *pmf.rel-map restrict-relp-def fun-eq-iff*)

lemma *pred-pmf-conj* [*simp*]: *pred-pmf* (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *pred-pmf P x* ∧ *pred-pmf Q x*)
by(*auto simp add*: *pred-pmf-def*)

lemma *pred-pmf-top* [*simp*]:
pred-pmf (λ-. *True*) = (λ-. *True*)

by(*simp add*: *pred-pmf-def*)

lemma *rel-pmf-of-setI*:
  assumes *A*: $A \neq \{\}$ *finite A*
  and *B*: $B \neq \{\}$ *finite B*
  and *card*: $\bigwedge X.\ X \subseteq A \Longrightarrow card\ B * card\ X \leq card\ A * card\ \{y\in B.\ \exists x\in X.\ R\ x\ y\}$
  shows *rel-pmf R* (*pmf-of-set A*) (*pmf-of-set B*)
apply(*rule rel-pmf-measureI*)
using *assms*
apply(*clarsimp simp add*: *measure-pmf-of-set card-gt-0-iff field-simps of-nat-mult*[*symmetric*] *simp del*: *of-nat-mult*)
apply(*subst mult.commute*)
apply(*erule meta-allE*)
apply(*erule meta-impE*)
 prefer *2*
 apply(*erule order-trans*)
apply(*auto simp add*: *card-gt-0-iff intro*: *card-mono*)
done

consts *rel-witness-pmf* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ pmf \times 'b\ pmf \Rightarrow ('a \times 'b)\ pmf$
specification (*rel-witness-pmf*)
  set-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) $\Longrightarrow$ *set-pmf* (*rel-witness-pmf A xy*) $\subseteq \{(a, b).\ A\ a\ b\}$
  map1-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) $\Longrightarrow$ *map-pmf fst* (*rel-witness-pmf A xy*) = *fst xy*
  map2-rel-witness-pmf': *rel-pmf A* (*fst xy*) (*snd xy*) $\Longrightarrow$ *map-pmf snd* (*rel-witness-pmf A xy*) = *snd xy*
  apply(*fold all-conj-distrib imp-conjR*)
  apply(*rule choice allI*)+
  apply(*unfold pmf.in-rel*)
  by *blast*

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

lemma *rel-witness-pmf1*:
  assumes *rel-pmf A p q*
  shows *rel-pmf* ($\lambda a\ (a', b).\ a = a' \wedge A\ a'\ b$) *p* (*rel-witness-pmf A* (*p*, *q*))
  using *map1-rel-witness-pmf*[*OF assms*, *symmetric*]
  unfolding *pmf.rel-eq*[*symmetric*] *pmf.rel-map*
  by(*rule pmf.rel-mono-strong*)(*auto dest*: *set-rel-witness-pmf*[*OF assms*, *THEN subsetD*])

lemma *rel-witness-pmf2*:

assumes *rel-pmf A p q*
shows *rel-pmf* $(\lambda(a, b')\ b.\ b = b' \wedge A\ a\ b')$ *(rel-witness-pmf A (p, q)) q*
using *map2-rel-witness-pmf[OF assms]*
unfolding *pmf.rel-eq[symmetric] pmf.rel-map*
by(*rule pmf.rel-mono-strong*)(*auto dest*: *set-rel-witness-pmf[OF assms, THEN subsetD]*)

lemma *cond-pmf-of-set*:
assumes *fin*: *finite A* **and** *nonempty*: $A \cap B \neq \{\}$
shows *cond-pmf (pmf-of-set A) B = pmf-of-set* $(A \cap B)$ (**is** *?lhs = ?rhs*)
proof(*rule pmf-eqI*)
from *nonempty* **have** *A*: $A \neq \{\}$ **by** *auto*
show *pmf ?lhs x = pmf ?rhs x* **for** *x*
by(*subst pmf-cond*; *clarsimp simp add*: *fin A nonempty measure-pmf-of-set split*: *split-indicator*)
qed

lemma *pair-pmf-of-set*:
assumes *A*: *finite A* $A \neq \{\}$
and *B*: *finite B* $B \neq \{\}$
shows *pair-pmf (pmf-of-set A) (pmf-of-set B) = pmf-of-set* $(A \times B)$
by(*rule pmf-eqI*)(*clarsimp simp add*: *pmf-pair assms split*: *split-indicator*)

lemma *emeasure-cond-pmf*:
fixes *p A*
defines $q \equiv$ *cond-pmf p A*
assumes *set-pmf p* $\cap A \neq \{\}$
shows *emeasure (measure-pmf q) B = emeasure (measure-pmf p)* $(A \cap B)$ / *emeasure (measure-pmf p) A*
proof −
note [*transfer-rule*] = *cond-pmf.transfer[OF assms(2), folded q-def]*
interpret *pmf-as-measure* **.**
show *?thesis* **by** *transfer simp*
qed

lemma *measure-cond-pmf*:
measure (measure-pmf (cond-pmf p A)) B = measure (measure-pmf p) $(A \cap B)$ / *measure (measure-pmf p) A*
if *set-pmf p* $\cap A \neq \{\}$
using *emeasure-cond-pmf[OF that, of B] that*
by(*auto simp add*: *measure-pmf.emeasure-eq-measure measure-pmf-posI divide-ennreal*)

lemma *emeasure-measure-pmf-zero-iff*: *emeasure (measure-pmf p) s = 0* $\longleftrightarrow$ *set-pmf p* $\cap$ *s* = {} (**is** *?lhs = ?rhs*)
proof −
have *?lhs* $\longleftrightarrow$ (*AE x in measure-pmf p.* $x \notin s$)
by(*subst AE-iff-measurable*)(*auto*)
also have ... = *?rhs* **by**(*auto simp add*: *AE-measure-pmf-iff*)
finally show *?thesis* **.**

qed

1.21 Subprobability mass functions

lemma *ord-spmf-return-spmf1*: *ord-spmf R* (*return-spmf x*) *p* ⟷ *lossless-spmf p*
∧ (∀ *y*∈*set-spmf p*. *R x y*)
by(*auto simp add*: *rel-pmf-return-pmf1 ord-option.simps in-set-spmf lossless-iff-set-pmf-None Ball-def*) (*metis option.exhaust*)

lemma *ord-spmf-conv*:
  ord-spmf R = *rel-spmf R OO ord-spmf* (=)
apply(*subst pmf.rel-compp*[*symmetric*])
apply(*rule arg-cong*[**where** *f*=*rel-pmf*])
apply(*rule ext*)+
apply(*auto elim*!: *ord-option.cases option.rel-cases intro*: *option.rel-intros*)
done

lemma *ord-spmf-expand*:
  NO-MATCH (=) *R* ⟹ *ord-spmf R* = *rel-spmf R OO ord-spmf* (=)
by(*rule ord-spmf-conv*)

lemma *ord-spmf-eqD-measure*: *ord-spmf* (=) *p q* ⟹ *measure* (*measure-spmf p*)
A ≤ *measure* (*measure-spmf q*) *A*
by(*drule ord-spmf-eqD-measure-spmf*)(*simp add*: *le-measure measure-spmf.emeasure-eq-measure*)

lemma *ord-spmf-measureD*:
  assumes *ord-spmf R p q*
  shows *measure* (*measure-spmf p*) *A* ≤ *measure* (*measure-spmf q*) {*y*. ∃ *x*∈*A*. *R*
x y}
    (**is** *?lhs* ≤ *?rhs*)
proof −
  from *assms* **obtain** *p′* **where** ∗: *rel-spmf R p p′* **and** ∗∗: *ord-spmf* (=) *p′ q*
    by(*auto simp add*: *ord-spmf-expand*)
  have *?lhs* ≤ *measure* (*measure-spmf p′*) {*y*. ∃ *x*∈*A*. *R x y*} **using** ∗ **by**(*rule*
rel-spmf-measureD)
  also have . . . ≤ *?rhs* **using** ∗∗ **by**(*rule ord-spmf-eqD-measure*)
  finally show *?thesis* **.**
qed

lemma *ord-spmf-bind-pmfI1*:
  (⋀*x*. *x* ∈ *set-pmf p* ⟹ *ord-spmf R* (*f x*) *q*) ⟹ *ord-spmf R* (*bind-pmf p f*) *q*
  apply(*rewrite at ord-spmf - - ⌑ bind-return-pmf*[*symmetric*, **where** *f*=λ- :: *unit*.
q])
  apply(*rule rel-pmf-bindI*[**where** *R*=λ*x y*. *x* ∈ *set-pmf p*])
  apply(*simp-all add*: *rel-pmf-return-pmf2*)
  done

lemma *ord-spmf-bind-spmfI1*:
  (⋀*x*. *x* ∈ *set-spmf p* ⟹ *ord-spmf R* (*f x*) *q*) ⟹ *ord-spmf R* (*bind-spmf p f*) *q*

unfolding *bind-spmf-def* **by**(*rule ord-spmf-bind-pmfI1*)(*auto split*: *option.split simp add*: *in-set-spmf*)

lemma *spmf-of-set-empty*: *spmf-of-set* {} = *return-pmf None*
by(*simp add*: *spmf-of-set-def*)

lemma *rel-spmf-of-setI*:
assumes *card*: $\bigwedge X.\ X \subseteq A \Longrightarrow card\ B * card\ X \leq card\ A * card\ \{y{\in}B.\ \exists x{\in}X.\ R\ x\ y\}$
and *eq*: $(finite\ A \wedge A \neq \{\}) \longleftrightarrow (finite\ B \wedge B \neq \{\})$
shows *rel-spmf R* (*spmf-of-set A*) (*spmf-of-set B*)
using *eq* **by**(*clarsimp simp add*: *spmf-of-set-def card rel-pmf-of-setI simp del*: *spmf-of-pmf-pmf-of-set cong*: *conj-cong*)

lemmas *map-bind-spmf* = *map-spmf-bind-spmf*

lemma *nn-integral-measure-spmf-conv-measure-pmf*:
assumes [*measurable*]: $f \in$ *borel-measurable* (*count-space UNIV*)
shows *nn-integral* (*measure-spmf p*) *f* = *nn-integral* (*restrict-space* (*measure-pmf p*) (*range Some*)) ($f \circ the$)
by(*simp add*: *measure-spmf-def nn-integral-distr o-def*)

lemma *nn-integral-spmf-neq-infinity*: $(\int^+ x.\ spmf\ p\ x\ \partial count\text{-}space\ UNIV) \neq \infty$
using *nn-integral-measure-spmf*[**where** *f*=λ-. *1*, *of p*, *symmetric*] **by** *simp*

lemma *return-pmf-bind-option*:
return-pmf (*Option.bind x f*) = *bind-spmf* (*return-pmf x*) (*return-pmf* ∘ *f*)
by(*cases x*) *simp-all*

lemma *rel-spmf-pos-distr*: *rel-spmf A OO rel-spmf B* ≤ *rel-spmf* (*A OO B*)
unfolding *option.rel-compp pmf.rel-compp* **..**

lemma *rel-spmf-OO-trans* [*trans*]:
⟦ *rel-spmf R p q*; *rel-spmf S q r* ⟧ ⟹ *rel-spmf* (*R OO S*) *p r*
by(*rule rel-spmf-pos-distr*[*THEN predicate2D*]) *auto*

lemma *map-spmf-eq-map-spmf-iff*: *map-spmf f p* = *map-spmf g q* ⟷ *rel-spmf* (λ*x y*. *f x* = *g y*) *p q*
by(*simp add*: *spmf-rel-eq*[*symmetric*] *spmf-rel-map*)

lemma *map-spmf-eq-map-spmfI*: *rel-spmf* (λ*x y*. *f x* = *g y*) *p q* ⟹ *map-spmf f p* = *map-spmf g q*
by(*simp add*: *map-spmf-eq-map-spmf-iff*)

lemma *spmf-rel-mono-strong*:
⟦*rel-spmf A f g*; ⋀*x y*. ⟦ *x* ∈ *set-spmf f*; *y* ∈ *set-spmf g*; *A x y* ⟧ ⟹ *B x y* ⟧ ⟹ *rel-spmf B f g*
apply(*erule pmf.rel-mono-strong*)
apply(*erule option.rel-mono-strong*)

by(*clarsimp simp add*: *in-set-spmf*)

lemma *set-spmf-eq-empty*: *set-spmf p* = {} ⟷ *p* = *return-pmf None*
by *auto* (*metis restrict-spmf-empty restrict-spmf-trivial*)

lemma *measure-pair-spmf-times*:
measure (*measure-spmf* (*pair-spmf p q*)) (*A* × *B*) = *measure* (*measure-spmf p*) *A* ∗ *measure* (*measure-spmf q*) *B*
proof −
have *emeasure* (*measure-spmf* (*pair-spmf p q*)) (*A* × *B*) = (∫⁺ *x*. *ennreal* (*spmf* (*pair-spmf p q*) *x*) ∗ *indicator* (*A* × *B*) *x* ∂*count-space UNIV*)
by(*simp add*: *nn-integral-spmf*[*symmetric*] *nn-integral-count-space-indicator*)
also have ... = (∫⁺ *x*. (∫⁺ *y*. (*ennreal* (*spmf p x*) ∗ *indicator A x*) ∗ (*ennreal* (*spmf q y*) ∗ *indicator B y*) ∂*count-space UNIV*) ∂*count-space UNIV*)
by(*subst nn-integral-fst-count-space*[*symmetric*])(*auto intro*!: *nn-integral-cong split*: *split-indicator simp add*: *ennreal-mult*)
also have ... = (∫⁺ *x*. *ennreal* (*spmf p x*) ∗ *indicator A x* ∗ *emeasure* (*measure-spmf q*) *B* ∂*count-space UNIV*)
by(*simp add*: *nn-integral-cmult nn-integral-spmf*[*symmetric*] *nn-integral-count-space-indicator*)
also have ... = *emeasure* (*measure-spmf p*) *A* ∗ *emeasure* (*measure-spmf q*) *B*
by(*simp add*: *nn-integral-multc*)(*simp add*: *nn-integral-spmf*[*symmetric*] *nn-integral-count-space-indicator*)
finally show *?thesis* **by**(*simp add*: *measure-spmf.emeasure-eq-measure ennreal-mult*[*symmetric*])
qed

lemma *lossless-spmfD-set-spmf-nonempty*: *lossless-spmf p* ⟹ *set-spmf p* ≠ {}
using *set-pmf-not-empty*[*of p*] **by**(*auto simp add*: *set-spmf-def bind-UNION lossless-iff-set-pmf-None*)

lemma *set-spmf-return-pmf*: *set-spmf* (*return-pmf x*) = *set-option x*
by(*cases x*) *simp-all*

lemma *bind-spmf-pmf-assoc*: *bind-spmf* (*bind-pmf p f*) *g* = *bind-pmf p* (λ*x*. *bind-spmf* (*f x*) *g*)
by(*simp add*: *bind-spmf-def bind-assoc-pmf*)

lemma *bind-spmf-of-set*: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *bind-spmf* (*spmf-of-set A*) *f* = *bind-pmf* (*pmf-of-set A*) *f*
by(*simp add*: *spmf-of-set-def del*: *spmf-of-pmf-pmf-of-set*)

lemma *bind-spmf-map-pmf*:
bind-spmf (*map-pmf f p*) *g* = *bind-pmf p* (λ*x*. *bind-spmf* (*return-pmf* (*f x*)) *g*)
by(*simp add*: *map-pmf-def bind-spmf-def bind-assoc-pmf*)

lemma *rel-spmf-eqI* [*simp*]: *rel-spmf* (=) *x x*
by(*simp add*: *option.rel-eq*)

lemma *set-spmf-map-pmf*: *set-spmf* (*map-pmf f p*) = (⋃*x*∈*set-pmf p*. *set-option* (*f x*))

by(*simp add*: *set-spmf-def bind-UNION*)

lemma *ord-spmf-return-spmf* [*simp*]: *ord-spmf* (=) (*return-spmf x*) *p* $\longleftrightarrow$ *p* = *return-spmf x*
proof −
  have *p* = *return-spmf x* $\Longrightarrow$ *ord-spmf* (=) (*return-spmf x*) *p* **by** *simp*
  thus *?thesis*
   by (*metis* (*no-types*) *ord-option-eq-simps*(*2*) *rel-pmf-return-pmf1 rel-pmf-return-pmf2 spmf.leq-antisym*)
qed

declare
  set-bind-spmf [*simp*]
  set-spmf-return-pmf [*simp*]

lemma *bind-spmf-pmf-commute*:
  bind-spmf p (λ*x*. *bind-pmf q* (*f x*)) = *bind-pmf q* (λ*y*. *bind-spmf p* (λ*x*. *f x y*))
unfolding *bind-spmf-def*
by(*subst bind-commute-pmf*)(*auto intro*: *bind-pmf-cong*[*OF refl*] *split*: *option.split*)

lemma *return-pmf-map-option-conv-bind*:
  return-pmf (*map-option f x*) = *bind-spmf* (*return-pmf x*) (*return-spmf* $\circ$ *f*)
by(*cases x*) *simp-all*

lemma *lossless-return-pmf-iff* [*simp*]: *lossless-spmf* (*return-pmf x*) $\longleftrightarrow$ *x* $\neq$ *None*
by(*cases x*) *simp-all*

lemma *lossless-map-pmf*: *lossless-spmf* (*map-pmf f p*) $\longleftrightarrow$ ($\forall$ *x* $\in$ *set-pmf p*. *f x* $\neq$ *None*)
using *image-iff* **by**(*fastforce simp add*: *lossless-iff-set-pmf-None*)

lemma *bind-pmf-spmf-assoc*:
  g None = *return-pmf None*
  $\Longrightarrow$ *bind-pmf* (*bind-spmf p f*) *g* = *bind-spmf p* (λ*x*. *bind-pmf* (*f x*) *g*)
by(*auto simp add*: *bind-spmf-def bind-assoc-pmf bind-return-pmf fun-eq-iff intro*!: *arg-cong2*[**where** *f*=*bind-pmf*] *split*: *option.split*)

abbreviation *pred-spmf* :: (*'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a spmf* $\Rightarrow$ *bool*
where *pred-spmf P* $\equiv$ *pred-pmf* (*pred-option P*)

lemma *pred-spmf-def*: *pred-spmf P p* $\longleftrightarrow$ ($\forall$ *x*$\in$*set-spmf p*. *P x*)
by(*auto simp add*: *pred-pmf-def pred-option-def set-spmf-def*)

lemma *spmf-pred-mono-strong*:
  $\llbracket$ *pred-spmf P p*; $\bigwedge$*a*. $\llbracket$ *a* $\in$ *set-spmf p*; *P a* $\rrbracket$ $\Longrightarrow$ *P' a* $\rrbracket$ $\Longrightarrow$ *pred-spmf P' p*
by(*simp add*: *pred-spmf-def*)

lemma *spmf-Domainp-rel*: *Domainp* (*rel-spmf R*) = *pred-spmf* (*Domainp R*)
by(*simp add*: *pmf.Domainp-rel option.Domainp-rel*)

lemma *rel-spmf-restrict-relpI* [*intro?*]:
⟦ *rel-spmf R p q*; *pred-spmf P p*; *pred-spmf Q q* ⟧ $\Longrightarrow$ *rel-spmf* ($R \upharpoonright P \otimes Q$) *p q*
by(*erule spmf-rel-mono-strong*)(*simp add*: *pred-spmf-def*)

lemma *rel-spmf-restrict-relpE* [*elim?*]:
assumes *rel-spmf* ($R \upharpoonright P \otimes Q$) *x y*
obtains *rel-spmf R x y pred-spmf P x pred-spmf Q y*
proof
show *rel-spmf R x y* **using** *assms* **by**(*auto elim*!: *spmf-rel-mono-strong*)
have *pred-spmf* (*Domainp* ($R \upharpoonright P \otimes Q$)) *x* **using** *assms* **by**(*fold spmf-Domainp-rel*)
blast
then show *pred-spmf P x* **by**(*rule spmf-pred-mono-strong*)(*blast dest*!: *restrict-relp-DomainpD*)
have *pred-spmf* (*Domainp* $(R \upharpoonright P \otimes Q)^{-1-1}$) *y* **using** *assms*
by(*fold spmf-Domainp-rel*)(*auto simp only*: *spmf-rel-conversep Domainp-conversep*)
then show *pred-spmf Q y* **by**(*rule spmf-pred-mono-strong*)(*auto dest*!: *restrict-relp-DomainpD*)
qed

lemma *rel-spmf-restrict-relp-iff*:
rel-spmf ($R \upharpoonright P \otimes Q$) *x y* $\longleftrightarrow$ *rel-spmf R x y* $\wedge$ *pred-spmf P x* $\wedge$ *pred-spmf Q y*
by(*blast intro*: *rel-spmf-restrict-relpI elim*: *rel-spmf-restrict-relpE*)

lemma *spmf-pred-map*: *pred-spmf P* (*map-spmf f p*) = *pred-spmf* ($P \circ f$) *p*
by(*simp*)

lemma *pred-spmf-bind* [*simp*]: *pred-spmf P* (*bind-spmf p f*) = *pred-spmf* (*pred-spmf*
$P \circ f$) *p*
by(*simp add*: *pred-spmf-def bind-UNION*)

lemma *pred-spmf-return*: *pred-spmf P* (*return-spmf x*) = *P x*
by *simp*

lemma *pred-spmf-return-pmf-None*: *pred-spmf P* (*return-pmf None*)
by *simp*

lemma *pred-spmf-spmf-of-pmf* [*simp*]: *pred-spmf P* (*spmf-of-pmf p*) = *pred-pmf P*
p
unfolding *pred-spmf-def* **by**(*simp add*: *pred-pmf-def*)

lemma *pred-spmf-of-set* [*simp*]: *pred-spmf P* (*spmf-of-set A*) = (*finite A* $\longrightarrow$ *Ball*
A P)
by(*auto simp add*: *pred-spmf-def set-spmf-of-set*)

lemma *pred-spmf-assert-spmf* [*simp*]: *pred-spmf P* (*assert-spmf b*) = (*b* $\longrightarrow$ *P* ())
by(*cases b*) *simp-all*

lemma *pred-spmf-pair* [*simp*]:
pred-spmf P (*pair-spmf p q*) = *pred-spmf* ($\lambda x.$ *pred-spmf* ($P \circ$ *Pair x*) *q*) *p*
by(*simp add*: *pred-spmf-def*)

lemma *set-spmf-try* [*simp*]:
set-spmf (*try-spmf p q*) = *set-spmf p* ∪ (*if lossless-spmf p then* {} *else set-spmf q*)
by(*auto simp add*: *try-spmf-def set-spmf-bind-pmf in-set-spmf lossless-iff-set-pmf-None split*: *option.splits*)(*metis option.collapse*)

lemma *try-spmf-bind-out1*:
(⋀*x*. *lossless-spmf* (*f x*)) ⟹ *bind-spmf* (*TRY p ELSE q*) *f* = *TRY* (*bind-spmf p f*) *ELSE* (*bind-spmf q f*)
apply(*clarsimp simp add*: *bind-spmf-def try-spmf-def bind-assoc-pmf bind-return-pmf intro*!: *bind-pmf-cong*[*OF refl*] *split*: *option.split*)
apply(*rewrite in* ⌑ = - *bind-return-pmf*′[*symmetric*])
apply(*rule bind-pmf-cong*[*OF refl*])
apply(*clarsimp split*: *option.split simp add*: *lossless-iff-set-pmf-None*)
done

lemma *pred-spmf-try* [*simp*]:
pred-spmf P (*try-spmf p q*) = (*pred-spmf P p* ∧ (¬ *lossless-spmf p* ⟶ *pred-spmf P q*))
by(*auto simp add*: *pred-spmf-def*)

lemma *pred-spmf-cond* [*simp*]:
pred-spmf P (*cond-spmf p A*) = *pred-spmf* (λ*x*. *x* ∈ *A* ⟶ *P x*) *p*
by(*auto simp add*: *pred-spmf-def*)

lemma *spmf-rel-map-restrict-relp*:
shows *spmf-rel-map-restrict-relp1*: *rel-spmf* (*R* ↾ *P* ⊗ *Q*) (*map-spmf f p*) = *rel-spmf* (*R* ∘ *f* ↾ *P* ∘ *f* ⊗ *Q*) *p*
and *spmf-rel-map-restrict-relp2*: *rel-spmf* (*R* ↾ *P* ⊗ *Q*) *p* (*map-spmf g q*) = *rel-spmf* ((λ*x*. *R x* ∘ *g*) ↾ *P* ⊗ *Q* ∘ *g*) *p q*
by(*simp-all add*: *spmf-rel-map restrict-relp-def*)

lemma *pred-spmf-conj*: *pred-spmf* (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *pred-spmf P x* ∧ *pred-spmf Q x*)
by *simp*

lemma *spmf-of-pmf-parametric* [*transfer-rule*]:
includes *lifting-syntax* **shows**
(*rel-pmf A* ===> *rel-spmf A*) *spmf-of-pmf spmf-of-pmf*
unfolding *spmf-of-pmf-def*[*abs-def*] **by** *transfer-prover*

lemma *mono2mono-return-pmf*[*THEN spmf.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-return-pmf*: *monotone option-ord* (*ord-spmf* (=)) *return-pmf*
by(*rule monotoneI*)(*auto simp add*: *flat-ord-def*)

lemma *mcont2mcont-return-pmf*[*THEN spmf.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-return-pmf*: *mcont* (*flat-lub None*) *option-ord lub-spmf* (*ord-spmf* (=)) *return-pmf*

by(*rule mcont-finite-chains*[*OF - - flat-interpretation*[*THEN ccpo*] *ccpo-spmf*]) *simp-all*

lemma *pred-spmf-top*:
pred-spmf (λ-. *True*) = (λ-. *True*)
by(*simp*)

lemma *rel-spmf-restrict-relpI'* [*intro?*]:
⟦ *rel-spmf* (λ*x y*. *P x* ⟶ *Q y* ⟶ *R x y*) *p q*; *pred-spmf P p*; *pred-spmf Q q* ⟧
⟹ *rel-spmf* (*R* ↿ *P* ⊗ *Q*) *p q*
by(*erule spmf-rel-mono-strong*)(*simp add*: *pred-spmf-def*)

lemma *set-spmf-map-pmf-MATCH* [*simp*]:
assumes *NO-MATCH* (*map-option g*) *f*
shows *set-spmf* (*map-pmf f p*) = (⋃*x*∈*set-pmf p*. *set-option* (*f x*))
by(*rule set-spmf-map-pmf*)

lemma *rel-spmf-bindI'*:
⟦ *rel-spmf A p q*; ⋀*x y*. ⟦ *A x y*; *x* ∈ *set-spmf p*; *y* ∈ *set-spmf q* ⟧ ⟹ *rel-spmf B* (*f x*) (*g y*) ⟧
⟹ *rel-spmf B* (*p* ≫= *f*) (*q* ≫= *g*)
apply(*rule rel-spmf-bindI*[**where** *R*=λ*x y*. *A x y* ∧ *x* ∈ *set-spmf p* ∧ *y* ∈ *set-spmf q*])
apply(*erule spmf-rel-mono-strong*; *simp*)
apply *simp*
done

definition *rel-witness-spmf* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a spmf* × '*b spmf* ⇒ ('*a* × '*b*) *spmf* **where**
rel-witness-spmf A = *map-pmf rel-witness-option* ∘ *rel-witness-pmf* (*rel-option A*)

lemma assumes *rel-spmf A p q*
shows *rel-witness-spmf1*: *rel-spmf* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) *p* (*rel-witness-spmf A* (*p*, *q*))
and *rel-witness-spmf2*: *rel-spmf* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (*rel-witness-spmf A* (*p*, *q*)) *q*
by(*auto simp add*: *pmf.rel-map rel-witness-spmf-def intro*: *pmf.rel-mono-strong*[*OF rel-witness-pmf1*[*OF assms*]] *rel-witness-option1 pmf.rel-mono-strong*[*OF rel-witness-pmf2*[*OF assms*]] *rel-witness-option2*)

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

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

lemma *enforce-spmf-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*A* ===> (=)) ===> *rel-spmf A* ===> *rel-spmf A*) *enforce-spmf enforce-spmf*
unfolding *enforce-spmf-def* **by** *transfer-prover*

lemma *enforce-return-spmf* [*simp*]:
enforce-spmf P (*return-spmf x*) = (*if P x then return-spmf x else return-pmf None*)
by(*simp add*: *enforce-spmf-def*)

lemma *enforce-return-pmf-None* [*simp*]:
enforce-spmf P (*return-pmf None*) = *return-pmf None*
by(*simp add*: *enforce-spmf-def*)

lemma *enforce-map-spmf*:
enforce-spmf P (*map-spmf f p*) = *map-spmf f* (*enforce-spmf* (*P* ∘ *f*) *p*)
by(*simp add*: *enforce-spmf-def pmf.map-comp o-def enforce-map-option*)

lemma *enforce-bind-spmf* [*simp*]:
enforce-spmf P (*bind-spmf p f*) = *bind-spmf p* (*enforce-spmf P* ∘ *f*)
by(*auto simp add*: *enforce-spmf-def bind-spmf-def map-bind-pmf intro*!: *bind-pmf-cong split*: *option.split*)

lemma *set-enforce-spmf* [*simp*]: *set-spmf* (*enforce-spmf P p*) = {*a* ∈ *set-spmf p*. *P a*}
by(*auto simp add*: *enforce-spmf-def in-set-spmf*)

lemma *enforce-spmf-alt-def*:
enforce-spmf P p = *bind-spmf p* (λ*a*. *bind-spmf* (*assert-spmf* (*P a*)) (λ- :: *unit*. *return-spmf a*))
by(*auto simp add*: *enforce-spmf-def assert-spmf-def map-pmf-def bind-spmf-def bind-return-pmf intro*!: *bind-pmf-cong split*: *option.split*)

lemma *bind-enforce-spmf* [*simp*]:
bind-spmf (*enforce-spmf P p*) *f* = *bind-spmf p* (λ*x*. *if P x then f x else return-pmf None*)
by(*auto simp add*: *enforce-spmf-alt-def assert-spmf-def intro*!: *bind-spmf-cong*)

lemma *weight-enforce-spmf*:
weight-spmf (*enforce-spmf P p*) = *weight-spmf p* − *measure* (*measure-spmf p*) {*x*. ¬ *P x*} (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *LINT x*|*measure-spmf p*. *indicator* {*x*. *P x*} *x*
by(*auto simp add*: *enforce-spmf-alt-def weight-bind-spmf o-def simp del*: *Bochner-Integration.integral-indicator intro*!: *Bochner-Integration.integral-cong split*: *split-indicator*)
also have ... = *?rhs*
by(*subst measure-spmf.finite-measure-Diff* [*symmetric*])(*auto simp add*: *space-measure-spmf intro*!: *arg-cong2* [**where** *f*=*measure*])
finally show *?thesis* **.**
qed

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

by(*auto simp add*: *enforce-spmf-alt-def*)

lemma *enforce-spmf-top* [*simp*]: *enforce-spmf* ⊤ = *id*
by(*simp add*: *enforce-spmf-def*)

lemma *enforce-spmf-K-True* [*simp*]: *enforce-spmf* (λ-. *True*) *p* = *p*
using *enforce-spmf-top*[*THEN fun-cong*, *of p*] **by**(*simp add*: *top-fun-def*)

lemma *enforce-spmf-bot* [*simp*]: *enforce-spmf* ⊥ = (λ-. *return-pmf None*)
by(*simp add*: *enforce-spmf-def fun-eq-iff*)

lemma *enforce-spmf-K-False* [*simp*]: *enforce-spmf* (λ-. *False*) *p* = *return-pmf None*
using *enforce-spmf-bot*[*THEN fun-cong*, *of p*] **by**(*simp add*: *bot-fun-def*)

lemma *enforce-pred-id-spmf*: *enforce-spmf P p* = *p* **if** *pred-spmf P p*
proof −
have *enforce-spmf P p* = *map-pmf id p* **using** *that*
by(*auto simp add*: *enforce-spmf-def enforce-pred-id-option simp del*: *map-pmf-id intro*!: *pmf.map-cong-pred*[*OF refl*] *elim*!: *pmf-pred-mono-strong*)
then show *?thesis* **by** *simp*
qed

lemma *map-the-spmf-of-pmf* [*simp*]: *map-pmf the* (*spmf-of-pmf p*) = *p*
by(*simp add*: *spmf-of-pmf-def pmf.map-comp o-def*)

lemma *bind-bind-conv-pair-spmf*:
bind-spmf p (λ*x*. *bind-spmf q* (*f x*)) = *bind-spmf* (*pair-spmf p q*) (λ(*x*, *y*). *f x y*)
by(*simp add*: *pair-spmf-alt-def*)

lemma *cond-spmf-spmf-of-set*:
cond-spmf (*spmf-of-set A*) *B* = *spmf-of-set* (*A* ∩ *B*) **if** *finite A*
by(*rule spmf-eqI*)(*auto simp add*: *spmf-of-set measure-spmf-of-set that split*: *split-indicator*)

lemma *pair-spmf-of-set*:
pair-spmf (*spmf-of-set A*) (*spmf-of-set B*) = *spmf-of-set* (*A* × *B*)
by(*rule spmf-eqI*)(*clarsimp simp add*: *spmf-of-set card-cartesian-product split*: *split-indicator*)

lemma *emeasure-cond-spmf*:
emeasure (*measure-spmf* (*cond-spmf p A*)) *B* = *emeasure* (*measure-spmf p*) (*A* ∩ *B*) / *emeasure* (*measure-spmf p*) *A*
apply(*clarsimp simp add*: *cond-spmf-def emeasure-measure-spmf-conv-measure-pmf emeasure-measure-pmf-zero-iff set-pmf-Int-Some split*!: *if-split*)
apply *blast*
apply(*subst* (*asm*) *emeasure-cond-pmf*)
by(*auto simp add*: *set-pmf-Int-Some image-Int*)

lemma *measure-cond-spmf*:
measure (*measure-spmf* (*cond-spmf p A*)) *B* = *measure* (*measure-spmf p*) (*A* ∩

B) / *measure* (*measure-spmf p*) *A*
apply(*clarsimp simp add*: *cond-spmf-def measure-measure-spmf-conv-measure-pmf measure-pmf-zero-iff set-pmf-Int-Some split*!: *if-split*)
apply(*subst* (*asm*) *measure-cond-pmf*)
by(*auto simp add*: *image-Int set-pmf-Int-Some*)

lemma *lossless-cond-spmf* [*simp*]: *lossless-spmf* (*cond-spmf p A*) ⟷ *set-spmf p* ∩ *A* ≠ {}
by(*clarsimp simp add*: *cond-spmf-def lossless-iff-set-pmf-None set-pmf-Int-Some*)

lemma *measure-spmf-eq-density*: *measure-spmf p* = *density* (*count-space UNIV*) (*spmf p*)
by(*rule measure-eqI*)(*simp-all add*: *emeasure-density nn-integral-spmf*[*symmetric*] *nn-integral-count-space-indicator*)

lemma *integral-measure-spmf*:
fixes *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
assumes *A*: *finite A*
shows (⋀*a*. *a* ∈ *set-spmf M* ⟹ *f a* ≠ *0* ⟹ *a* ∈ *A*) ⟹ (*LINT x*|*measure-spmf M*. *f x*) = (∑ *a*∈*A*. *spmf M a* $*_R$ *f a*)
unfolding *measure-spmf-eq-density*
apply (*simp add*: *integral-density*)
apply (*subst lebesgue-integral-count-space-finite-support*)
by (*auto intro*!: *finite-subset*[*OF* - ‹*finite A*›] *sum.mono-neutral-left simp*: *spmf-eq-0-set-spmf*)

lemma *image-set-spmf-eq*:
f ‘ *set-spmf p* = *g* ‘ *set-spmf q* **if** *ASSUMPTION* (*map-spmf f p* = *map-spmf g q*)
using *that*[*unfolded ASSUMPTION-def*, *THEN arg-cong*[**where** *f*=*set-spmf*]] **by** *simp*

lemma *map-spmf-const*: *map-spmf* (λ-. *x*) *p* = *scale-spmf* (*weight-spmf p*) (*return-spmf x*)
by(*simp add*: *map-spmf-conv-bind-spmf bind-spmf-const*)

lemma *cond-return-pmf* [*simp*]: *cond-pmf* (*return-pmf x*) *A* = *return-pmf x* **if** *x* ∈ *A*
using *that* **by**(*intro pmf-eqI*)(*auto simp add*: *pmf-cond split*: *split-indicator*)

lemma *cond-return-spmf* [*simp*]: *cond-spmf* (*return-spmf x*) *A* = (*if x* ∈ *A then return-spmf x else return-pmf None*)
by(*simp add*: *cond-spmf-def*)

lemma *measure-range-Some-eq-weight*:
measure (*measure-pmf p*) (*range Some*) = *weight-spmf p*
by (*simp add*: *measure-measure-spmf-conv-measure-pmf space-measure-spmf*)

lemma *restrict-spmf-eq-return-pmf-None* [*simp*]:
restrict-spmf p A = *return-pmf None* ⟷ *set-spmf p* ∩ *A* = {}
by(*auto 4 3 simp add*: *restrict-spmf-def map-pmf-eq-return-pmf-iff bind-UNION in-set-spmf bind-eq-None-conv option.the-def dest*: *bspec split*: *if-split-asm option.split-asm*)

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

lemma *mk-lossless-idem* [*simp*]: *mk-lossless* (*mk-lossless p*) = *mk-lossless p*
by(*simp add*: *mk-lossless-def weight-scale-spmf min-def max-def inverse-eq-divide*)

lemma *mk-lossless-return* [*simp*]: *mk-lossless* (*return-pmf x*) = *return-pmf x*
by(*cases x*)(*simp-all add*: *mk-lossless-def*)

lemma *mk-lossless-map* [*simp*]: *mk-lossless* (*map-spmf f p*) = *map-spmf f* (*mk-lossless p*)
by(*simp add*: *mk-lossless-def map-scale-spmf*)

lemma *spmf-mk-lossless* [*simp*]: *spmf* (*mk-lossless p*) *x* = *spmf p x* / *weight-spmf p*
by(*simp add*: *mk-lossless-def spmf-scale-spmf inverse-eq-divide max-def*)

lemma *set-spmf-mk-lossless* [*simp*]: *set-spmf* (*mk-lossless p*) = *set-spmf p*
by(*simp add*: *mk-lossless-def set-scale-spmf measure-spmf-zero-iff zero-less-measure-iff*)

lemma *mk-lossless-lossless* [*simp*]: *lossless-spmf p* ⟹ *mk-lossless p* = *p*
by(*simp add*: *mk-lossless-def lossless-weight-spmfD*)

lemma *mk-lossless-eq-return-pmf-None* [*simp*]: *mk-lossless p* = *return-pmf None* ⟷ *p* = *return-pmf None*
proof −
have *aux*: *weight-spmf p* = *0* ⟹ *spmf p i* = *0* **for** *i*
by(*rule antisym*, *rule order-trans*[*OF spmf-le-weight*]) (*auto intro*!: *order-trans*[*OF spmf-le-weight*])

have[*simp*]: *spmf* (*scale-spmf* (*inverse* (*weight-spmf p*)) *p*) = *spmf* (*return-pmf None*) ⟹ *spmf p i* = *0* **for** *i*
by(*drule fun-cong*[**where** *x*=*i*]) (*auto simp add*: *aux spmf-scale-spmf max-def*)

show *?thesis* **by**(*auto simp add*: *mk-lossless-def intro*: *spmf-eqI*)
qed

lemma *return-pmf-None-eq-mk-lossless* [*simp*]: *return-pmf None* = *mk-lossless p* ⟷ *p* = *return-pmf None*
by(*metis mk-lossless-eq-return-pmf-None*)

lemma *mk-lossless-spmf-of-set* [*simp*]: *mk-lossless* (*spmf-of-set A*) = *spmf-of-set A*
by(*simp add*: *spmf-of-set-def del*: *spmf-of-pmf-pmf-of-set*)

lemma *weight-mk-lossless*: *weight-spmf* (*mk-lossless p*) = (*if p* = *return-pmf None then 0 else 1*)
  by(*simp add*: *mk-lossless-def weight-scale-spmf min-def max-def inverse-eq-divide weight-spmf-eq-0*)

lemma *mk-lossless-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
  (*rel-spmf A* ===> *rel-spmf A*) *mk-lossless mk-lossless*
  by(*simp add*: *mk-lossless-def rel-fun-def rel-spmf-weightD rel-spmf-scaleI*)

lemma *rel-spmf-mk-losslessI*:
  rel-spmf A p q $\Longrightarrow$ *rel-spmf A* (*mk-lossless p*) (*mk-lossless q*)
  by(*rule mk-lossless-parametric*[*THEN rel-funD*])

lemma *rel-spmf-restrict-spmfI*:
  rel-spmf ($\lambda x\ y.$ ($x \in A \wedge y \in B \wedge R\ x\ y$) $\vee$ $x \notin A \wedge y \notin B$) *p q*
  $\Longrightarrow$ *rel-spmf R* (*restrict-spmf p A*) (*restrict-spmf q B*)
  by(*auto simp add*: *restrict-spmf-def pmf.rel-map elim*!: *option.rel-cases pmf.rel-mono-strong*)

lemma *cond-spmf-alt*: *cond-spmf p A* = *mk-lossless* (*restrict-spmf p A*)
proof(*cases set-spmf p* $\cap$ *A* = {})
  case *True*
  then show *?thesis* **by**(*simp add*: *cond-spmf-def measure-spmf-zero-iff*)
next
  case *False*
  show *?thesis*
   by(*rule spmf-eqI*)(*simp add*: *False cond-spmf-def pmf-cond set-pmf-Int-Some image-iff measure-measure-spmf-conv-measure-pmf*[*symmetric*] *spmf-scale-spmf max-def inverse-eq-divide*)
qed

lemma *cond-spmf-bind*:
  cond-spmf (*bind-spmf p f*) *A* = *mk-lossless* (*p* $\gg=$ ($\lambda x.$ *f x* $\upharpoonright$ *A*))
  by(*simp add*: *cond-spmf-alt restrict-bind-spmf scale-bind-spmf*)

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

lemma *cond-pmf-singleton*:
  cond-pmf p A = *return-pmf x* **if** *set-pmf p* $\cap$ *A* = {*x*}
proof −
  have[*simp*]: *set-pmf p* $\cap$ *A* = {*x*} $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *measure-pmf.prob p A* = *pmf p x*
    by(*auto simp add*: *measure-pmf-single*[*symmetric*] *AE-measure-pmf-iff intro*!: *measure-pmf.finite-measure-eq-AE*)

  have *pmf* (*cond-pmf p A*) *i* = *pmf* (*return-pmf x*) *i* **for** *i*
    using *that* **by**(*auto simp add*: *pmf-cond measure-pmf-zero-iff pmf-eq-0-set-pmf split*: *split-indicator*)

then show *?thesis* **by**(*rule pmf-eqI*)
qed

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

lemma *cond-spmf-fst-return-spmf* [*simp*]:
cond-spmf-fst (*return-spmf* (*x*, *y*)) *x* = *return-spmf y*
by(*simp add*: *cond-spmf-fst-def*)

lemma *cond-spmf-fst-map-Pair* [*simp*]: *cond-spmf-fst* (*map-spmf* (*Pair x*) *p*) *x* = *mk-lossless p*
by(*clarsimp simp add*: *cond-spmf-fst-def spmf.map-comp o-def*)

lemma *cond-spmf-fst-map-Pair'* [*simp*]: *cond-spmf-fst* (*map-spmf* (λ*y*. (*x*, *f y*)) *p*) *x* = *map-spmf f* (*mk-lossless p*)
by(*subst spmf.map-comp*[**where** *f*=*Pair x*, *symmetric*, *unfolded o-def*]) *simp*

lemma *cond-spmf-fst-eq-return-None* [*simp*]: *cond-spmf-fst p x* = *return-pmf None* ⟷ *x* ∉ *fst* ' *set-spmf p*
by(*auto 4 4 simp add*: *cond-spmf-fst-def map-pmf-eq-return-pmf-iff in-set-spmf*[*symmetric*] *dest*: *bspec*[**where** *x*=*Some* -] *intro*: *ccontr rev-image-eqI*)

lemma *cond-spmf-fst-map-Pair1*:
cond-spmf-fst (*map-spmf* (λ*x*. (*f x*, *g x*)) *p*) (*f x*) = *return-spmf* (*g* (*inv-into* (*set-spmf p*) *f* (*f x*)))
if *x* ∈ *set-spmf p inj-on f* (*set-spmf p*)
proof −
let *?foo*=λ*y*. *map-option* (λ*x*. (*f x*, *g x*)) −' *Some* ' ({*f y*} × *UNIV*)
have[*simp*]: *y* ∈ *set-spmf p* ⟹ *f x* = *f y* ⟹ *set-pmf p* ∩ (*?foo y*) ≠ {} **for** *y*
by(*auto simp add*: *vimage-def image-def in-set-spmf*)

have[*simp*]: *y* ∈ *set-spmf p* ⟹ *f x* = *f y* ⟹ *map-spmf snd* (*map-spmf* (λ*x*. (*f x*, *g x*)) (*cond-pmf p* (*?foo y*))) = *return-spmf* (*g x*) **for** *y*
using *that* **by**(*subst cond-pmf-singleton*[**where** *x*=*Some x*]) (*auto simp add*: *in-set-spmf elim*: *inj-onD*)

show *?thesis*
using *that*
by(*auto simp add*: *cond-spmf-fst-def cond-spmf-def*)
(*erule notE*, *subst cond-map-pmf*, *simp-all*)
qed

lemma *lossless-cond-spmf-fst* [*simp*]: *lossless-spmf* (*cond-spmf-fst p x*) ⟷ *x* ∈ *fst* ' *set-spmf p*
by(*auto simp add*: *cond-spmf-fst-def intro*: *rev-image-eqI*)

lemma *cond-spmf-fst-inverse*:
bind-spmf (*map-spmf fst p*) (λx. *map-spmf* (*Pair x*) (*cond-spmf-fst p x*)) = *p*
(**is** *?lhs* = *?rhs*)
proof(*rule spmf-eqI*)
fix *i* :: *'a* × *'b*
have ∗: ({*x*} × *UNIV* ∩ (*Pair x* ∘ *snd*) −‘ {*i*}) = (*if x* = *fst i then* {*i*} *else* {})
for *x* **by**(*cases i*)*auto*
have *spmf ?lhs i* = *LINT x*|*measure-spmf* (*map-spmf fst p*). *spmf* (*map-spmf* (*Pair x* ∘ *snd*) (*cond-spmf p* ({*x*} × *UNIV*))) *i*
by(*auto simp add*: *spmf-bind spmf.map-comp*[*symmetric*] *cond-spmf-fst-def intro*!: *integral-cong-AE*)
also have ... = *LINT x*|*measure-spmf* (*map-spmf fst p*). *measure* (*measure-spmf* (*cond-spmf p* ({*x*} × *UNIV*))) ((*Pair x* ∘ *snd*) −‘ {*i*})
by(*rule integral-cong-AE*)(*auto simp add*: *spmf-map*)
also have ... = *LINT x*|*measure-spmf* (*map-spmf fst p*). *measure* (*measure-spmf p*) ({*x*} × *UNIV* ∩ (*Pair x* ∘ *snd*) −‘ {*i*}) /
measure (*measure-spmf p*) ({*x*} × *UNIV*)
by(*rule integral-cong-AE*; *clarsimp simp add*: *measure-cond-spmf*)
also have ... = *spmf* (*map-spmf fst p*) (*fst i*) ∗ *spmf p i* / *measure* (*measure-spmf p*) ({*fst i*} × *UNIV*)
by(*simp add*: ∗ *if-distrib*[**where** *f*=*measure* (*measure-spmf* -)] *cong*: *if-cong*)
(*subst integral-measure-spmf*[**where** *A*={*fst i*}]; *auto split*: *if-split-asm simp add*: *spmf-conv-measure-spmf*)
also have ... = *spmf p i*
by(*clarsimp simp add*: *spmf-map vimage-fst*)(*metis* (*no-types*, *lifting*) *Int-insert-left-if1 in-set-spmf-iff-spmf insertI1 insert-UNIV insert-absorb insert-not-empty measure-spmf-zero-iff mem-Sigma-iff prod.collapse*)
finally show *spmf ?lhs i* = *spmf ?rhs i* **.**
qed

1.21.1 Embedding of *'a option* into *'a spmf*

This theoretically follows from the embedding between - *Monomorphic-Monad.id* into - *prob* and the isomorphism between (-, - *prob*) *optionT* and - *spmf*, but we would only get the monomorphic version via this connection. So we do it directly.

lemma *bind-option-spmf-monad* [*simp*]: *monad.bind-option* (*return-pmf None*) *x* = *bind-spmf* (*return-pmf x*)
by(*cases x*)(*simp-all add*: *fun-eq-iff*)

locale *option-to-spmf* **begin**

We have to get the embedding into the lifting package such that we can use the parametrisation of transfer rules.

definition *the-pmf* :: *'a pmf* ⇒ *'a* **where** *the-pmf p* = (*THE x*. *p* = *return-pmf x*)

lemma *the-pmf-return* [*simp*]: *the-pmf* (*return-pmf x*) = *x*

by(*simp add*: *the-pmf-def*)

lemma *type-definition-option-spmf*: *type-definition return-pmf the-pmf* {*x*. ∃ *y* :: ′*a option*. *x* = *return-pmf y*}
by *unfold-locales*(*auto*)

context begin
private setup-lifting *type-definition-option-spmf*
abbreviation *cr-spmf-option* **where** *cr-spmf-option* ≡ *cr-option*
abbreviation *pcr-spmf-option* **where** *pcr-spmf-option* ≡ *pcr-option*
lemmas *Quotient-spmf-option* = *Quotient-option*
  and *cr-spmf-option-def* = *cr-option-def*
  and *pcr-spmf-option-bi-unique* = *option.bi-unique*
  and *Domainp-pcr-spmf-option* = *option.domain*
  and *Domainp-pcr-spmf-option-eq* = *option.domain-eq*
  and *Domainp-pcr-spmf-option-par* = *option.domain-par*
  and *Domainp-pcr-spmf-option-left-total* = *option.domain-par-left-total*
  and *pcr-spmf-option-left-unique* = *option.left-unique*
  and *pcr-spmf-option-cr-eq* = *option.pcr-cr-eq*
  and *pcr-spmf-option-return-pmf-transfer* = *option.rep-transfer*
  and *pcr-spmf-option-right-total* = *option.right-total*
  and *pcr-spmf-option-right-unique* = *option.right-unique*
  and *pcr-spmf-option-def* = *pcr-option-def*
bundle *spmf-option-lifting* = [[*Lifting.lifting-restore-internal Misc-CryptHOL.option.lifting*]]
end

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

lemma *return-option-spmf-transfer* [*transfer-parametric return-spmf-parametric*, *transfer-rule*]:
  ((=) ===> *cr-spmf-option*) *return-spmf Some*
by(*rule rel-funI*)(*simp add*: *cr-spmf-option-def*)

lemma *map-option-spmf-transfer* [*transfer-parametric map-spmf-parametric*, *transfer-rule*]:
  (((=) ===> (=)) ===> *cr-spmf-option* ===> *cr-spmf-option*) *map-spmf map-option*
unfolding *rel-fun-eq* **by**(*auto simp add*: *rel-fun-def cr-spmf-option-def*)

lemma *fail-option-spmf-transfer* [*transfer-parametric return-spmf-None-parametric*, *transfer-rule*]:
  cr-spmf-option (*return-pmf None*) *None*
by(*simp add*: *cr-spmf-option-def*)

lemma *bind-option-spmf-transfer* [*transfer-parametric bind-spmf-parametric*, *transfer-rule*]:
  (*cr-spmf-option* ===> ((=) ===> *cr-spmf-option*) ===> *cr-spmf-option*) *bind-spmf Option.bind*

apply(*clarsimp simp add*: *rel-fun-def cr-spmf-option-def*)
subgoal for *x f g* **by**(*cases x*; *simp*)
done

lemma *set-option-spmf-transfer* [*transfer-parametric set-spmf-parametric*, *transfer-rule*]:
(*cr-spmf-option* ===> *rel-set* (=)) *set-spmf set-option*
by(*clarsimp simp add*: *rel-fun-def cr-spmf-option-def rel-set-eq*)

lemma *rel-option-spmf-transfer* [*transfer-parametric rel-spmf-parametric*, *transfer-rule*]:
(((=) ===> (=) ===> (=)) ===> *cr-spmf-option* ===> *cr-spmf-option* ===> (=)) *rel-spmf rel-option*
unfolding *rel-fun-eq* **by**(*simp add*: *rel-fun-def cr-spmf-option-def*)

end

end

locale *option-le-spmf* **begin**

Embedding where only successful computations in the option monad are related to Dirac spmf.

definition *cr-option-le-spmf* :: *'a option* ⇒ *'a spmf* ⇒ *bool*
where *cr-option-le-spmf x p* ⟷ *ord-spmf* (=) (*return-pmf x*) *p*

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

lemma *return-option-le-spmf-transfer* [*transfer-rule*]:
((=) ===> *cr-option-le-spmf*) (λ*x*. *x*) *return-pmf*
by(*rule rel-funI*)(*simp add*: *cr-option-le-spmf-def ord-option-reflI*)

lemma *map-option-le-spmf-transfer* [*transfer-rule*]:
(((=) ===> (=)) ===> *cr-option-le-spmf* ===> *cr-option-le-spmf*) *map-option map-spmf*
unfolding *rel-fun-eq*
apply(*clarsimp simp add*: *rel-fun-def cr-option-le-spmf-def rel-pmf-return-pmf1 ord-option-map1 ord-option-map2*)
subgoal for *f x p y* **by**(*cases x*; *simp add*: *ord-option-reflI*)
done

lemma *bind-option-le-spmf-transfer* [*transfer-rule*]:
(*cr-option-le-spmf* ===> ((=) ===> *cr-option-le-spmf*) ===> *cr-option-le-spmf*) *Option.bind bind-spmf*
apply(*clarsimp simp add*: *rel-fun-def cr-option-le-spmf-def*)
subgoal for *x p f g* **by**(*cases x*; *auto 4 3 simp add*: *rel-pmf-return-pmf1 set-pmf-bind-spmf*)
done

end

end

interpretation *rel-spmf-characterisation* **by** *unfold-locales*(*rule rel-pmf-measureI*)

lemma *if-distrib-bind-spmf1* [*if-distribs*]:
bind-spmf (*if b then x else y*) *f* = (*if b then bind-spmf x f else bind-spmf y f*)
by *simp*

lemma *if-distrib-bind-spmf2* [*if-distribs*]:
bind-spmf x (λ*y*. *if b then f y else g y*) = (*if b then bind-spmf x f else bind-spmf x g*)
by *simp*

lemma *rel-spmf-if-distrib* [*if-distribs*]:
rel-spmf R (*if b then x else y*) (*if b then x′ else y′*) ⟷
(*b* ⟶ *rel-spmf R x x′*) ∧ (¬ *b* ⟶ *rel-spmf R y y′*)
by(*simp*)

lemma *if-distrib-map-spmf* [*if-distribs*]:
map-spmf f (*if b then p else q*) = (*if b then map-spmf f p else map-spmf f q*)
by *simp*

lemma *if-distrib-restrict-spmf1* [*if-distribs*]:
restrict-spmf (*if b then p else q*) *A* = (*if b then restrict-spmf p A else restrict-spmf q A*)
by *simp*

end
theory *Set-Applicative* **imports**
Applicative-Lifting.Applicative-Set
begin

1.22 Applicative instance for *′a set*

lemma *ap-set-conv-bind*: *ap-set f x* = *Set.bind f* (λ*f*. *Set.bind x* (λ*x*. {*f x*}))
by(*auto simp add*: *ap-set-def bind-UNION*)

context includes *applicative-syntax* **begin**

lemma *in-ap-setI*: ⟦ *f′* ∈ *f*; *x′* ∈ *x* ⟧ ⟹ *f′ x′* ∈ *f* ⋄ *x*
by(*auto simp add*: *ap-set-def*)

lemma *in-ap-setE* [*elim*!]:
⟦ *x* ∈ *f* ⋄ *y*; ⋀*f′ y′*. ⟦ *x* = *f′ y′*; *f′* ∈ *f*; *y′* ∈ *y* ⟧ ⟹ *thesis* ⟧ ⟹ *thesis*
by(*auto simp add*: *ap-set-def*)

lemma *in-ap-pure-set* [*iff*]: *x* ∈ {*f*} ⋄ *y* ⟷ (∃ *y′*∈*y*. *x* = *f y′*)
unfolding *ap-set-def* **by** *auto*

end

end
theory *SPMF-Applicative* **imports**
Applicative-Lifting.Applicative-PMF
Set-Applicative
HOL−Probability.SPMF
begin

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

1.23 Applicative instance for *'a spmf*

abbreviation (*input*) *pure-spmf* :: *'a* ⇒ *'a spmf*
where *pure-spmf* ≡ *return-spmf*

definition *ap-spmf* :: (*'a* ⇒ *'b*) *spmf* ⇒ *'a spmf* ⇒ *'b spmf*
where *ap-spmf f x* = *map-spmf* (λ(*f*, *x*). *f x*) (*pair-spmf f x*)

lemma *ap-spmf-conv-bind*: *ap-spmf f x* = *bind-spmf f* (λ*f*. *bind-spmf x* (λ*x*. *return-spmf* (*f x*)))
by(*simp add*: *ap-spmf-def map-spmf-conv-bind-spmf pair-spmf-alt-def*)

adhoc-overloading *Applicative.ap* ⇌ *ap-spmf*

context includes *applicative-syntax* **begin**

lemma *ap-spmf-id*: *pure-spmf* (λ*x*. *x*) ⋄ *x* = *x*
by(*simp add*: *ap-spmf-def pair-spmf-return-spmf1 spmf.map-comp o-def*)

lemma *ap-spmf-comp*: *pure-spmf* (∘) ⋄ *u* ⋄ *v* ⋄ *w* = *u* ⋄ (*v* ⋄ *w*)
by(*simp add*: *ap-spmf-def pair-spmf-return-spmf1 pair-map-spmf1 pair-map-spmf2 spmf.map-comp o-def split-def pair-pair-spmf*)

lemma *ap-spmf-homo*: *pure-spmf f* ⋄ *pure-spmf x* = *pure-spmf* (*f x*)
by(*simp add*: *ap-spmf-def pair-spmf-return-spmf1*)

lemma *ap-spmf-interchange*: *u* ⋄ *pure-spmf x* = *pure-spmf* (λ*f*. *f x*) ⋄ *u*
by(*simp add*: *ap-spmf-def pair-spmf-return-spmf1 pair-spmf-return-spmf2 spmf.map-comp o-def*)

lemma *ap-spmf-C*: *return-spmf* (λ*f x y*. *f y x*) ⋄ *f* ⋄ *x* ⋄ *y* = *f* ⋄ *y* ⋄ *x*
apply(*simp add*: *ap-spmf-def pair-map-spmf1 spmf.map-comp pair-spmf-return-spmf1 pair-pair-spmf o-def split-def*)
apply(*subst* (*2*) *pair-commute-spmf*)
apply(*simp add*: *pair-map-spmf2 spmf.map-comp o-def split-def*)
done

applicative *spmf* (*C*)

for
  pure: *pure-spmf*
  ap: *ap-spmf*
by(*rule ap-spmf-id ap-spmf-comp*[*unfolded o-def*[*abs-def*]] *ap-spmf-homo ap-spmf-interchange ap-spmf-C*)+

lemma *set-ap-spmf* [*simp*]: *set-spmf* ($p \diamond q$) = *set-spmf* $p \diamond$ *set-spmf* q
by(*auto simp add*: *ap-spmf-def ap-set-def*)

lemma *bind-ap-spmf*: *bind-spmf* ($p \diamond x$) f = *bind-spmf* p ($\lambda p.\ x \ggg (\lambda x.\ f\ (p\ x))$)
by(*simp add*: *ap-spmf-conv-bind*)

lemma *bind-pmf-ap-return-spmf* [*simp*]: *bind-pmf* (*ap-spmf* (*return-spmf* f) p) g = *bind-pmf* p ($g \circ$ *map-option* f)
by(*auto simp add*: *ap-spmf-conv-bind bind-spmf-def bind-return-pmf bind-assoc-pmf intro*: *bind-pmf-cong split*: *option.split*)

lemma *map-spmf-conv-ap* [*applicative-unfold*]: *map-spmf* f p = *return-spmf* $f \diamond p$
by(*simp add*: *map-spmf-conv-bind-spmf ap-spmf-conv-bind*)

end

end

1.24 Exclusive or on lists

theory *List-Bits* **imports** *Misc-CryptHOL* **begin**

definition *xor* :: $'a \Rightarrow 'a \Rightarrow 'a$ :: {*uminus*,*inf*,*sup*} (**infixr** ‹⊕› *67*)
where $x \oplus y$ = *inf* (*sup* x y) ($-$ (*inf* x y))

lemma *xor-bool-def* [*iff*]: **fixes** x y :: *bool* **shows** $x \oplus y \longleftrightarrow x \neq y$
by(*auto simp add*: *xor-def*)

lemma *xor-commute*:
  fixes x y :: $'a$ :: {*semilattice-sup*,*semilattice-inf*,*uminus*}
  shows $x \oplus y = y \oplus x$
by(*simp add*: *xor-def sup.commute inf.commute*)

lemma *xor-assoc*:
  fixes x y :: $'a$ :: *boolean-algebra*
  shows $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
by(*simp add*: *xor-def inf-sup-aci inf-sup-distrib1 inf-sup-distrib2*)

lemma *xor-left-commute*:
  fixes x y :: $'a$ :: *boolean-algebra*
  shows $x \oplus (y \oplus z) = y \oplus (x \oplus z)$
by (*metis xor-assoc xor-commute*)

lemma [*simp*]:
fixes *x* :: *'a* :: *boolean-algebra*
shows *xor-bot*: *x* ⊕ *bot* = *x*
and *bot-xor*: *bot* ⊕ *x* = *x*
and *xor-top*: *x* ⊕ *top* = − *x*
and *top-xor*: *top* ⊕ *x* = − *x*
by(*simp-all add*: *xor-def*)

lemma *xor-inverse* [*simp*]:
fixes *x* :: *'a* :: *boolean-algebra*
shows *x* ⊕ *x* = *bot*
by(*simp add*: *xor-def*)

lemma *xor-left-inverse* [*simp*]:
fixes *x* :: *'a* :: *boolean-algebra*
shows *x* ⊕ *x* ⊕ *y* = *y*
by(*metis xor-left-commute xor-inverse xor-bot*)

lemmas *xor-ac* = *xor-assoc xor-commute xor-left-commute*

definition *xor-list* :: *'a* :: {*uminus*,*inf*,*sup*} *list* ⇒ *'a list* ⇒ *'a list* (**infixr** ‹[⊕]› *67*)
where *xor-list xs ys* = *map* (*case-prod* (⊕)) (*zip xs ys*)

lemma *xor-list-unfold*:
xs [⊕] *ys* = (*case xs of* [] ⇒ [] | *x* # *xs'* ⇒ (*case ys of* [] ⇒ [] | *y* # *ys'* ⇒ *x* ⊕ *y* # *xs'* [⊕] *ys'*))
by(*simp add*: *xor-list-def split*: *list.split*)

lemma *xor-list-commute*: **fixes** *xs ys* :: *'a* :: {*semilattice-sup*,*semilattice-inf*,*uminus*} *list*
shows *xs* [⊕] *ys* = *ys* [⊕] *xs*
unfolding *xor-list-def* **by**(*subst zip-commute*)(*auto simp add*: *split-def xor-commute*)

lemma *xor-list-assoc* [*simp*]:
fixes *xs ys* :: *'a* :: *boolean-algebra list*
shows (*xs* [⊕] *ys*) [⊕] *zs* = *xs* [⊕] (*ys* [⊕] *zs*)
unfolding *xor-list-def zip-map1 zip-map2*
apply(*subst* (*2*) *zip-commute*)
apply(*subst zip-left-commute*)
apply(*subst* (*2*) *zip-commute*)
apply(*auto simp add*: *zip-map2 split-def xor-assoc*)
done

lemma *xor-list-left-commute*:
fixes *xs ys zs* :: *'a* :: *boolean-algebra list*
shows *xs* [⊕] (*ys* [⊕] *zs*) = *ys* [⊕] (*xs* [⊕] *zs*)
by(*metis xor-list-assoc xor-list-commute*)

lemmas *xor-list-ac* = *xor-list-assoc xor-list-commute xor-list-left-commute*

lemma *xor-list-inverse* [*simp*]:
  fixes *xs* :: *'a* :: *boolean-algebra list*
  shows *xs* [⊕] *xs* = *replicate* (*length xs*) *bot*
by(*simp add*: *xor-list-def zip-same-conv-map o-def map-replicate-const*)

lemma *xor-replicate-bot-right* [*simp*]:
  fixes *xs* :: *'a* :: *boolean-algebra list*
  shows ⟦ *length xs* ≤ *n*; *x* = *bot* ⟧ ⟹ *xs* [⊕] *replicate n x* = *xs*
by(*simp add*: *xor-list-def zip-replicate2 o-def*)

lemma *xor-replicate-bot-left* [*simp*]:
  fixes *xs* :: *'a* :: *boolean-algebra list*
  shows ⟦ *length xs* ≤ *n*; *x* = *bot* ⟧ ⟹ *replicate n x* [⊕] *xs* = *xs*
by(*simp add*: *xor-list-commute*)

lemma *xor-list-left-inverse* [*simp*]:
  fixes *xs* :: *'a* :: *boolean-algebra list*
  shows *length ys* ≤ *length xs* ⟹ *xs* [⊕] (*xs* [⊕] *ys*) = *ys*
by(*subst xor-list-assoc*[*symmetric*])(*simp*)

lemma *length-xor-list* [*simp*]: *length* (*xor-list xs ys*) = *min* (*length xs*) (*length ys*)
by(*simp add*: *xor-list-def*)

lemma *inj-on-xor-list-nlists* [*simp*]:
  fixes *xs* :: *'a* :: *boolean-algebra list*
  shows *n* ≤ *length xs* ⟹ *inj-on* (*xor-list xs*) (*nlists UNIV n*)
apply(*clarsimp simp add*: *inj-on-def in-nlists-UNIV*)
using *xor-list-left-inverse* **by** *fastforce*

lemma *one-time-pad*:
  fixes *xs* :: - :: *boolean-algebra list*
  shows *length xs* ≥ *n* ⟹ *map-spmf* (*xor-list xs*) (*spmf-of-set* (*nlists UNIV n*))
= *spmf-of-set* (*nlists UNIV n*)
by(*auto 4 3 simp add*: *in-nlists-UNIV intro*: *xor-list-left-inverse*[*symmetric*] *rev-image-eqI intro*!: *arg-cong*[**where** *f*=*spmf-of-set*])

end
theory *Environment-Functor* **imports**
  Applicative-Lifting.Applicative-Environment
begin

1.25 The environment functor

type-synonym (*'i*, *'a*) *envir* = *'i* ⇒ *'a*

lemma *const-apply* [*simp*]: *const x i* = *x*

by(*simp add*: *const-def*)

context includes *applicative-syntax* **begin**

lemma *ap-envir-apply* [*simp*]: $(f \diamond x)\ i = f\ i\ (x\ i)$
by(*simp add*: *apf-def*)

definition *all-envir* :: $('i,\ bool)\ envir \Rightarrow bool$
where *all-envir p* $\longleftrightarrow (\forall\, x.\ p\ x)$

lemma *all-envirI* [*Pure.intro*!, *intro*!]: $(\bigwedge x.\ p\ x) \Longrightarrow$ *all-envir p*
by(*simp add*: *all-envir-def*)

lemma *all-envirE* [*Pure.elim 2*, *elim*]: *all-envir p* $\Longrightarrow (p\ x \Longrightarrow thesis) \Longrightarrow thesis$
by(*simp add*: *all-envir-def*)

lemma *all-envirD*: *all-envir p* $\Longrightarrow p\ x$
by(*simp add*: *all-envir-def*)

definition *pred-envir* :: $('a \Rightarrow bool) \Rightarrow ('i,\ 'a)\ envir \Rightarrow bool$
where *pred-envir p f* = *all-envir* (*const p* $\diamond$ *f*)

lemma *pred-envir-conv*: *pred-envir p f* $\longleftrightarrow (\forall\, x.\ p\ (f\ x))$
by(*auto simp add*: *pred-envir-def*)

lemma *pred-envirI* [*Pure.intro*!, *intro*!]: $(\bigwedge x.\ p\ (f\ x)) \Longrightarrow$ *pred-envir p f*
by(*auto simp add*: *pred-envir-def*)

lemma *pred-envirD*: *pred-envir p f* $\Longrightarrow p\ (f\ x)$
by(*auto simp add*: *pred-envir-def*)

lemma *pred-envirE* [*Pure.elim 2*, *elim*]: *pred-envir p f* $\Longrightarrow (p\ (f\ x) \Longrightarrow thesis)$ $\Longrightarrow thesis$
by(*simp add*: *pred-envir-conv*)

lemma *pred-envir-mono*: $[\![$ *pred-envir p f*; $\bigwedge x.\ p\ (f\ x) \Longrightarrow q\ (g\ x)$ $]\!] \Longrightarrow$ *pred-envir q g*
by *blast*

definition *rel-envir* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('i,\ 'a)\ envir \Rightarrow ('i,\ 'b)\ envir \Rightarrow bool$
where *rel-envir p f g* $\longleftrightarrow$ *all-envir* (*const p* $\diamond$ *f* $\diamond$ *g*)

lemma *rel-envir-conv*: *rel-envir p f g* $\longleftrightarrow (\forall\, x.\ p\ (f\ x)\ (g\ x))$
by(*auto simp add*: *rel-envir-def*)

lemma *rel-envir-conv-rel-fun*: *rel-envir* = *rel-fun* (=)
by(*simp add*: *rel-envir-conv rel-fun-def fun-eq-iff*)

lemma *rel-envirI* [*Pure.intro*!, *intro*!]: ($\bigwedge x.\ p\ (f\ x)\ (g\ x)$) $\Longrightarrow$ *rel-envir p f g*
by(*auto simp add*: *rel-envir-def*)

lemma *rel-envirD*: *rel-envir p f g* $\Longrightarrow$ *p* (*f x*) (*g x*)
by(*auto simp add*: *rel-envir-def*)

lemma *rel-envirE* [*Pure.elim 2*, *elim*]: *rel-envir p f g* $\Longrightarrow$ (*p* (*f x*) (*g x*) $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*
by(*simp add*: *rel-envir-conv*)

lemma *rel-envir-mono*: $\llbracket$ *rel-envir p f g*; $\bigwedge x.\ p\ (f\ x)\ (g\ x) \Longrightarrow q\ (f'\ x)\ (g'\ x)$ $\rrbracket \Longrightarrow$ *rel-envir q f′ g′*
by *blast*

lemma *rel-envir-mono1*: $\llbracket$ *pred-envir p f*; $\bigwedge x.\ p\ (f\ x) \Longrightarrow q\ (f'\ x)\ (g'\ x)$ $\rrbracket \Longrightarrow$ *rel-envir q f′ g′*
by *blast*

lemma *pred-envir-mono2*: $\llbracket$ *rel-envir p f g*; $\bigwedge x.\ p\ (f\ x)\ (g\ x) \Longrightarrow q\ (f'\ x)$ $\rrbracket \Longrightarrow$ *pred-envir q f′*
by *blast*

end

end

theory *Partial-Function-Set* **imports** *Main* **begin**

1.26 Setup for *partial-function* for sets

lemma (**in** *complete-lattice*) *lattice-partial-function-definition*:
partial-function-definitions (≤) *Sup*
by(*unfold-locales*)(*auto intro*: *Sup-upper Sup-least*)

interpretation *set*: *partial-function-definitions* (⊆) *Union*
by(*rule lattice-partial-function-definition*)

lemma *fun-lub-Sup*: *fun-lub Sup* = (*Sup* :: - ⇒ - :: *complete-lattice*)
by(*fastforce simp add*: *fun-lub-def fun-eq-iff Sup-fun-def intro*: *Sup-eqI SUP-upper SUP-least*)

lemma *set-admissible*: *set.admissible* (λf :: ′*a* ⇒ ′*b set*. $\forall x\ y.\ y \in f\ x \longrightarrow P\ x\ y$)
by(*rule ccpo.admissibleI*)(*auto simp add*: *fun-lub-Sup*)

abbreviation *mono-set* ≡ *monotone* (*fun-ord* (⊆)) (⊆)

lemma *fixp-induct-set-scott*:
fixes *F* :: ′*c* ⇒ ′*c*

and *U* :: *'c* ⇒ *'b* ⇒ *'a set*
and *C* :: (*'b* ⇒ *'a set*) ⇒ *'c*
and *P* :: *'b* ⇒ *'a* ⇒ *bool*
and *x* **and** *y*
assumes *mono*: ⋀*x*. *mono-set* (λ*f*. *U* (*F* (*C f*)) *x*)
and *eq*: *f* ≡ *C* (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* (≤)) (λ*f*. *U* (*F* (*C f*))))
and *inverse2*: ⋀*f*. *U* (*C f*) = *f*
and *step*: ⋀*f x y*. ⟦ ⋀*x y*. *y* ∈ *U f x* ⟹ *P x y*; *y* ∈ *U* (*F f*) *x* ⟧ ⟹ *P x y*
and *enforce-variable-ordering*: *x* = *x*
and *elem*: *y* ∈ *U f x*
shows *P x y*
using *step elem set.fixp-induct-uc*[*of U F C*, *OF mono eq inverse2 set-admissible*, *of P*]
by *blast*

lemma *fixp-Sup-le*:
defines *le* ≡ ((≤) :: - :: *complete-lattice* ⇒ -)
shows *ccpo.fixp Sup le* = *ccpo-class.fixp*
proof –
have *class.ccpo Sup le* (<) **unfolding** *le-def* **by** *unfold-locales*
thus *?thesis*
by(*simp add*: *ccpo.fixp-def fixp-def ccpo.iterates-def iterates-def ccpo.iteratesp-def iteratesp-def fun-eq-iff le-def*)
qed

lemma *fun-ord-le*: *fun-ord* (≤) = (≤)
by(*auto simp add*: *fun-ord-def fun-eq-iff le-fun-def*)

lemma *fixp-induct-set*:
fixes *F* :: *'c* ⇒ *'c*
and *U* :: *'c* ⇒ *'b* ⇒ *'a set*
and *C* :: (*'b* ⇒ *'a set*) ⇒ *'c*
and *P* :: *'b* ⇒ *'a* ⇒ *bool*
and *x* **and** *y*
assumes *mono*: ⋀*x*. *mono-set* (λ*f*. *U* (*F* (*C f*)) *x*)
and *eq*: *f* ≡ *C* (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* (≤)) (λ*f*. *U* (*F* (*C f*))))
and *inverse2*: ⋀*f*. *U* (*C f*) = *f*

and *step*: ⋀*f' x y*. ⟦ ⋀*x*. *U f' x* = *U f' x*; *y* ∈ *U* (*F* (*C* (*inf* (*U f*) (λ*x*. {*y*. *P x y*})))) *x* ⟧ ⟹ *P x y*
— partial_function requires a quantifier over f', so let's have a fake one
and *elem*: *y* ∈ *U f x*
shows *P x y*
proof –
from *mono*
have *mono'*: *mono* (λ*f*. *U* (*F* (*C f*)))
by(*simp add*: *fun-ord-le mono-def le-fun-def*)
hence *eq'*: *f* ≡ *C* (*lfp* (λ*f*. *U* (*F* (*C f*))))

using *eq* **unfolding** *fun-ord-le fun-lub-Sup fixp-Sup-le* **by**(*simp add*: *lfp-eq-fixp*)

let *?f* = *C* (*lfp* (λ*f*. *U* (*F* (*C f*))))
have *step′*: ⋀*x y*. ⟦ *y* ∈ *U* (*F* (*C* (*inf* (*U ?f*) (λ*x*. {*y*. *P x y*})))) *x* ⟧ ⟹ *P x y*
unfolding *eq′*[*symmetric*] **by**(*rule step*[*OF refl*])

let *?P* = λ*x*. {*y*. *P x y*}
from *mono′* **have** *lfp* (λ*f*. *U* (*F* (*C f*))) ≤ *?P*
by(*rule lfp-induct*)(*auto intro*!: *le-funI step′ simp add*: *inverse2*)
with *elem* **show** *?thesis*
by(*subst* (*asm*) *eq′*)(*auto simp add*: *inverse2 le-fun-def*)
qed

declaration ‹*Partial-Function.init set* @{*term set.fixp-fun*}
@{*term set.mono-body*} @{*thm set.fixp-rule-uc*} @{*thm set.fixp-induct-uc*}
(*SOME* @{*thm fixp-induct-set*})›

lemma [*partial-function-mono*]:
shows *insert-mono*: *mono-set A* ⟹ *mono-set* (λ*f*. *insert x* (*A f*))
and *UNION-mono*: ⟦*mono-set B*; ⋀*y*. *mono-set* (λ*f*. *C y f*)⟧ ⟹ *mono-set* (λ*f*. ⋃*y*∈*B f*. *C y f*)
and *set-bind-mono*: ⟦*mono-set B*; ⋀*y*. *mono-set* (λ*f*. *C y f*)⟧ ⟹ *mono-set* (λ*f*. *Set.bind* (*B f*) (λ*y*. *C y f*))
and *Un-mono*: ⟦ *mono-set A*; *mono-set B* ⟧ ⟹ *mono-set* (λ*f*. *A f* ∪ *B f*)
and *Int-mono*: ⟦ *mono-set A*; *mono-set B* ⟧ ⟹ *mono-set* (λ*f*. *A f* ∩ *B f*)
and *Diff-mono1*: *mono-set A* ⟹ *mono-set* (λ*f*. *A f* − *X*)
and *image-mono*: *mono-set A* ⟹ *mono-set* (λ*f*. *g* ‘ *A f*)
and *vimage-mono*: *mono-set A* ⟹ *mono-set* (λ*f*. *g* −‘ *A f*)
unfolding *bind-UNION* **by**(*fast intro*!: *monotoneI dest*: *monotoneD*)+

partial-function (*set*) *test* :: *′a list* ⇒ *nat* ⇒ *bool* ⇒ *int set*
where
test xs i j = *insert 4* (*test* [] *0 j* ∪ *test* [] *1 True* ∩ *test* [] *2 False* − {*5*} ∪ *uminus* ‘ *test* [*undefined*] *0 True* ∪ *uminus* −‘ *test* [] *1 False*)

interpretation *coset*: *partial-function-definitions* (⊇) *Inter*
by(*rule complete-lattice.lattice-partial-function-definition*[*OF dual-complete-lattice*])

lemma *fun-lub-Inf*: *fun-lub Inf* = (*Inf* :: - ⇒ - :: *complete-lattice*)
by(*auto simp add*: *fun-lub-def fun-eq-iff Inf-fun-def intro*: *Inf-eqI INF-lower INF-greatest*)

lemma *fun-ord-ge*: *fun-ord* (≥) = (≥)
by(*auto simp add*: *fun-ord-def fun-eq-iff le-fun-def*)

lemma *coset-admissible*: *coset.admissible* (λ*f* :: *′a* ⇒ *′b set*. ∀ *x y*. *P x y* ⟶ *y* ∈ *f x*)
by(*rule ccpo.admissibleI*)(*auto simp add*: *fun-lub-Inf*)

abbreviation *mono-coset* ≡ *monotone* (*fun-ord* (⊇)) (⊇)

lemma *gfp-eq-fixp*:
  fixes *f* :: *'a* :: *complete-lattice* ⇒ *'a*
  assumes *f*: *monotone* (≥) (≥) *f*
  shows *gfp f* = *ccpo.fixp Inf* (≥) *f*
proof (*rule antisym*)
  from *f* **have** *f'*: *mono f* **by**(*simp add*: *mono-def monotone-def*)

  interpret *ccpo Inf* (≥) *mk-less* (≥) :: *'a* ⇒ -
   by(*rule ccpo*)(*rule complete-lattice.lattice-partial-function-definition*[*OF dual-complete-lattice*])
  show *ccpo.fixp Inf* (≥) *f* ≤ *gfp f*
    by(*rule gfp-upperbound*)(*subst fixp-unfold*[*OF f*], *rule order-refl*)

  show *gfp f* ≤ *ccpo.fixp Inf* (≥) *f*
    by(*rule fixp-lowerbound*[*OF f*])(*subst gfp-unfold*[*OF f'*], *rule order-refl*)
qed

lemma *fixp-coinduct-set*:
  fixes *F* :: *'c* ⇒ *'c*
  and *U* :: *'c* ⇒ *'b* ⇒ *'a set*
  and *C* :: (*'b* ⇒ *'a set*) ⇒ *'c*
  and *P* :: *'b* ⇒ *'a* ⇒ *bool*
  and *x* **and** *y*
  assumes *mono*: ⋀*x*. *mono-coset* (λ*f*. *U* (*F* (*C f*)) *x*)
  and *eq*: *f* ≡ *C* (*ccpo.fixp* (*fun-lub Inter*) (*fun-ord* (≥)) (λ*f*. *U* (*F* (*C f*))))
  and *inverse2*: ⋀*f*. *U* (*C f*) = *f*

  and *step*: ⋀*f' x y*. ⟦ ⋀*x*. *U f' x* = *U f' x*; ¬ *P x y* ⟧ ⟹ *y* ∈ *U* (*F* (*C* (*sup* (λ*x*. {*y*. ¬ *P x y*}) (*U f*)))) *x*
    — partial_function requires a quantifier over f', so let's have a fake one
  and *elem*: *y* ∉ *U f x*
  shows *P x y*
using *elem*
proof(*rule contrapos-np*)
  have *mono'*: *monotone* (≥) (≥) (λ*f*. *U* (*F* (*C f*)))
    and *mono''*: *mono* (λ*f*. *U* (*F* (*C f*)))
    using *mono* **by**(*simp-all add*: *monotone-def fun-ord-def le-fun-def mono-def*)
  hence *eq'*: *U f* = *gfp* (λ*f*. *U* (*F* (*C f*)))
    by(*subst eq*)(*simp add*: *fun-lub-Inf fun-ord-ge gfp-eq-fixp inverse2*)

  let *?P* = λ*x*. {*y*. ¬ *P x y*}
  have *?P* ≤ *gfp* (λ*f*. *U* (*F* (*C f*)))
    using *mono''* **by**(*rule coinduct*)(*auto intro*!: *le-funI dest*: *step*[*OF refl*] *simp add*: *eq'*)
  moreover
  assume ¬ *P x y*
  ultimately show *y* ∈ *U f x* **by**(*auto simp add*: *le-fun-def eq'*)
qed

declaration ‹*Partial-Function.init coset* @{*term coset.fixp-fun*}
@{*term coset.mono-body*} @{*thm coset.fixp-rule-uc*} @{*thm coset.fixp-induct-uc*}
(*SOME* @{*thm fixp-coinduct-set*})›

abbreviation $mono\text{-}set' \equiv monotone\ (fun\text{-}ord\ (\supseteq))\ (\supseteq)$

lemma [*partial-function-mono*]:
shows *insert-mono'*: $mono\text{-}set'\ A \Longrightarrow mono\text{-}set'\ (\lambda f.\ insert\ x\ (A\ f))$
and *UNION-mono'*: $[\![mono\text{-}set'\ B;\ \bigwedge y.\ mono\text{-}set'\ (\lambda f.\ C\ y\ f)]\!] \Longrightarrow mono\text{-}set'$ $(\lambda f.\ \bigcup y{\in}B\ f.\ C\ y\ f)$
and *set-bind-mono'*: $[\![mono\text{-}set'\ B;\ \bigwedge y.\ mono\text{-}set'\ (\lambda f.\ C\ y\ f)]\!] \Longrightarrow mono\text{-}set'$ $(\lambda f.\ Set.bind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
and *Un-mono'*: $[\![\ mono\text{-}set'\ A;\ mono\text{-}set'\ B\ ]\!] \Longrightarrow mono\text{-}set'\ (\lambda f.\ A\ f \cup B\ f)$
and *Int-mono'*: $[\![\ mono\text{-}set'\ A;\ mono\text{-}set'\ B\ ]\!] \Longrightarrow mono\text{-}set'\ (\lambda f.\ A\ f \cap B\ f)$
unfolding *bind-UNION* **by**(*fast intro*!: *monotoneI dest*: *monotoneD*)+

context begin
private partial-function (*coset*) *test2* :: *nat* ⇒ *nat set*
where *test2 x* = *insert x* (*test2* (*Suc x*))

private lemma *test2-coinduct*:
assumes *P x y*
and ∗: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow y = x \lor (P\ (Suc\ x)\ y \lor y \in test2\ (Suc\ x))$
shows $y \in test2\ x$
using ‹*P x y*›
apply(*rule contrapos-pp*)
apply(*erule test2.raw-induct*[*rotated*])
apply(*simp add*: ∗)
done

end

end

2 Negligibility

theory *Negligible* **imports**
Complex-Main
Landau-Symbols.Landau-More
begin

named-theorems *negligible-intros*

definition *negligible* :: (*nat* ⇒ *real*) ⇒ *bool*
where $negligible\ f \longleftrightarrow (\forall c{>}0.\ f \in o(\lambda x.\ inverse\ (x\ powr\ c)))$

lemma *negligibleI* [*intro?*]:
$(\bigwedge c.\ c > 0 \Longrightarrow f \in o(\lambda x.\ inverse\ (x\ powr\ c))) \Longrightarrow negligible\ f$
unfolding *negligible-def* **by**(*simp*)

lemma *negligibleD*:
⟦ *negligible f*; *c* > *0* ⟧ ⟹ *f* ∈ *o*(*λx*. *inverse* (*x powr c*))
unfolding *negligible-def* **by**(*simp*)

lemma *negligibleD-real*:
assumes *negligible f*
shows *f* ∈ *o*(*λx*. *inverse* (*x powr c*))
proof −
let *?c* = *max 1 c*
have *f* ∈ *o*(*λx*. *inverse* (*x powr ?c*)) **using** *assms* **by**(*rule negligibleD*) *simp*
also have (*λx*. *x powr c*) ∈ *O*(*λx*. *real x powr max 1 c*)
by(*rule bigoI*[**where** *c=1*])(*auto simp add*: *eventually-at-top-linorder intro*!: *exI*[**where** *x=1*] *powr-mono*)
then have (*λx*. *inverse* (*real x powr max 1 c*)) ∈ *O*(*λx*. *inverse* (*x powr c*))
by(*auto simp add*: *eventually-at-top-linorder exI*[**where** *x=1*] *intro*: *landau-o.big.inverse*)
finally show *?thesis* **.**
qed

lemma *negligible-mono*: ⟦ *negligible g*; *f* ∈ *O*(*g*) ⟧ ⟹ *negligible f*
by(*rule negligibleI*)(*drule* (*1*) *negligibleD*; *erule* (*1*) *landau-o.big-small-trans*)

lemma *negligible-le*: ⟦ *negligible g*; ⋀*η*. |*f η*| ≤ *g η* ⟧ ⟹ *negligible f*
by(*erule negligible-mono*)(*force intro*: *order-trans intro*!: *eventually-sequentiallyI landau-o.big-mono*)

lemma *negligible-K0* [*negligible-intros*, *simp*, *intro*!]: *negligible* (*λ-*. *0*)
by(*rule negligibleI*) *simp*

lemma *negligible-0* [*negligible-intros*, *simp*, *intro*!]: *negligible 0*
by(*simp add*: *zero-fun-def*)

lemma *negligible-const-iff* [*simp*]: *negligible* (*λ-*. *c* :: *real*) ⟷ *c* = *0*
by(*auto simp add*: *negligible-def const-smallo-inverse-powr filterlim-real-sequentially dest*!: *spec*[**where** *x=1*])

lemma *not-negligible-1*: ¬ *negligible* (*λ-*. *1* :: *real*)
by *simp*

lemma *negligible-plus* [*negligible-intros*]:
⟦ *negligible f*; *negligible g* ⟧ ⟹ *negligible* (*λη*. *f η* + *g η*)
by(*auto intro*!: *negligibleI dest*!: *negligibleD intro*: *sum-in-smallo*)

lemma *negligible-uminus* [*simp*]: *negligible* (*λη*. − *f η*) ⟷ *negligible f*
by(*simp add*: *negligible-def*)

lemma *negligible-uminusI* [*negligible-intros*]: *negligible f* ⟹ *negligible* (*λη*. − *f η*)
by *simp*

lemma *negligible-minus* [*negligible-intros*]:
  ⟦ *negligible f*; *negligible g* ⟧ ⟹ *negligible* (λη. *f* η − *g* η)
by(*auto simp add*: *uminus-add-conv-diff*[*symmetric*] *negligible-plus simp del*: *uminus-add-conv-diff*)

lemma *negligible-cmult*: *negligible* (λη. *c* ∗ *f* η) ⟷ *negligible f* ∨ *c* = *0*
by(*auto intro*!: *negligibleI dest*!: *negligibleD*)

lemma *negligible-cmultI* [*negligible-intros*]:
  (*c* ≠ *0* ⟹ *negligible f*) ⟹ *negligible* (λη. *c* ∗ *f* η)
by(*auto simp add*: *negligible-cmult*)

lemma *negligible-multc*: *negligible* (λη. *f* η ∗ *c*) ⟷ *negligible f* ∨ *c* = *0*
by(*subst mult.commute*)(*simp add*: *negligible-cmult*)

lemma *negligible-multcI* [*negligible-intros*]:
  (*c* ≠ *0* ⟹ *negligible f*) ⟹ *negligible* (λη. *f* η ∗ *c*)
by(*auto simp add*: *negligible-multc*)

lemma *negligible-times* [*negligible-intros*]:
  assumes *f*: *negligible f*
  and *g*: *negligible g*
  shows *negligible* (λη. *f* η ∗ *g* η :: *real*)
proof
  fix *c* :: *real*
  assume *0* < *c*
  hence *0* < *c* / *2* **by** *simp*
  from *negligibleD*[*OF f this*] *negligibleD*[*OF g this*]
  have (λη. *f* η ∗ *g* η) ∈ *o*(λ*x*. *inverse* (*x powr* (*c* / *2*)) ∗ *inverse* (*x powr* (*c* / *2*)))
    by(*rule landau-o.small-mult*)
  also have ... = *o*(λ*x*. *inverse* (*x powr c*))
    by(*rule landau-o.small.cong*)(*auto simp add*: *inverse-mult-distrib*[*symmetric*] *powr-add*[*symmetric*] *eventually-at-top-linorder intro*!: *exI*[**where** *x*=*1*] *simp del*: *inverse-mult-distrib*)
  finally show (λη. *f* η ∗ *g* η) ∈ ... **.**
qed

lemma *negligible-power* [*negligible-intros*]:
  assumes *negligible f*
  and *n* > *0*
  shows *negligible* (λη. *f* η ^ *n* :: *real*)
using ‹*n* > *0*›
proof(*induct n*)
  case (*Suc n*)
  thus *?case* **using** ‹*negligible f*› **by**(*cases n*)(*simp-all add*: *negligible-times*)
qed *simp*

lemma *negligible-powr* [*negligible-intros*]:
  assumes *f*: *negligible f*

and *p*: *p* > *0*
shows *negligible* (λ*x*. |*f x*| *powr p* :: *real*)
proof
fix *c* :: *real*
let *?c* = *c* / *p*
assume *c*: *0* < *c*
with *p* **have** *0* < *?c* **by** *simp*
with *f* **have** *f* ∈ *o*(λ*x*. *inverse* (*x powr ?c*)) **by**(*rule negligibleD*)
hence (λ*x*. |*f x*| *powr p*) ∈ *o*(λ*x*. |*inverse* (*x powr ?c*)| *powr p*) **using** *p* **by**(*rule smallo-powr*)
also have ... = *o*(λ*x*. *inverse* (*x powr c*))
apply(*rule landau-o.small.cong*) **using** *p* **by**(*auto simp add*: *powr-powr*)
finally show (λ*x*. |*f x*| *powr p*) ∈ ... **.**
qed

lemma *negligible-abs* [*simp*]: *negligible* (λ*x*. |*f x*|) ⟷ *negligible f*
by(*simp add*: *negligible-def*)

lemma *negligible-absI* [*negligible-intros*]: *negligible f* ⟹ *negligible* (λ*x*. |*f x*|)
by(*simp*)

lemma *negligible-powrI* [*negligible-intros*]:
assumes *0* ≤ *k k* < *1*
shows *negligible* (λ*x*. *k powr x*)
proof(*cases k* = *0*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof
fix *c* :: *real*
assume *0* < *c*
then have (λ*x*. *real x powr c*) ∈ *o*(λ*x*. *inverse k powr real x*) **using** *assms False* **by**(*intro powr-fast-growth-tendsto*)(*simp-all add*: *one-less-inverse-iff filterlim-real-sequentially*)
then have (λ*x*. *inverse* (*k powr* − *real x*)) ∈ *o*(λ*x*. *inverse* (*real x powr c*)) **using** *assms*
by(*intro landau-o.small.inverse*)(*auto simp add*: *False eventually-sequentially powr-minus intro*: *exI*[**where** *x*=*1*])
also have (λ*x*. *inverse* (*k powr* − *real x*)) = (λ*x*. *k powr real x*) **by**(*simp add*: *powr-minus*)
finally show ... ∈ *o*(λ*x*. *inverse* (*x powr c*)) **.**
qed
qed

lemma *negligible-powerI* [*negligible-intros*]:
fixes *k* :: *real*
assumes |*k*| < *1*

shows *negligible* ($\lambda n.\ k\ \widehat{}\ n$)
proof(*cases k = 0*)
case *True*
show *?thesis* **using** *negligible-K0*
by(*rule negligible-mono*)(*auto intro*: *exI*[**where** *x=1*] *simp add*: *True eventually-at-top-linorder*)
next
case *False*
hence $0 < |k|$ **by** *auto*
from *assms* **have** *negligible* ($\lambda x.\ |k|$ *powr real x*) **using** *negligible-powrI*[*of* $|k|$] **by** *simp*
hence *negligible* ($\lambda x.\ |k|\ \widehat{}\ x$) **using** *False*
by(*elim negligible-mono*)(*simp add*: *powr-realpow*)
then show *?thesis* **by**(*simp add*: *power-abs*[*symmetric*])
qed

lemma *negligible-inverse-powerI* [*negligible-intros*]: $|k| > 1 \Longrightarrow$ *negligible* ($\lambda\eta.\ 1\ /\ k\ \widehat{}\ \eta$)
using *negligible-powerI*[*of* $1\ /\ k$] **by**(*simp add*: *power-one-over*)

inductive *polynomial* :: (*nat* ⇒ *real*) ⇒ *bool*
for *f*
where $f \in O(\lambda x.\ x$ *powr n*) $\Longrightarrow$ *polynomial f*

lemma *negligible-times-poly*:
assumes *f*: *negligible f*
and *g*: $g \in O(\lambda x.\ x$ *powr n*)
shows *negligible* ($\lambda x.\ f\ x * g\ x$)
proof
fix *c* :: *real*
assume *c*: $0 < c$
from *negligibleD-real*[*OF f*] *g*
have ($\lambda x.\ f\ x * g\ x$) $\in o(\lambda x.$ *inverse* (*x powr* ($c + n$)) $*$ *x powr n*)
by(*rule landau-o.small-big-mult*)
also have ... $= o(\lambda x.$ *inverse* (*x powr c*))
by(*rule landau-o.small.cong*)(*auto simp add*: *powr-minus*[*symmetric*] *powr-add*[*symmetric*] *intro*!: *exI*[**where** *x=0*])
finally show ($\lambda x.\ f\ x * g\ x$) $\in o(\lambda x.$ *inverse* (*x powr c*)) **.**
qed

lemma *negligible-poly-times*:
⟦ $f \in O(\lambda x.\ x$ *powr n*); *negligible g* ⟧ $\Longrightarrow$ *negligible* ($\lambda x.\ f\ x * g\ x$)
by(*subst mult.commute*)(*rule negligible-times-poly*)

lemma *negligible-times-polynomial* [*negligible-intros*]:
⟦ *negligible f*; *polynomial g* ⟧ $\Longrightarrow$ *negligible* ($\lambda x.\ f\ x * g\ x$)
by(*clarsimp simp add*: *polynomial.simps negligible-times-poly*)

lemma *negligible-polynomial-times* [*negligible-intros*]:

⟦ *polynomial f*; *negligible g* ⟧ ⟹ *negligible* (λ*x*. *f x* ∗ *g x*)
by(*clarsimp simp add*: *polynomial.simps negligible-poly-times*)

lemma *negligible-divide-poly1*:
⟦ *f* ∈ *O*(λ*x*. *x powr n*); *negligible* (λη. *1* / *g* η) ⟧ ⟹ *negligible* (λη. *real* (*f* η) / *g* η)
by(*drule* (*1*) *negligible-times-poly*) *simp*

lemma *negligible-divide-polynomial1* [*negligible-intros*]:
⟦ *polynomial f*; *negligible* (λη. *1* / *g* η) ⟧ ⟹ *negligible* (λη. *real* (*f* η) / *g* η)
by(*clarsimp simp add*: *polynomial.simps negligible-divide-poly1*)

end

3 The resumption-error monad

theory *Resumption*
imports
Misc-CryptHOL
Partial-Function-Set
begin

codatatype (*results*: ′*a*, *outputs*: ′*out*, ′*in*) *resumption*
= *Done* (*result*: ′*a option*)
| *Pause* (*output*: ′*out*) (*resume*: ′*in* ⇒ (′*a*, ′*out*, ′*in*) *resumption*)
where
resume (*Done a*) = (λ*inp*. *Done None*)

code-datatype *Done Pause*

primcorec *bind-resumption* ::
(′*a*, ′*out*, ′*in*) *resumption*
⇒ (′*a* ⇒ (′*b*, ′*out*, ′*in*) *resumption*) ⇒ (′*b*, ′*out*, ′*in*) *resumption*
where
⟦ *is-Done x*; *result x* ≠ *None* ⟶ *is-Done* (*f* (*the* (*result x*))) ⟧ ⟹ *is-Done* (*bind-resumption x f*)
| *result* (*bind-resumption x f*) = *result x* ⤜ *result* ∘ *f*
| *output* (*bind-resumption x f*) = (*if is-Done x then output* (*f* (*the* (*result x*))) *else output x*)
| *resume* (*bind-resumption x f*) = (λ*inp*. *if is-Done x then resume* (*f* (*the* (*result x*))) *inp else bind-resumption* (*resume x inp*) *f*)

declare *bind-resumption.sel* [*simp del*]

adhoc-overloading *Monad-Syntax.bind* ⇌ *bind-resumption*

lemma *is-Done-bind-resumption* [*simp*]:
is-Done (*x* ⤜ *f*) ⟷ *is-Done x* ∧ (*result x* ≠ *None* ⟶ *is-Done* (*f* (*the* (*result x*))))

by(*simp add*: *bind-resumption-def*)

lemma *result-bind-resumption* [*simp*]:
  $is\text{-}Done\ (x \gg= f) \Longrightarrow result\ (x \gg= f) = result\ x \gg= result \circ f$
by(*simp add*: *bind-resumption-def*)

lemma *output-bind-resumption* [*simp*]:
  $\neg\ is\text{-}Done\ (x \gg= f) \Longrightarrow output\ (x \gg= f) = (if\ is\text{-}Done\ x\ then\ output\ (f\ (the\ (result\ x)))\ else\ output\ x)$
by(*simp add*: *bind-resumption-def*)

lemma *resume-bind-resumption* [*simp*]:
  $\neg\ is\text{-}Done\ (x \gg= f) \Longrightarrow$
  $resume\ (x \gg= f) =$
  $(if\ is\text{-}Done\ x\ then\ resume\ (f\ (the\ (result\ x)))$
   $else\ (\lambda inp.\ resume\ x\ inp \gg= f))$
by(*auto simp add*: *bind-resumption-def*)

definition $DONE :: 'a \Rightarrow ('a, 'out, 'in)\ resumption$
where $DONE = Done \circ Some$

definition $ABORT :: ('a, 'out, 'in)\ resumption$
where $ABORT = Done\ None$

lemma [*simp*]:
  shows *is-Done-DONE*: $is\text{-}Done\ (DONE\ a)$
  and *is-Done-ABORT*: $is\text{-}Done\ ABORT$
  and *result-DONE*: $result\ (DONE\ a) = Some\ a$
  and *result-ABORT*: $result\ ABORT = None$
  and *DONE-inject*: $DONE\ a = DONE\ b \longleftrightarrow a = b$
  and *DONE-neq-ABORT*: $DONE\ a \neq ABORT$
  and *ABORT-neq-DONE*: $ABORT \neq DONE\ a$
  and *ABORT-eq-Done*: $\bigwedge a.\ ABORT = Done\ a \longleftrightarrow a = None$
  and *Done-eq-ABORT*: $\bigwedge a.\ Done\ a = ABORT \longleftrightarrow a = None$
  and *DONE-eq-Done*: $\bigwedge b.\ DONE\ a = Done\ b \longleftrightarrow b = Some\ a$
  and *Done-eq-DONE*: $\bigwedge b.\ Done\ b = DONE\ a \longleftrightarrow b = Some\ a$
  and *DONE-neq-Pause*: $DONE\ a \neq Pause\ out\ c$
  and *Pause-neq-DONE*: $Pause\ out\ c \neq DONE\ a$
  and *ABORT-neq-Pause*: $ABORT \neq Pause\ out\ c$
  and *Pause-neq-ABORT*: $Pause\ out\ c \neq ABORT$
by(*auto simp add*: *DONE-def ABORT-def*)

lemma *resume-ABORT* [*simp*]:
  $resume\ (Done\ r) = (\lambda inp.\ ABORT)$
by(*simp add*: *ABORT-def*)

declare *resumption.sel*(*3*)[*simp del*]

lemma *results-DONE* [*simp*]: $results\ (DONE\ x) = \{x\}$

by(*simp add*: *DONE-def*)

lemma *results-ABORT* [*simp*]: *results ABORT* = {}
by(*simp add*: *ABORT-def*)

lemma *outputs-ABORT* [*simp*]: *outputs ABORT* = {}
by(*simp add*: *ABORT-def*)

lemma *outputs-DONE* [*simp*]: *outputs* (*DONE x*) = {}
by(*simp add*: *DONE-def*)

lemma *is-Done-cases* [*cases pred*]:
  assumes *is-Done r*
  obtains (*DONE*) *x* **where** *r* = *DONE x* | (*ABORT*) *r* = *ABORT*
using *assms* **by**(*cases r*) *auto*

lemma *not-is-Done-conv-Pause*: ¬ *is-Done r* ⟷ (∃ *out c*. *r* = *Pause out c*)
by(*cases r*) *auto*

lemma *Done-bind* [*code*]:
  Done a ≫= *f* = (*case a of None* ⇒ *Done None* | *Some a* ⇒ *f a*)
by(*rule resumption.expand*)(*auto split*: *option.split*)

lemma *DONE-bind* [*simp*]:
  DONE a ≫= *f* = *f a*
by(*simp add*: *DONE-def Done-bind*)

lemma *bind-resumption-Pause* [*simp*, *code*]: **fixes** *cont* **shows**
  Pause out cont ≫= *f*
  = *Pause out* (*λinp*. *cont inp* ≫= *f*)
by(*rule resumption.expand*)(*simp-all*)

lemma *bind-DONE* [*simp*]:
  x ≫= *DONE* = *x*
by(*coinduction arbitrary*: *x*)(*auto simp add*: *split-beta o-def*)

lemma *bind-bind-resumption*:
  fixes *r* :: (*'a*, *'in*, *'out*) *resumption*
  shows (*r* ≫= *f*) ≫= *g* = *do* { *x* ← *r*; *f x* ≫= *g* }
apply(*coinduction arbitrary*: *r rule*: *resumption.coinduct-strong*)
apply(*auto simp add*: *split-beta bind-eq-Some-conv*)
apply(*case-tac* [!] *result r*)
apply *simp-all*
done

lemmas *resumption-monad* = *DONE-bind bind-DONE bind-bind-resumption*

lemma *ABORT-bind* [*simp*]: *ABORT* ≫= *f* = *ABORT*
by(*simp add*: *ABORT-def Done-bind*)

lemma *bind-resumption-is-Done*: *is-Done f* $\Longrightarrow$ *f* $\gg=$ *g* = (*if result f* = *None then ABORT else g* (*the* (*result f*)))
by(*rule resumption.expand*) *auto*

lemma *bind-resumption-eq-Done-iff* [*simp*]:
f $\gg=$ *g* = *Done x* $\longleftrightarrow$ ($\exists$ *y*. *f* = *DONE y* $\wedge$ *g y* = *Done x*) $\vee$ *f* = *ABORT* $\wedge$ *x* = *None*
by(*cases f*)(*auto simp add*: *Done-bind split*: *option.split*)

lemma *bind-resumption-cong*:
assumes *x* = *y*
and $\bigwedge$*z*. *z* $\in$ *results y* $\Longrightarrow$ *f z* = *g z*
shows *x* $\gg=$ *f* = *y* $\gg=$ *g*
using *assms*(*2*) **unfolding** ‹*x* = *y*›
proof(*coinduction arbitrary*: *y rule*: *resumption.coinduct-strong*)
case *Eq-resumption* **thus** *?case*
by(*auto intro*: *resumption.set-sel simp add*: *is-Done-def rel-fun-def*)
(*fastforce del*: *exI intro*!: *exI intro*: *resumption.set-sel*(*2*) *simp add*: *is-Done-def*)
qed

lemma *results-bind-resumption*:
results (*bind-resumption x f*) = ($\bigcup$ *a*$\in$*results x*. *results* (*f a*))
(**is** *?lhs* = *?rhs*)
proof(*intro set-eqI iffI*)
show *z* $\in$ *?rhs* **if** *z* $\in$ *?lhs* **for** *z* **using** *that*
proof(*induction r*$\equiv$*x* $\gg=$ *f arbitrary*: *x*)
case (*Done z z' x*)
from *Done*(*1*) *Done*(*2*)[*symmetric*] **show** *?case* **by**(*auto*)
next
case (*Pause out c r z x*)
then show *?case*
proof(*cases x*)
case (*Done x'*)
show *?thesis*
proof(*cases x'*)
case *None*
with *Done Pause*(*4*) **show** *?thesis* **by**(*auto simp add*: *ABORT-def*[*symmetric*])
next
case (*Some x''*)
thus *?thesis* **using** *Pause*(*1*,*2*,*4*) *Done*
by(*auto 4 3 simp add*: *DONE-def*[*unfolded o-def*, *symmetric*, *unfolded fun-eq-iff*] *dest*: *sym*)
qed
qed(*fastforce*)
qed
next
fix *z*
assume *z* $\in$ *?rhs*

then obtain *z′* **where** *z′*: *z′* ∈ *results x*
and *z*: *z* ∈ *results* (*f z′*) **by** *blast*
from *z′* **show** *z* ∈ *?lhs*
proof(*induction z′≡z′ x*)
case (*Done r*)
then show *?case* **using** *z*
by(*auto simp add*: *DONE-def*[*unfolded o-def*, *symmetric*, *unfolded fun-eq-iff*])
qed *auto*
qed

lemma *outputs-bind-resumption* [*simp*]:
outputs (*bind-resumption r f*) = *outputs r* ∪ (⋃*x*∈*results r*. *outputs* (*f x*))
(**is** *?lhs* = *?rhs*)
proof(*rule set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
proof(*induction r′≡bind-resumption r f arbitrary*: *r*)
case (*Pause1 out c*)
thus *?case* **by**(*cases r*)(*auto simp add*: *Done-bind split*: *option.split-asm dest*: *sym*)
next
case (*Pause2 out c r′ x*)
thus *?case* **by**(*cases r*)(*auto 4 3 simp add*: *Done-bind split*: *option.split-asm dest*: *sym*)
qed
next
fix *x*
assume *x* ∈ *?rhs*
then consider (*left*) *x* ∈ *outputs r* | (*right*) *a* **where** *a* ∈ *results r x* ∈ *outputs* (*f a*) **by** *auto*
then show *x* ∈ *?lhs*
proof *cases*
{ case *left* **thus** *?thesis* **by** *induction auto* **}**
{ case *right* **thus** *?thesis* **by** *induction*(*auto simp add*: *Done-bind*) **}**
qed
qed

primrec *ensure* :: *bool* ⇒ (*unit*, *′out*, *′in*) *resumption*
where
ensure True = *DONE* ()
| *ensure False* = *ABORT*

lemma *is-Done-map-resumption* [*simp*]:
is-Done (*map-resumption f1 f2 r*) ⟷ *is-Done r*
by(*cases r*) *simp-all*

lemma *result-map-resumption* [*simp*]:
is-Done r ⟹ *result* (*map-resumption f1 f2 r*) = *map-option f1* (*result r*)
by(*clarsimp simp add*: *is-Done-def*)

lemma *output-map-resumption* [*simp*]:
¬ *is-Done r* ⟹ *output* (*map-resumption f1 f2 r*) = *f2* (*output r*)
by(*cases r*) *simp-all*

lemma *resume-map-resumption* [*simp*]:
¬ *is-Done r*
⟹ *resume* (*map-resumption f1 f2 r*) = *map-resumption f1 f2* ∘ *resume r*
by(*cases r*) *simp-all*

lemma *rel-resumption-is-DoneD*: *rel-resumption A B r1 r2* ⟹ *is-Done r1* ⟷ *is-Done r2*
by(*cases r1 r2 rule*: *resumption.exhaust*[*case-product resumption.exhaust*]) *simp-all*

lemma *rel-resumption-resultD1*:
⟦ *rel-resumption A B r1 r2*; *is-Done r1* ⟧ ⟹ *rel-option A* (*result r1*) (*result r2*)
by(*cases r1 r2 rule*: *resumption.exhaust*[*case-product resumption.exhaust*]) *simp-all*

lemma *rel-resumption-resultD2*:
⟦ *rel-resumption A B r1 r2*; *is-Done r2* ⟧ ⟹ *rel-option A* (*result r1*) (*result r2*)
by(*cases r1 r2 rule*: *resumption.exhaust*[*case-product resumption.exhaust*]) *simp-all*

lemma *rel-resumption-outputD1*:
⟦ *rel-resumption A B r1 r2*; ¬ *is-Done r1* ⟧ ⟹ *B* (*output r1*) (*output r2*)
by(*cases r1 r2 rule*: *resumption.exhaust*[*case-product resumption.exhaust*]) *simp-all*

lemma *rel-resumption-outputD2*:
⟦ *rel-resumption A B r1 r2*; ¬ *is-Done r2* ⟧ ⟹ *B* (*output r1*) (*output r2*)
by(*cases r1 r2 rule*: *resumption.exhaust*[*case-product resumption.exhaust*]) *simp-all*

lemma *rel-resumption-resumeD1*:
⟦ *rel-resumption A B r1 r2*; ¬ *is-Done r1* ⟧
⟹ *rel-resumption A B* (*resume r1 inp*) (*resume r2 inp*)
by(*cases r1 r2 rule*: *resumption.exhaust*[*case-product resumption.exhaust*])(*auto dest*: *rel-funD*)

lemma *rel-resumption-resumeD2*:
⟦ *rel-resumption A B r1 r2*; ¬ *is-Done r2* ⟧
⟹ *rel-resumption A B* (*resume r1 inp*) (*resume r2 inp*)
by(*cases r1 r2 rule*: *resumption.exhaust*[*case-product resumption.exhaust*])(*auto dest*: *rel-funD*)

lemma *rel-resumption-coinduct*
[*consumes 1*, *case-names Done Pause*,
case-conclusion Done is-Done result,
case-conclusion Pause output resume,
coinduct pred: *rel-resumption*]:
assumes *X*: *X r1 r2*
and *Done*: ⋀*r1 r2*. *X r1 r2* ⟹ (*is-Done r1* ⟷ *is-Done r2*) ∧ (*is-Done r1* ⟶ *is-Done r2* ⟶ *rel-option A* (*result r1*) (*result r2*))

and *Pause*: $\bigwedge$*r1 r2*. ⟦ *X r1 r2*; ¬ *is-Done r1*; ¬ *is-Done r2* ⟧ ⟹ *B* (*output r1*) (*output r2*) ∧ (∀ *inp*. *X* (*resume r1 inp*) (*resume r2 inp*))
shows *rel-resumption A B r1 r2*
using *X*
apply(*rule resumption.rel-coinduct*)
apply(*unfold rel-fun-def*)
apply(*rule conjI*)
apply(*erule Done*[*THEN conjunct1*])
apply(*rule conjI*)
apply(*erule Done*[*THEN conjunct2*])
apply(*rule impI*)+
apply(*drule* (*2*) *Pause*)
apply *blast*
done

3.1 Setup for *partial-function*

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

coinductive *resumption-ord* :: (*'a*, *'out*, *'in*) *resumption* ⇒ (*'a*, *'out*, *'in*) *resumption* ⇒ *bool*
where
Done-Done: *flat-ord None a a'* ⟹ *resumption-ord* (*Done a*) (*Done a'*)
| *Done-Pause*: *resumption-ord ABORT* (*Pause out c*)
| *Pause-Pause*: ((=) ===> *resumption-ord*) *c c'* ⟹ *resumption-ord* (*Pause out c*) (*Pause out c'*)

inductive-simps *resumption-ord-simps* [*simp*]:
resumption-ord (*Pause out c*) *r*
resumption-ord r (*Done a*)

lemma *resumption-ord-is-DoneD*:
⟦ *resumption-ord r r'*; *is-Done r'* ⟧ ⟹ *is-Done r*
by(*cases r'*)(*auto simp add*: *fun-ord-def*)

lemma *resumption-ord-resultD*:
⟦ *resumption-ord r r'*; *is-Done r'* ⟧ ⟹ *flat-ord None* (*result r*) (*result r'*)
by(*cases r'*)(*auto simp add*: *flat-ord-def*)

lemma *resumption-ord-outputD*:
⟦ *resumption-ord r r'*; ¬ *is-Done r* ⟧ ⟹ *output r* = *output r'*
by(*cases r*) *auto*

lemma *resumption-ord-resumeD*:
⟦ *resumption-ord r r'*; ¬ *is-Done r* ⟧ ⟹ ((=) ===> *resumption-ord*) (*resume r*) (*resume r'*)
by(*cases r*) *auto*

lemma *resumption-ord-abort*:

⟦ *resumption-ord r r'*; *is-Done r*; ¬ *is-Done r'* ⟧ ⟹ *result r* = *None*
by(*auto elim*: *resumption-ord.cases*)

lemma *resumption-ord-coinduct* [*consumes 1*, *case-names Done Abort Pause*, *case-conclusion Pause output resume*, *coinduct pred*: *resumption-ord*]:
assumes *X r r'*
and *Done*: ⋀*r r'*. ⟦ *X r r'*; *is-Done r'* ⟧ ⟹ *is-Done r* ∧ *flat-ord None* (*result r*) (*result r'*)
and *Abort*: ⋀*r r'*. ⟦ *X r r'*; ¬ *is-Done r'*; *is-Done r* ⟧ ⟹ *result r* = *None*
and *Pause*: ⋀*r r'*. ⟦ *X r r'*; ¬ *is-Done r*; ¬ *is-Done r'* ⟧
⟹ *output r* = *output r'* ∧ ((=) ===> (λ*r r'*. *X r r'* ∨ *resumption-ord r r'*)) (*resume r*) (*resume r'*)
shows *resumption-ord r r'*
using ‹*X r r'*›
proof *coinduct*
case (*resumption-ord r r'*)
thus *?case*
by(*cases r r' rule*: *resumption.exhaust*[*case-product resumption.exhaust*])(*auto dest*: *Done Pause Abort*)
qed

end

lemma *resumption-ord-ABORT* [*intro*!, *simp*]: *resumption-ord ABORT r*
by(*cases r*)(*simp-all add*: *flat-ord-def resumption-ord.Done-Pause*)

lemma *resumption-ord-ABORT2* [*simp*]: *resumption-ord r ABORT* ⟷ *r* = *ABORT*
by(*simp add*: *ABORT-def flat-ord-def*)

lemma *resumption-ord-DONE1* [*simp*]: *resumption-ord* (*DONE x*) *r* ⟷ *r* = *DONE x*
by(*cases r*)(*auto simp add*: *option-ord-Some1-iff DONE-def dest*: *resumption-ord-abort*)

lemma *resumption-ord-refl*: *resumption-ord r r*
by(*coinduction arbitrary*: *r*)(*auto simp add*: *flat-ord-def*)

lemma *resumption-ord-antisym*:
⟦ *resumption-ord r r'*; *resumption-ord r' r* ⟧
⟹ *r* = *r'*
proof(*coinduction arbitrary*: *r r' rule*: *resumption.coinduct-strong*)
case (*Eq-resumption r r'*)
thus *?case*
by *cases*(*auto simp add*: *flat-ord-def rel-fun-def*)
qed

lemma *resumption-ord-trans*:
⟦ *resumption-ord r r'*; *resumption-ord r' r''* ⟧
⟹ *resumption-ord r r''*
proof(*coinduction arbitrary*: *r r' r''*)

case (*Done r r′ r′′*)
thus *?case* **by**(*auto 4 4 elim*: *resumption-ord.cases simp add*: *flat-ord-def*)
next
case (*Abort r r′ r′′*)
thus *?case* **by**(*auto 4 4 elim*: *resumption-ord.cases simp add*: *flat-ord-def*)
next
case (*Pause r r′ r′′*)
hence *resumption-ord r r′ resumption-ord r′ r′′* **by** *simp-all*
thus *?case* **using** ‹¬ *is-Done r*› ‹¬ *is-Done r′′*›
by(*cases*)(*auto simp add*: *rel-fun-def*)
qed

primcorec *resumption-lub* :: (′*a*, ′*out*, ′*in*) *resumption set* ⇒ (′*a*, ′*out*, ′*in*) *resumption*
where
∀ *r* ∈ *R*. *is-Done r* ⟹ *is-Done* (*resumption-lub R*)
| *result* (*resumption-lub R*) = *flat-lub None* (*result* ‘ *R*)
| *output* (*resumption-lub R*) = (*THE out*. *out* ∈ *output* ‘ (*R* ∩ {*r*. ¬ *is-Done r*}))
| *resume* (*resumption-lub R*) = (*λinp*. *resumption-lub* ((*λc*. *c inp*) ‘ *resume* ‘ (*R* ∩ {*r*. ¬ *is-Done r*})))

lemma *is-Done-resumption-lub* [*simp*]:
is-Done (*resumption-lub R*) ⟷ (∀ *r* ∈ *R*. *is-Done r*)
by(*simp add*: *resumption-lub-def*)

lemma *result-resumption-lub* [*simp*]:
∀ *r* ∈ *R*. *is-Done r* ⟹ *result* (*resumption-lub R*) = *flat-lub None* (*result* ‘ *R*)
by(*simp add*: *resumption-lub-def*)

lemma *output-resumption-lub* [*simp*]:
∃ *r*∈*R*. ¬ *is-Done r* ⟹ *output* (*resumption-lub R*) = (*THE out*. *out* ∈ *output* ‘ (*R* ∩ {*r*. ¬ *is-Done r*}))
by(*simp add*: *resumption-lub-def*)

lemma *resume-resumption-lub* [*simp*]:
∃ *r*∈*R*. ¬ *is-Done r*
⟹ *resume* (*resumption-lub R*) *inp* =
resumption-lub ((*λc*. *c inp*) ‘ *resume* ‘ (*R* ∩ {*r*. ¬ *is-Done r*}))
by(*simp add*: *resumption-lub-def*)

lemma *resumption-lub-empty*: *resumption-lub* {} = *ABORT*
by(*subst resumption-lub.code*)(*simp add*: *flat-lub-def*)

context
fixes *R state inp R′*
defines *R′-def*: *R′* ≡ (*λc*. *c inp*) ‘ *resume* ‘ (*R* ∩ {*r*. ¬ *is-Done r*})
assumes *chain*: *Complete-Partial-Order.chain resumption-ord R*
begin

lemma *resumption-ord-chain-resume*: *Complete-Partial-Order.chain resumption-ord R′*
proof(*rule chainI*)
  fix *r′ r′′*
  assume *r′* ∈ *R′*
    and *r′′* ∈ *R′*
  then obtain r′ r′′
    where *r′*: *r′* = *resume* r′ *inp* r′ ∈ *R* ¬ *is-Done* r′
    and *r′′*: *r′′* = *resume* r′′ *inp* r′′ ∈ *R* ¬ *is-Done* r′′
    by(*auto simp add*: *R′-def*)
  from *chain* ‹r′ ∈ *R*› ‹r′′ ∈ *R*›
  have *resumption-ord* r′ r′′ ∨ *resumption-ord* r′′ r′
    by(*auto elim*: *chainE*)
  with *r′ r′′*
  have *resumption-ord* (*resume* r′ *inp*) (*resume* r′′ *inp*) ∨
        resumption-ord (*resume* r′′ *inp*) (*resume* r′ *inp*)
    by(*auto elim*: *resumption-ord.cases simp add*: *rel-fun-def*)
  with *r′ r′′*
  show *resumption-ord r′ r′′* ∨ *resumption-ord r′′ r′* **by** *auto*
qed

end

lemma *resumption-partial-function-definition*:
  partial-function-definitions resumption-ord resumption-lub
proof
  show *resumption-ord r r* **for** *r* :: (*′a*, *′b*, *′c*) *resumption* **by**(*rule resumption-ord-refl*)
  show *resumption-ord r r′′* **if** *resumption-ord r r′ resumption-ord r′ r′′*
    for *r r′ r′′* :: (*′a*, *′b*, *′c*) *resumption* **using** *that* **by**(*rule resumption-ord-trans*)
  show *r* = *r′* **if** *resumption-ord r r′ resumption-ord r′ r* **for** *r r′* :: (*′a*, *′b*, *′c*) *resumption*
    using *that* **by**(*rule resumption-ord-antisym*)
next
  fix *R* **and** *r* :: (*′a*, *′b*, *′c*) *resumption*
  assume *Complete-Partial-Order.chain resumption-ord R r* ∈ *R*
  thus *resumption-ord r* (*resumption-lub R*)
  proof(*coinduction arbitrary*: *r R*)
    case (*Done r R*)
    note *chain* = ‹*Complete-Partial-Order.chain resumption-ord R*›
      and *r* = ‹*r* ∈ *R*›
    from ‹*is-Done* (*resumption-lub R*)› **have** *A*: ∀ *r* ∈ *R*. *is-Done r* **by** *simp*
    with *r* **obtain** *a′* **where** *r* = *Done a′* **by**(*cases r*) *auto*
    { fix *r′*
      assume *a′* ≠ *None*
      hence (*THE x*. *x* ∈ *result* ‘ *R* ∧ *x* ≠ *None*) = *a′*
        using *r A* ‹*r* = *Done a′*›
        by(*auto 4 3 del*: *the-equality intro*!: *the-equality intro*: *rev-image-eqI elim*: *chainE*[*OF chain*] *simp add*: *flat-ord-def is-Done-def*)
    }

with *A r* ‹*r* = *Done a*′› **show** *?case*
by(*cases a*′)(*auto simp add*: *flat-ord-def flat-lub-def*)
next
case (*Abort r R*)
hence *chain*: *Complete-Partial-Order.chain resumption-ord R* **and** *r* ∈ *R* **by** *simp-all*
from ‹*r* ∈ *R*› ‹¬ *is-Done* (*resumption-lub R*)› ‹*is-Done r*›
show *?case* **by**(*auto elim*: *chainE*[*OF chain*] *dest*: *resumption-ord-abort resumption-ord-is-DoneD*)
next
case (*Pause r R*)
hence *chain*: *Complete-Partial-Order.chain resumption-ord R*
and *r*: *r* ∈ *R* **by** *simp-all*
have *?resume*
using *r* ‹¬ *is-Done r*› *resumption-ord-chain-resume*[*OF chain*]
by(*auto simp add*: *rel-fun-def bexI*)
moreover
from *r* ‹¬ *is-Done r*› **have** *output* (*resumption-lub R*) = *output r*
by(*auto 4 4 simp add*: *bexI del*: *the-equality intro*!: *the-equality elim*: *chainE*[*OF chain*] *dest*: *resumption-ord-outputD*)
ultimately show *?case* **by** *simp*
qed
next
fix *R* **and** *r* :: (′*a*, ′*b*, ′*c*) *resumption*
assume *Complete-Partial-Order.chain resumption-ord R* ⋀*r*′. *r*′ ∈ *R* ⟹ *resumption-ord r*′ *r*
thus *resumption-ord* (*resumption-lub R*) *r*
proof(*coinduction arbitrary*: *R r*)
case (*Done R r*)
hence *chain*: *Complete-Partial-Order.chain resumption-ord R*
and *ub*: ∀ *r*′∈*R*. *resumption-ord r*′ *r* **by** *simp-all*
from ‹*is-Done r*› *ub* **have** *is-Done*: ∀ *r*′ ∈ *R*. *is-Done r*′
and *ub*′: ⋀*r*′. *r*′ ∈ *result* ‘ *R* ⟹ *flat-ord None r*′ (*result r*)
by(*auto dest*: *resumption-ord-is-DoneD resumption-ord-resultD*)
from *is-Done* **have** *chain*′: *Complete-Partial-Order.chain* (*flat-ord None*) (*result* ‘ *R*)
by(*auto 5 2 intro*!: *chainI elim*: *chainE*[*OF chain*] *dest*: *resumption-ord-resultD*)
hence *flat-ord None* (*flat-lub None* (*result* ‘ *R*)) (*result r*)
by(*rule partial-function-definitions.lub-least*[*OF flat-interpretation*])(*rule ub*′)
thus *?case* **using** *is-Done* **by** *simp*
next
case (*Abort R r*)
hence *chain*: *Complete-Partial-Order.chain resumption-ord R*
and *ub*: ∀ *r*′∈*R*. *resumption-ord r*′ *r* **by** *simp-all*
from ‹¬ *is-Done r*› ‹*is-Done* (*resumption-lub R*)› *ub*
show *?case* **by**(*auto simp add*: *flat-lub-def dest*: *resumption-ord-abort*)
next
case (*Pause R r*)
hence *chain*: *Complete-Partial-Order.chain resumption-ord R*

and *ub*: $\bigwedge r'. \, r' \in R \Longrightarrow$ *resumption-ord* r' r **by** *simp-all*
from ‹¬ *is-Done* (*resumption-lub R*)› **have** *exR*: $\exists\, r \in R.\ \neg$ *is-Done r* **by** *simp*
then obtain r' **where** r': $r' \in R$ ¬ *is-Done* r' **by** *auto*
with *ub*[*of* r'] **have** *output r* = *output* r' **by**(*auto dest*: *resumption-ord-outputD*)
also have [*symmetric*]: *output* (*resumption-lub R*) = *output* r' **using** *exR* r'
by(*auto 4 4 elim*: *chainE*[*OF chain*] *dest*: *resumption-ord-outputD*)
finally have *?output* **..**
moreover
{ fix *inp* r''
assume $r'' \in R$ ¬ *is-Done* r''
with *ub*[*of* r'']
have *resumption-ord* (*resume* r'' *inp*) (*resume r inp*)
by(*auto dest*!: *resumption-ord-resumeD simp add*: *rel-fun-def*) **}**
with *exR resumption-ord-chain-resume*[*OF chain*] r'
have *?resume* **by**(*auto simp add*: *rel-fun-def*)
ultimately show *?case* **..**
qed
qed

interpretation *resumption*:
partial-function-definitions resumption-ord resumption-lub
rewrites *resumption-lub* {} = (*ABORT* :: ('*a*, '*b*, '*c*) *resumption*)
by (*rule resumption-partial-function-definition resumption-lub-empty*)+

declaration ‹*Partial-Function.init resumption* @{*term resumption.fixp-fun*}
@{*term resumption.mono-body*} @{*thm resumption.fixp-rule-uc*} @{*thm resumption.fixp-induct-uc*} *NONE*›

abbreviation *mono-resumption* ≡ *monotone* (*fun-ord resumption-ord*) *resumption-ord*

lemma *mono-resumption-resume*:
assumes *mono-resumption B*
shows *mono-resumption* (λ*f*. *resume* (*B f*) *inp*)
proof
fix *f g* :: '*a* ⇒ ('*b*, '*c*, '*d*) *resumption*
assume *fg*: *fun-ord resumption-ord f g*
hence *resumption-ord* (*B f*) (*B g*) **by**(*rule monotoneD*[*OF assms*])
with *resumption-ord-resumeD*[*OF this*]
show *resumption-ord* (*resume* (*B f*) *inp*) (*resume* (*B g*) *inp*)
by(*cases is-Done* (*B f*))(*auto simp add*: *rel-fun-def is-Done-def*)
qed

lemma *bind-resumption-mono* [*partial-function-mono*]:
assumes *mf*: *mono-resumption B*
and *mg*: ⋀*y*. *mono-resumption* (*C y*)
shows *mono-resumption* (λ*f*. *do* { *y* ← *B f*; *C y f* })
proof(*rule monotoneI*)
fix *f g* :: '*a* ⇒ ('*b*, '*c*, '*d*) *resumption*

assume ∗: *fun-ord resumption-ord f g*
define f' **where** $f' \equiv B\ f$ **define** g' **where** $g' \equiv B\ g$
define h **where** $h \equiv \lambda x.\ C\ x\ f$ **define** k **where** $k \equiv \lambda x.\ C\ x\ g$
from *mf*[*THEN monotoneD*, *OF* ∗] *mg*[*THEN monotoneD*, *OF* ∗] *f′-def g′-def h-def k-def*
have *resumption-ord* f' g' $\bigwedge x.$ *resumption-ord* $(h\ x)$ $(k\ x)$ **by** *auto*
thus *resumption-ord* $(f' \ggg h)$ $(g' \ggg k)$
proof(*coinduction arbitrary*: f' g' h k)
case (*Done* f' g' h k)
hence *le*: *resumption-ord* f' g'
and *mg*: $\bigwedge y.$ *resumption-ord* $(h\ y)$ $(k\ y)$ **by** *simp-all*
from ‹*is-Done* $(g' \ggg k)$›
have *done-Bg*: *is-Done* g'
and *result* $g' \neq$ *None* $\Longrightarrow$ *is-Done* (*k* (*the* (*result* g'))) **by** *simp-all*
moreover
have *is-Done* f' **using** *le done-Bg* **by**(*rule resumption-ord-is-DoneD*)
moreover
from *le done-Bg* **have** *flat-ord None* (*result* f') (*result* g')
by(*rule resumption-ord-resultD*)
hence *result* $f' \neq$ *None* $\Longrightarrow$ *result* g' = *result* f'
by(*auto simp add*: *flat-ord-def*)
moreover
have *resumption-ord* (*h* (*the* (*result* f'))) (*k* (*the* (*result* f'))) **by**(*rule mg*)
ultimately show *?case*
by(*subst* (*1 2*) *result-bind-resumption*)(*auto dest*: *resumption-ord-is-DoneD resumption-ord-resultD simp add*: *flat-ord-def bind-eq-None-conv*)
next
case (*Abort* f' g' h k)
hence *resumption-ord* (*h* (*the* (*result* f'))) (*k* (*the* (*result* f'))) **by** *simp*
thus *?case* **using** *Abort*
by(*cases is-Done* g')(*auto 4 4 simp add*: *bind-eq-None-conv flat-ord-def dest*: *resumption-ord-abort resumption-ord-resultD resumption-ord-is-DoneD*)
next
case (*Pause* f' g' h k)
hence *?output*
by(*auto 4 4 dest*: *resumption-ord-outputD resumption-ord-is-DoneD resumption-ord-resultD resumption-ord-abort simp add*: *flat-ord-def*)
moreover have *?resume*
proof(*cases is-Done* f')
case *False*
with *Pause* **show** *?thesis*
by(*auto simp add*: *rel-fun-def dest*: *resumption-ord-is-DoneD intro*: *resumption-ord-resumeD*[*THEN rel-funD*] *del*: *exI intro*!: *exI*)
next
case *True*
hence *is-Done* g' **using** *Pause* **by**(*auto dest*: *resumption-ord-abort*)
thus *?thesis* **using** *True Pause resumption-ord-resultD*[*OF* ‹*resumption-ord* f' g'›]
by(*auto del*: *rel-funI intro*!: *rel-funI simp add*: *bind-resumption-is-Done*

flat-ord-def intro: *resumption-ord-resumeD*[*THEN rel-funD*] *exI*[**where** *x=f'*] *exI*[**where** *x=g'*])
qed
ultimately show *?case* **..**
qed
qed

lemma fixes *f F*
defines $F \equiv \lambda results\ r.\ case\ r\ of\ resumption.Done\ x \Rightarrow set\text{-}option\ x \mid resumption.Pause\ out\ c \Rightarrow \bigcup input.\ results\ (c\ input)$
shows *results-conv-fixp*: $results \equiv ccpo.fixp\ (fun\text{-}lub\ Union)\ (fun\text{-}ord\ (\subseteq))\ F$ (**is** $- \equiv ?fixp$)
and *results-mono*: $\bigwedge x.\ monotone\ (fun\text{-}ord\ (\subseteq))\ (\subseteq)\ (\lambda f.\ F\ f\ x)$ (**is** *PROP ?mono*)
proof(*rule eq-reflection ext antisym subsetI*)+
show *mono*: *PROP ?mono* **unfolding** *F-def* **by**(*tactic* ‹*Partial-Function.mono-tac* @{*context*} *1*›)
fix *x r*
show $?fixp\ r \subseteq results\ r$
by(*induction arbitrary*: *r rule*: *lfp.fixp-induct-uc*[*of* $\lambda x.\ x\ F\ \lambda x.\ x$, *OF mono reflexive refl*])
(*fastforce simp add*: *F-def split*: *resumption.split-asm*)+

assume $x \in results\ r$
thus $x \in ?fixp\ r$ **by** *induct*(*subst lfp.mono-body-fixp*[*OF mono*]; *auto simp add*: *F-def*)+
qed

lemma *mcont-case-resumption*:
fixes *f g*
defines $h \equiv \lambda r.\ if\ is\text{-}Done\ r\ then\ f\ (result\ r)\ else\ g\ (output\ r)\ (resume\ r)\ r$
assumes *mcont1*: *mcont* (*flat-lub None*) *option-ord lub ord f*
and *mcont2*: $\bigwedge out.$ *mcont* (*fun-lub resumption-lub*) (*fun-ord resumption-ord*) *lub ord* ($\lambda c.$ *g out c* (*Pause out c*))
and *ccpo*: *class.ccpo lub ord* (*mk-less ord*)
and *bot*: $\bigwedge x.\ ord\ (f\ None)\ x$
shows *mcont resumption-lub resumption-ord lub ord* ($\lambda r.$ *case r of Done x* $\Rightarrow$ *f x* | *Pause out c* $\Rightarrow$ *g out c r*)
(**is** *mcont ?lub ?ord - - ?f*)
proof(*rule resumption.mcont-if-bot*[*OF ccpo bot*, **where** *bound=ABORT* **and** *f=h*])
show *?f x* = (*if ?ord x ABORT then f None else h x*) **for** *x*
by(*simp add*: *h-def split*: *resumption.split*)
show *ord* (*h x*) (*h y*) **if** *?ord x y* ¬ *?ord x ABORT* **for** *x y* **using** *that*
by(*cases x*)(*simp-all add*: *h-def mcont-monoD*[*OF mcont1*] *fun-ord-conv-rel-fun mcont-monoD*[*OF mcont2*])

fix *Y* :: ('*a*, '*b*, '*c*) *resumption set*
assume *chain*: *Complete-Partial-Order.chain ?ord Y*
and *Y*: $Y \neq \{\}$
and *nbot*: $\bigwedge x.\ x \in Y \Longrightarrow \neg\ ?ord\ x\ ABORT$

show *h* (*?lub Y*) = *lub* (*h* ‘ *Y*)
proof(*cases* ∃ *x*. *DONE x* ∈ *Y*)
case *True*
then obtain *x* **where** *x*: *DONE x* ∈ *Y* **..**
have *is-Done*: *is-Done r* **if** *r* ∈ *Y* **for** *r* **using** *chainD*[*OF chain that x*]
by(*auto dest*: *resumption-ord-is-DoneD*)
from *is-Done* **have** *chain′*: *Complete-Partial-Order.chain* (*flat-ord None*) (*result* ‘ *Y*)
by(*auto 5 2 intro*!: *chainI elim*: *chainE*[*OF chain*] *dest*: *resumption-ord-resultD*)
from *is-Done* **have** *is-Done* (*?lub Y*) *Y* ∩ {*r*. *is-Done r*} = *Y Y* ∩ {*r*. ¬ *is-Done r*} = {} **by** *auto*
then show *?thesis* **using** *Y* **by**(*simp add*: *h-def mcont-contD*[*OF mcont1 chain′*] *image-image*)
next
case *False*
have *is-Done*: ¬ *is-Done r* **if** *r* ∈ *Y* **for** *r* **using** *that False nbot*
by(*auto elim*!: *is-Done-cases*)
from *Y* **obtain** *out c* **where** *Pause*: *Pause out c* ∈ *Y*
by(*auto 5 2 dest*: *is-Done iff*: *not-is-Done-conv-Pause*)

have *out*: (*THE out*. *out* ∈ *output* ‘ (*Y* ∩ {*r*. ¬ *is-Done r*})) = *out* **using** *Pause*
by(*auto 4 3 intro*: *rev-image-eqI iff*: *not-is-Done-conv-Pause dest*: *chainD*[*OF chain*])
have (λ*r*. *g* (*output r*) (*resume r*) *r*) ‘ (*Y* ∩ {*r*. ¬ *is-Done r*}) = (λ*r*. *g out* (*resume r*) *r*) ‘ (*Y* ∩ {*r*. ¬ *is-Done r*})
by(*auto 4 3 simp add*: *not-is-Done-conv-Pause dest*: *chainD*[*OF chain Pause*] *intro*: *rev-image-eqI*)
moreover have ¬ *is-Done* (*?lub Y*) **using** *Y is-Done* **by**(*auto*)
moreover from *is-Done* **have** *Y* ∩ {*r*. *is-Done r*} = {} *Y* ∩ {*r*. ¬ *is-Done r*} = *Y* **by** *auto*
moreover have (λ*inp*. *resumption-lub* ((λ*x*. *resume x inp*) ‘ *Y*)) = *fun-lub resumption-lub* (*resume* ‘ *Y*)
by(*auto simp add*: *fun-lub-def fun-eq-iff intro*!: *arg-cong*[**where** *f*=*resumption-lub*])
moreover have *resumption-lub Y* = *Pause out* (*fun-lub resumption-lub* (*resume* ‘ *Y*))
using *Y is-Done out*
by(*intro resumption.expand*)(*auto simp add*: *fun-lub-def fun-eq-iff image-image intro*!: *arg-cong*[**where** *f*=*resumption-lub*])
moreover have *chain′*: *Complete-Partial-Order.chain resumption.le-fun* (*resume* ‘ *Y*) **using** *chain*
by(*rule chain-imageI*)(*auto dest*!: *is-Done simp add*: *not-is-Done-conv-Pause fun-ord-conv-rel-fun*)
moreover have (λ*r*. *g out* (*resume r*) (*Pause out* (*resume r*))) ‘ *Y* = (λ*r*. *g out* (*resume r*) *r*) ‘ *Y*
by(*intro image-cong*[*OF refl*])(*frule nbot*; *auto dest*!: *chainD*[*OF chain Pause*] *elim*: *resumption-ord.cases*)
ultimately show *?thesis* **using** *False out Y*
by(*simp add*: *h-def image-image mcont-contD*[*OF mcont2*])

qed
qed

lemma *mcont2mcont-results*[*THEN mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-results*: *mcont resumption-lub resumption-ord Union* (⊆) *results*
apply(*rule lfp.fixp-preserves-mcont1* [*OF results-mono results-conv-fixp*])
apply(*rule mcont-case-resumption*)
apply(*simp-all add*: *mcont-applyI*)
done

lemma *mono2mono-results*[*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-results*: *monotone resumption-ord* (⊆) *results*
using *mcont-results* **by**(*rule mcont-mono*)

lemma fixes *f F*
defines *F* ≡ *λoutputs xs. case xs of resumption.Done x* ⇒ {} | *resumption.Pause out c* ⇒ *insert out* (⋃ *input. outputs* (*c input*))
shows *outputs-conv-fixp*: *outputs* ≡ *ccpo.fixp* (*fun-lub Union*) (*fun-ord* (⊆)) *F* (**is** - ≡ *?fixp*)
and *outputs-mono*: ⋀*x. monotone* (*fun-ord* (⊆)) (⊆) (*λf. F f x*) (**is** *PROP ?mono*)
proof(*rule eq-reflection ext antisym subsetI*)+
show *mono*: *PROP ?mono* **unfolding** *F-def* **by**(*tactic* ‹*Partial-Function.mono-tac* @{*context*} *1*›)
show *?fixp r* ⊆ *outputs r* **for** *r*
by(*induct arbitrary*: *r rule*: *lfp.fixp-induct-uc*[*of λx. x F λx. x*, *OF mono reflexive refl*])(*auto simp add*: *F-def split*: *resumption.split*)
show *x* ∈ *?fixp r* **if** *x* ∈ *outputs r* **for** *x r* **using** *that*
by *induct*(*subst lfp.mono-body-fixp*[*OF mono*]; *auto simp add*: *F-def*; *fail*)+
qed

lemma *mcont2mcont-outputs*[*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-outputs*: *mcont resumption-lub resumption-ord Union* (⊆) *outputs*
apply(*rule lfp.fixp-preserves-mcont1* [*OF outputs-mono outputs-conv-fixp*])
apply(*auto intro*: *lfp.mcont2mcont intro*!: *mcont2mcont-insert mcont-SUP mcont-case-resumption*)
done

lemma *mono2mono-outputs*[*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-outputs*: *monotone resumption-ord* (⊆) *outputs*
using *mcont-outputs* **by**(*rule mcont-mono*)

lemma *pred-resumption-antimono*:
assumes *r*: *pred-resumption A C r′*
and *le*: *resumption-ord r r′*
shows *pred-resumption A C r*
using *r monotoneD*[*OF monotone-results le*] *monotoneD*[*OF monotone-outputs le*]
by(*auto simp add*: *pred-resumption-def*)

3.2 Setup for lifting and transfer

declare *resumption.rel-eq* [*id-simps*, *relator-eq*]
declare *resumption.rel-mono* [*relator-mono*]

lemma *rel-resumption-OO* [*relator-distr*]:
rel-resumption A B OO rel-resumption C D = *rel-resumption* (*A OO C*) (*B OO D*)
by(*simp add*: *resumption.rel-compp*)

lemma *left-total-rel-resumption* [*transfer-rule*]:
⟦ *left-total R1*; *left-total R2* ⟧ ⟹ *left-total* (*rel-resumption R1 R2*)
by(*simp only*: *left-total-alt-def resumption.rel-eq*[*symmetric*] *resumption.rel-conversep*[*symmetric*] *rel-resumption-OO resumption.rel-mono*)

lemma *left-unique-rel-resumption* [*transfer-rule*]:
⟦ *left-unique R1*; *left-unique R2* ⟧ ⟹ *left-unique* (*rel-resumption R1 R2*)
by(*simp only*: *left-unique-alt-def resumption.rel-eq*[*symmetric*] *resumption.rel-conversep*[*symmetric*] *rel-resumption-OO resumption.rel-mono*)

lemma *right-total-rel-resumption* [*transfer-rule*]:
⟦ *right-total R1*; *right-total R2* ⟧ ⟹ *right-total* (*rel-resumption R1 R2*)
by(*simp only*: *right-total-alt-def resumption.rel-eq*[*symmetric*] *resumption.rel-conversep*[*symmetric*] *rel-resumption-OO resumption.rel-mono*)

lemma *right-unique-rel-resumption* [*transfer-rule*]:
⟦ *right-unique R1*; *right-unique R2* ⟧ ⟹ *right-unique* (*rel-resumption R1 R2*)
by(*simp only*: *right-unique-alt-def resumption.rel-eq*[*symmetric*] *resumption.rel-conversep*[*symmetric*] *rel-resumption-OO resumption.rel-mono*)

lemma *bi-total-rel-resumption* [*transfer-rule*]:
⟦ *bi-total A*; *bi-total B* ⟧ ⟹ *bi-total* (*rel-resumption A B*)
unfolding *bi-total-alt-def*
by(*blast intro*: *left-total-rel-resumption right-total-rel-resumption*)

lemma *bi-unique-rel-resumption* [*transfer-rule*]:
⟦ *bi-unique A*; *bi-unique B* ⟧ ⟹ *bi-unique* (*rel-resumption A B*)
unfolding *bi-unique-alt-def*
by(*blast intro*: *left-unique-rel-resumption right-unique-rel-resumption*)

lemma *Quotient-resumption* [*quot-map*]:
⟦ *Quotient R1 Abs1 Rep1 T1*; *Quotient R2 Abs2 Rep2 T2* ⟧
⟹ *Quotient* (*rel-resumption R1 R2*) (*map-resumption Abs1 Abs2*) (*map-resumption Rep1 Rep2*) (*rel-resumption T1 T2*)
by(*simp add*: *Quotient-alt-def5 resumption.rel-Grp*[*of UNIV - UNIV -*, *symmetric*, *simplified*] *resumption.rel-compp resumption.rel-conversep*[*symmetric*] *resumption.rel-mono*)

end

4 Generative probabilistic values

theory *Generat* **imports**
Misc-CryptHOL
begin

4.1 Single-step generative

datatype (*generat-pures*: *'a*, *generat-outs*: *'b*, *generat-conts*: *'c*) *generat*
= *Pure* (*result*: *'a*)
| *IO* (*output*: *'b*) (*continuation*: *'c*)

datatype-compat *generat*

lemma *IO-code-cong*: $out = out' \Longrightarrow IO\ out\ c = IO\ out'\ c$ **by** *simp*
setup ‹*Code-Simp.map-ss* (*Simplifier.add-cong* @{*thm IO-code-cong*})›

lemma *is-Pure-map-generat* [*simp*]: *is-Pure* (*map-generat f g h x*) = *is-Pure x*
by(*cases x*) *simp-all*

lemma *result-map-generat* [*simp*]: *is-Pure x* ⟹ *result* (*map-generat f g h x*) = *f* (*result x*)
by(*cases x*) *simp-all*

lemma *output-map-generat* [*simp*]: ¬ *is-Pure x* ⟹ *output* (*map-generat f g h x*) = *g* (*output x*)
by(*cases x*) *simp-all*

lemma *continuation-map-generat* [*simp*]: ¬ *is-Pure x* ⟹ *continuation* (*map-generat f g h x*) = *h* (*continuation x*)
by(*cases x*) *simp-all*

lemma [*simp*]:
shows *map-generat-eq-Pure*:
map-generat f g h generat = *Pure x* ⟷ (∃ *x'*. *generat* = *Pure x'* ∧ *x* = *f x'*)
and *Pure-eq-map-generat*:
Pure x = *map-generat f g h generat* ⟷ (∃ *x'*. *generat* = *Pure x'* ∧ *x* = *f x'*)
by(*cases generat*; *auto*; *fail*)+

lemma [*simp*]:
shows *map-generat-eq-IO*:
map-generat f g h generat = *IO out c* ⟷ (∃ *out' c'*. *generat* = *IO out' c'* ∧ *out* = *g out'* ∧ *c* = *h c'*)
and *IO-eq-map-generat*:
IO out c = *map-generat f g h generat* ⟷ (∃ *out' c'*. *generat* = *IO out' c'* ∧ *out* = *g out'* ∧ *c* = *h c'*)
by(*cases generat*; *auto*; *fail*)+

lemma *is-PureE* [*cases pred*]:
assumes *is-Pure generat*

obtains (*Pure*) *x* **where** *generat* = *Pure x*
using *assms* **by**(*auto simp add*: *is-Pure-def*)

lemma *not-is-PureE*:
assumes ¬ *is-Pure generat*
obtains (*IO*) *out c* **where** *generat* = *IO out c*
using *assms* **by**(*cases generat*) *auto*

lemma *rel-generatI*:
⟦ *is-Pure x* ⟷ *is-Pure y*;
⟦ *is-Pure x*; *is-Pure y* ⟧ ⟹ *A* (*result x*) (*result y*);
⟦ ¬ *is-Pure x*; ¬ *is-Pure y* ⟧ ⟹ *Out* (*output x*) (*output y*) ∧ *R* (*continuation x*) (*continuation y*) ⟧
⟹ *rel-generat A Out R x y*
by(*cases x y rule*: *generat.exhaust*[*case-product generat.exhaust*]) *simp-all*

lemma *rel-generatD′*:
rel-generat A Out R x y
⟹ (*is-Pure x* ⟷ *is-Pure y*) ∧
(*is-Pure x* ⟶ *is-Pure y* ⟶ *A* (*result x*) (*result y*)) ∧
(¬ *is-Pure x* ⟶ ¬ *is-Pure y* ⟶ *Out* (*output x*) (*output y*) ∧ *R* (*continuation x*) (*continuation y*))
by(*cases x y rule*: *generat.exhaust*[*case-product generat.exhaust*]) *simp-all*

lemma *rel-generatD*:
assumes *rel-generat A Out R x y*
shows *rel-generat-is-PureD*: *is-Pure x* ⟷ *is-Pure y*
and *rel-generat-resultD*: *is-Pure x* ∨ *is-Pure y* ⟹ *A* (*result x*) (*result y*)
and *rel-generat-outputD*: ¬ *is-Pure x* ∨ ¬ *is-Pure y* ⟹ *Out* (*output x*) (*output y*)
and *rel-generat-continuationD*: ¬ *is-Pure x* ∨ ¬ *is-Pure y* ⟹ *R* (*continuation x*) (*continuation y*)
using *rel-generatD′*[*OF assms*] **by** *simp-all*

lemma *rel-generat-mono*:
⟦ *rel-generat A B C x y*; ⋀*x y*. *A x y* ⟹ *A′ x y*; ⋀*x y*. *B x y* ⟹ *B′ x y*; ⋀*x y*. *C x y* ⟹ *C′ x y* ⟧
⟹ *rel-generat A′ B′ C′ x y*
using *generat.rel-mono*[*of A A′ B B′ C C′*] **by**(*auto simp add*: *le-fun-def*)

lemma *rel-generat-mono′* [*mono*]:
⟦ ⋀*x y*. *A x y* ⟶ *A′ x y*; ⋀*x y*. *B x y* ⟶ *B′ x y*; ⋀*x y*. *C x y* ⟶ *C′ x y* ⟧
⟹ *rel-generat A B C x y* ⟶ *rel-generat A′ B′ C′ x y*
by(*blast intro*: *rel-generat-mono*)

lemma *rel-generat-same*:
rel-generat A B C r r ⟷
(∀ *x* ∈ *generat-pures r*. *A x x*) ∧
(∀ *out* ∈ *generat-outs r*. *B out out*) ∧

(∀ *c* ∈*generat-conts r*. *C c c*)
by(*cases r*)(*auto simp add*: *rel-fun-def*)

lemma *rel-generat-reflI*:
⟦ ⋀*y*. *y* ∈ *generat-pures x* ⟹ *A y y*;
⋀*out*. *out* ∈ *generat-outs x* ⟹ *B out out*;
⋀*cont*. *cont* ∈ *generat-conts x* ⟹ *C cont cont* ⟧
⟹ *rel-generat A B C x x*
by(*cases x*) *auto*

lemma *reflp-rel-generat* [*simp*]: *reflp* (*rel-generat A B C*) ⟷ *reflp A* ∧ *reflp B* ∧ *reflp C*
by(*auto 4 3 intro*!: *reflpI rel-generatI dest*: *reflpD reflpD*[**where** *x*=*Pure* -] *reflpD*[**where** *x*=*IO* - -])

lemma *transp-rel-generatI*:
assumes *transp A transp B transp C*
shows *transp* (*rel-generat A B C*)
by(*rule transpI*)(*auto 6 5 dest*: *rel-generatD′ intro*!: *rel-generatI intro*: *assms*[*THEN transpD*] *simp add*: *rel-fun-def*)

lemma *rel-generat-inf*:
inf (*rel-generat A B C*) (*rel-generat A′ B′ C′*) = *rel-generat* (*inf A A′*) (*inf B B′*) (*inf C C′*)
(**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by(*auto elim*!: *generat.rel-cases simp add*: *rel-fun-def*)
qed(*auto elim*: *rel-generat-mono*)

lemma *rel-generat-Pure1*: *rel-generat A B C* (*Pure x*) = (λ*r*. ∃ *y*. *r* = *Pure y* ∧ *A x y*)
by(*rule ext*)(*case-tac r*, *simp-all*)

lemma *rel-generat-IO1*: *rel-generat A B C* (*IO out c*) = (λ*r*. ∃ *out′ c′*. *r* = *IO out′ c′* ∧ *B out out′* ∧ *C c c′*)
by(*rule ext*)(*case-tac r*, *simp-all*)

lemma *not-is-Pure-conv*: ¬ *is-Pure r* ⟷ (∃ *out c*. *r* = *IO out c*)
by(*cases r*) *auto*

lemma *finite-generat-outs* [*simp*]: *finite* (*generat-outs generat*)
by(*cases generat*) *auto*

lemma *countable-generat-outs* [*simp*]: *countable* (*generat-outs generat*)
by(*simp add*: *countable-finite*)

lemma *case-map-generat*:
case-generat pure io (*map-generat a b d r*) =

case-generat (*pure* ∘ *a*) (λ*out*. *io* (*b out*) ∘ *d*) *r*
by(*cases r*) *simp-all*

lemma *continuation-in-generat-conts*:
¬ *is-Pure r* ⟹ *continuation r* ∈ *generat-conts r*
by(*cases r*) *auto*

fun *dest-IO* :: (′*a*, ′*out*, ′*c*) *generat* ⇒ (′*out* × ′*c*) *option*
where
dest-IO (*Pure* -) = *None*
| *dest-IO* (*IO out c*) = *Some* (*out*, *c*)

lemma *dest-IO-eq-Some-iff* [*simp*]: *dest-IO generat* = *Some* (*out*, *c*) ⟷ *generat* = *IO out c*
by(*cases generat*) *simp-all*

lemma *dest-IO-eq-None-iff* [*simp*]: *dest-IO generat* = *None* ⟷ *is-Pure generat*
by(*cases generat*) *simp-all*

lemma *dest-IO-comp-Pure* [*simp*]: *dest-IO* ∘ *Pure* = (λ-. *None*)
by(*simp add*: *fun-eq-iff*)

lemma *dom-dest-IO*: *dom dest-IO* = {*x*. ¬ *is-Pure x*}
by(*auto simp add*: *not-is-Pure-conv*)

definition *generat-lub* :: (′*a set* ⇒ ′*b*) ⇒ (′*out set* ⇒ ′*out*′) ⇒ (′*cont set* ⇒ ′*cont*′)

⇒ (′*a*, ′*out*, ′*cont*) *generat set* ⇒ (′*b*, ′*out*′, ′*cont*′) *generat*
where
generat-lub lub1 lub2 lub3 A =
(*if* ∃ *x*∈*A*. *is-Pure x then Pure* (*lub1* (*result* ‘ (*A* ∩ {*f*. *is-Pure f*})))
else IO (*lub2* (*output* ‘ (*A* ∩ {*f*. ¬ *is-Pure f*}))) (*lub3* (*continuation* ‘ (*A* ∩ {*f*. ¬ *is-Pure f*}))))

lemma *is-Pure-generat-lub* [*simp*]:
is-Pure (*generat-lub lub1 lub2 lub3 A*) ⟷ (∃ *x*∈*A*. *is-Pure x*)
by(*simp add*: *generat-lub-def*)

lemma *result-generat-lub* [*simp*]:
∃ *x*∈*A*. *is-Pure x* ⟹ *result* (*generat-lub lub1 lub2 lub3 A*) = *lub1* (*result* ‘ (*A* ∩ {*f*. *is-Pure f*}))
by(*simp add*: *generat-lub-def*)

lemma *output-generat-lub*:
∀ *x*∈*A*. ¬ *is-Pure x* ⟹ *output* (*generat-lub lub1 lub2 lub3 A*) = *lub2* (*output* ‘ (*A* ∩ {*f*. ¬ *is-Pure f*}))
by(*simp add*: *generat-lub-def*)

lemma *continuation-generat-lub*:
$\forall x \in A.\ \neg$ *is-Pure* $x \Longrightarrow$ *continuation* (*generat-lub lub1 lub2 lub3 A*) = *lub3* (*continuation* ‘ ($A \cap \{f.\ \neg$ *is-Pure* $f\}$))
by(*simp add*: *generat-lub-def*)

lemma *generat-lub-map* [*simp*]:
generat-lub lub1 lub2 lub3 (*map-generat f g h* ‘ *A*) = *generat-lub* (*lub1* ∘ (‘) *f*) (*lub2* ∘ (‘) *g*) (*lub3* ∘ (‘) *h*) *A*
by(*auto 4 3 simp add*: *generat-lub-def intro*: *arg-cong*[**where** *f*=*lub1*] *arg-cong*[**where** *f*=*lub2*] *arg-cong*[**where** *f*=*lub3*] *rev-image-eqI del*: *ext intro*!: *ext*)

lemma *map-generat-lub* [*simp*]:
map-generat f g h (*generat-lub lub1 lub2 lub3 A*) = *generat-lub* (*f* ∘ *lub1*) (*g* ∘ *lub2*) (*h* ∘ *lub3*) *A*
by(*simp add*: *generat-lub-def o-def*)

abbreviation *generat-lub'* :: (*'cont set* ⇒ *'cont'*) ⇒ (*'a*, *'out*, *'cont*) *generat set* ⇒ (*'a*, *'out*, *'cont'*) *generat*
where *generat-lub'* ≡ *generat-lub* (λ*A*. *THE x*. *x* ∈ *A*) (λ*A*. *THE x*. *x* ∈ *A*)

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

lemma *rel-witness-generat*:
assumes *rel-generat A C R x y*
shows *pures-rel-witness-generat*: *generat-pures* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*a*, *b*). *A a b*}
and *outs-rel-witness-generat*: *generat-outs* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*c*, *d*). *C c d*}
and *conts-rel-witness-generat*: *generat-conts* (*rel-witness-generat* (*x*, *y*)) ⊆ {(*e*, *f*). *R e f*}
and *map1-rel-witness-generat*: *map-generat fst fst fst* (*rel-witness-generat* (*x*, *y*)) = *x*
and *map2-rel-witness-generat*: *map-generat snd snd snd* (*rel-witness-generat* (*x*, *y*)) = *y*
using *assms* **by**(*cases*; *simp*; *fail*)+

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

lemma *rel-witness-generat1*:
assumes *rel-generat A C R x y*
shows *rel-generat* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) (λ*c* (*c'*, *d*). *c* = *c'* ∧ *C c' d*) (λ*r* (*r'*, *s*). *r* = *r'* ∧ *R r' s*) *x* (*rel-witness-generat* (*x*, *y*))
using *map1-rel-witness-generat*[*OF assms*, *symmetric*]

unfolding *generat.rel-eq*[*symmetric*] *generat.rel-map*
by(*rule generat.rel-mono-strong*)(*auto dest*: *set-rel-witness-generat*[*OF assms*, *THEN subsetD*])

lemma *rel-witness-generat2*:
assumes *rel-generat A C R x y*
shows *rel-generat* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (λ(*c*, *d'*) *d*. *d* = *d'* ∧ *C c d'*) (λ(*r*, *s'*) *s*. *s* = *s'* ∧ *R r s'*) (*rel-witness-generat* (*x*, *y*)) *y*
using *map2-rel-witness-generat*[*OF assms*]
unfolding *generat.rel-eq*[*symmetric*] *generat.rel-map*
by(*rule generat.rel-mono-strong*)(*auto dest*: *set-rel-witness-generat*[*OF assms*, *THEN subsetD*])

end

theory *Generative-Probabilistic-Value* **imports**
Resumption
Generat
HOL−Types-To-Sets.Types-To-Sets
begin

hide-const (**open**) *Done*

4.2 Type definition

context notes [[*bnf-internals*]] **begin**

codatatype (*results'-gpv*: *'a*, *outs'-gpv*: *'out*, *'in*) *gpv*
= *GPV* (*the-gpv*: (*'a*, *'out*, *'in* ⇒ (*'a*, *'out*, *'in*) *gpv*) *generat spmf*)

end

declare *gpv.rel-eq* [*relator-eq*]

Reactive values are like generative, except that they take an input first.

type-synonym (*'a*, *'out*, *'in*) *rpv* = *'in* ⇒ (*'a*, *'out*, *'in*) *gpv*
print-translation — pretty printing for (*'a*, *'out*, *'in*) *rpv* ‹
let
fun tr' [*in1*, *Const* (@{*type-syntax gpv*}, -) $ *a* $ *out* $ *in2*] =
if in1 = *in2 then Syntax.const* @{*type-syntax rpv*} $ *a* $ *out* $ *in1*
else raise Match;
in [(@{*type-syntax fun*}, *K tr'*)]
end
›
typ (*'a*, *'out*, *'in*) *rpv*

Effectively, (*'a*, *'out*, *'in*) *gpv* and (*'a*, *'out*, *'in*) *rpv* are mutually recursive.

lemma *eq-GPV-iff*: *f* = *GPV g* ⟷ *the-gpv f* = *g*

by(*cases f*) *auto*

declare *gpv.set*[*simp del*]

declare *gpv.set-map*[*simp*]

lemma *rel-gpv-def'*:
  rel-gpv A B gpv gpv' $\longleftrightarrow$
  ($\exists$ *gpv''*. ($\forall$ (*x*, *y*) $\in$ *results'-gpv gpv''*. *A x y*) $\wedge$ ($\forall$ (*x*, *y*) $\in$ *outs'-gpv gpv''*. *B x y*) $\wedge$
    map-gpv fst fst gpv'' = *gpv* $\wedge$ *map-gpv snd snd gpv''* = *gpv'*)
unfolding *rel-gpv-def* **by**(*auto simp add*: *BNF-Def.Grp-def*)

definition *results'-rpv* :: (*'a*, *'out*, *'in*) *rpv* $\Rightarrow$ *'a set*
where *results'-rpv rpv* = *range rpv* $\gg=$ *results'-gpv*

definition *outs'-rpv* :: (*'a*, *'out*, *'in*) *rpv* $\Rightarrow$ *'out set*
where *outs'-rpv rpv* = *range rpv* $\gg=$ *outs'-gpv*

abbreviation *rel-rpv*
  :: (*'a* $\Rightarrow$ *'b* $\Rightarrow$ *bool*) $\Rightarrow$ (*'out* $\Rightarrow$ *'out'* $\Rightarrow$ *bool*)
  $\Rightarrow$ (*'in* $\Rightarrow$ (*'a*, *'out*, *'in*) *gpv*) $\Rightarrow$ (*'in* $\Rightarrow$ (*'b*, *'out'*, *'in*) *gpv*) $\Rightarrow$ *bool*
where *rel-rpv A B* $\equiv$ *rel-fun* (=) (*rel-gpv A B*)

lemma *in-results'-rpv* [*iff*]: *x* $\in$ *results'-rpv rpv* $\longleftrightarrow$ ($\exists$ *input*. *x* $\in$ *results'-gpv* (*rpv input*))
by(*simp add*: *results'-rpv-def*)

lemma *in-outs-rpv* [*iff*]: *out* $\in$ *outs'-rpv rpv* $\longleftrightarrow$ ($\exists$ *input*. *out* $\in$ *outs'-gpv* (*rpv input*))
by(*simp add*: *outs'-rpv-def*)

lemma *results'-GPV* [*simp*]:
  results'-gpv (*GPV r*) =
  (*set-spmf r* $\gg=$ *generat-pures*) $\cup$
  ((*set-spmf r* $\gg=$ *generat-conts*) $\gg=$ *results'-rpv*)
by(*auto simp add*: *gpv.set bind-UNION set-spmf-def*)

lemma *outs'-GPV* [*simp*]:
  outs'-gpv (*GPV r*) =
  (*set-spmf r* $\gg=$ *generat-outs*) $\cup$
  ((*set-spmf r* $\gg=$ *generat-conts*) $\gg=$ *outs'-rpv*)
by(*auto simp add*: *gpv.set bind-UNION set-spmf-def*)

lemma *outs'-gpv-unfold*:
  outs'-gpv r =
  (*set-spmf* (*the-gpv r*) $\gg=$ *generat-outs*) $\cup$
  ((*set-spmf* (*the-gpv r*) $\gg=$ *generat-conts*) $\gg=$ *outs'-rpv*)
by(*cases r*) *simp*

lemma *outs′-gpv-induct* [*consumes 1*, *case-names Out Cont*, *induct set*: *outs′-gpv*]:
assumes *x*: $x \in$ *outs′-gpv gpv*
and *Out*: ⋀*generat gpv*. ⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $x \in$ *generat-outs generat* ⟧ $\Longrightarrow$ *P gpv*
and *Cont*: ⋀*generat gpv c input*.
⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $c \in$ *generat-conts generat*; $x \in$ *outs′-gpv* (*c input*); *P* (*c input*) ⟧ $\Longrightarrow$ *P gpv*
shows *P gpv*
using *x*
apply(*induction y≡x gpv*)
apply(*rule Out*, *simp add*: *in-set-spmf*, *simp*)
apply(*erule imageE*, *rule Cont*, *simp add*: *in-set-spmf*, *simp*, *simp*, *simp*)
.

lemma *outs′-gpv-cases* [*consumes 1*, *case-names Out Cont*, *cases set*: *outs′-gpv*]:
assumes $x \in$ *outs′-gpv gpv*
obtains (*Out*) *generat* **where** *generat* $\in$ *set-spmf* (*the-gpv gpv*) $x \in$ *generat-outs generat*
| (*Cont*) *generat c input* **where** *generat* $\in$ *set-spmf* (*the-gpv gpv*) $c \in$ *generat-conts generat* $x \in$ *outs′-gpv* (*c input*)
using *assms* **by** *cases*(*auto simp add*: *in-set-spmf*)

lemma *outs′-gpvI* [*intro?*]:
shows *outs′-gpv-Out*: ⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $x \in$ *generat-outs generat* ⟧ $\Longrightarrow$ $x \in$ *outs′-gpv gpv*
and *outs′-gpv-Cont*: ⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $c \in$ *generat-conts generat*; $x \in$ *outs′-gpv* (*c input*) ⟧ $\Longrightarrow$ $x \in$ *outs′-gpv gpv*
by(*auto intro*: *gpv.set-sel simp add*: *in-set-spmf*)

lemma *results′-gpv-induct* [*consumes 1*, *case-names Pure Cont*, *induct set*: *results′-gpv*]:
assumes *x*: $x \in$ *results′-gpv gpv*
and *Pure*: ⋀*generat gpv*. ⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $x \in$ *generat-pures generat* ⟧ $\Longrightarrow$ *P gpv*
and *Cont*: ⋀*generat gpv c input*.
⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $c \in$ *generat-conts generat*; $x \in$ *results′-gpv* (*c input*); *P* (*c input*) ⟧ $\Longrightarrow$ *P gpv*
shows *P gpv*
using *x*
apply(*induction y≡x gpv*)
apply(*rule Pure*; *simp add*: *in-set-spmf*)
apply(*erule imageE*, *rule Cont*, *simp add*: *in-set-spmf*, *simp*, *simp*, *simp*)
.

lemma *results′-gpv-cases* [*consumes 1*, *case-names Pure Cont*, *cases set*: *results′-gpv*]:
assumes $x \in$ *results′-gpv gpv*
obtains (*Pure*) *generat* **where** *generat* $\in$ *set-spmf* (*the-gpv gpv*) $x \in$ *generat-pures generat*

| (*Cont*) *generat c input* **where** *generat* ∈ *set-spmf* (*the-gpv gpv*) *c* ∈ *generat-conts generat x* ∈ *results′-gpv* (*c input*)
using *assms* **by** *cases*(*auto simp add*: *in-set-spmf*)

lemma *results′-gpvI* [*intro?*]:
shows *results′-gpv-Pure*: ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *x* ∈ *generat-pures generat* ⟧ ⟹ *x* ∈ *results′-gpv gpv*
and *results′-gpv-Cont*: ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *c* ∈ *generat-conts generat*; *x* ∈ *results′-gpv* (*c input*) ⟧ ⟹ *x* ∈ *results′-gpv gpv*
by(*auto intro*: *gpv.set-sel simp add*: *in-set-spmf*)

lemma *left-unique-rel-gpv* [*transfer-rule*]:
⟦ *left-unique A*; *left-unique B* ⟧ ⟹ *left-unique* (*rel-gpv A B*)
unfolding *left-unique-alt-def gpv.rel-conversep*[*symmetric*] *gpv.rel-compp*[*symmetric*]
by(*subst gpv.rel-eq*[*symmetric*])(*rule gpv.rel-mono*)

lemma *right-unique-rel-gpv* [*transfer-rule*]:
⟦ *right-unique A*; *right-unique B* ⟧ ⟹ *right-unique* (*rel-gpv A B*)
unfolding *right-unique-alt-def gpv.rel-conversep*[*symmetric*] *gpv.rel-compp*[*symmetric*]
by(*subst gpv.rel-eq*[*symmetric*])(*rule gpv.rel-mono*)

lemma *bi-unique-rel-gpv* [*transfer-rule*]:
⟦ *bi-unique A*; *bi-unique B* ⟧ ⟹ *bi-unique* (*rel-gpv A B*)
unfolding *bi-unique-alt-def* **by**(*simp add*: *left-unique-rel-gpv right-unique-rel-gpv*)

lemma *left-total-rel-gpv* [*transfer-rule*]:
⟦ *left-total A*; *left-total B* ⟧ ⟹ *left-total* (*rel-gpv A B*)
unfolding *left-total-alt-def gpv.rel-conversep*[*symmetric*] *gpv.rel-compp*[*symmetric*]
by(*subst gpv.rel-eq*[*symmetric*])(*rule gpv.rel-mono*)

lemma *right-total-rel-gpv* [*transfer-rule*]:
⟦ *right-total A*; *right-total B* ⟧ ⟹ *right-total* (*rel-gpv A B*)
unfolding *right-total-alt-def gpv.rel-conversep*[*symmetric*] *gpv.rel-compp*[*symmetric*]
by(*subst gpv.rel-eq*[*symmetric*])(*rule gpv.rel-mono*)

lemma *bi-total-rel-gpv* [*transfer-rule*]: ⟦ *bi-total A*; *bi-total B* ⟧ ⟹ *bi-total* (*rel-gpv A B*)
unfolding *bi-total-alt-def* **by**(*simp add*: *left-total-rel-gpv right-total-rel-gpv*)

declare *gpv.map-transfer*[*transfer-rule*]

lemma *if-distrib-map-gpv* [*if-distribs*]:
map-gpv f g (*if b then gpv else gpv′*) = (*if b then map-gpv f g gpv else map-gpv f g gpv′*)
by *simp*

lemma *gpv-pred-mono-strong*:
⟦ *pred-gpv P Q x*; ⋀*a*. ⟦ *a* ∈ *results′-gpv x*; *P a* ⟧ ⟹ *P′ a*; ⋀*b*. ⟦ *b* ∈ *outs′-gpv x*; *Q b* ⟧ ⟹ *Q′ b* ⟧ ⟹ *pred-gpv P′ Q′ x*

by(*simp add*: *pred-gpv-def*)

lemma *pred-gpv-top* [*simp*]:
pred-gpv (λ-. *True*) (λ-. *True*) = (λ-. *True*)
by(*simp add*: *pred-gpv-def*)

lemma *pred-gpv-conj* [*simp*]:
shows *pred-gpv-conj1*: $\bigwedge P\ Q\ R$. *pred-gpv* (λ*x*. *P x* ∧ *Q x*) *R* = (λ*x*. *pred-gpv P R x* ∧ *pred-gpv Q R x*)
and *pred-gpv-conj2*: $\bigwedge P\ Q\ R$. *pred-gpv P* (λ*x*. *Q x* ∧ *R x*) = (λ*x*. *pred-gpv P Q x* ∧ *pred-gpv P R x*)
by(*auto simp add*: *pred-gpv-def*)

lemma *rel-gpv-restrict-relp1I* [*intro?*]:
⟦ *rel-gpv R R′ x y*; *pred-gpv P P′ x*; *pred-gpv Q Q′ y* ⟧ ⟹ *rel-gpv* (*R* ↿ *P* ⊗ *Q*) (*R′* ↿ *P′* ⊗ *Q′*) *x y*
by(*erule gpv.rel-mono-strong*)(*simp-all add*: *pred-gpv-def*)

lemma *rel-gpv-restrict-relpE* [*elim?*]:
assumes *rel-gpv* (*R* ↿ *P* ⊗ *Q*) (*R′* ↿ *P′* ⊗ *Q′*) *x y*
obtains *rel-gpv R R′ x y pred-gpv P P′ x pred-gpv Q Q′ y*
proof
show *rel-gpv R R′ x y* **using** *assms* **by**(*auto elim*!: *gpv.rel-mono-strong*)
have *pred-gpv* (*Domainp* (*R* ↿ *P* ⊗ *Q*)) (*Domainp* (*R′* ↿ *P′* ⊗ *Q′*)) *x* **using** *assms* **by**(*fold gpv.Domainp-rel*) *blast*
then show *pred-gpv P P′ x* **by**(*rule gpv-pred-mono-strong*)(*blast dest*!: *restrict-relp-DomainpD*)+
have *pred-gpv* (*Domainp* $(R \upharpoonleft P \otimes Q)^{-1-1}$) (*Domainp* $(R' \upharpoonleft P' \otimes Q')^{-1-1}$) *y* **using** *assms*
by(*fold gpv.Domainp-rel*)(*auto simp only*: *gpv.rel-conversep Domainp-conversep*)
then show *pred-gpv Q Q′ y* **by**(*rule gpv-pred-mono-strong*)(*auto dest*!: *restrict-relp-DomainpD*)
qed

lemma *gpv-pred-map* [*simp*]: *pred-gpv P Q* (*map-gpv f g gpv*) = *pred-gpv* (*P* ∘ *f*) (*Q* ∘ *g*) *gpv*
by(*simp add*: *pred-gpv-def*)

4.3 Generalised mapper and relator

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

primcorec *map-gpv′* :: (*′a* ⇒ *′b*) ⇒ (*′out* ⇒ *′out′*) ⇒ (*′ret′* ⇒ *′ret*) ⇒ (*′a*, *′out*, *′ret*) *gpv* ⇒ (*′b*, *′out′*, *′ret′*) *gpv*
where
map-gpv′ f g h gpv =
GPV (*map-spmf* (*map-generat f g* ((∘) (*map-gpv′ f g h*))) (*map-spmf* (*map-generat id id* (*map-fun h id*)) (*the-gpv gpv*)))

declare *map-gpv′.sel* [*simp del*]

lemma *map-gpv′-sel* [*simp*]:
the-gpv (*map-gpv′ f g h gpv*) = *map-spmf* (*map-generat f g* (*h* −−−> *map-gpv′ f g h*)) (*the-gpv gpv*)
by(*simp add*: *map-gpv′.sel spmf.map-comp o-def generat.map-comp map-fun-def*[*abs-def*])

lemma *map-gpv′-GPV* [*simp*]:
map-gpv′ f g h (*GPV p*) = *GPV* (*map-spmf* (*map-generat f g* (*h* −−−> *map-gpv′ f g h*)) *p*)
by(*rule gpv.expand*) *simp*

lemma *map-gpv′-id*: *map-gpv′ id id id* = *id*
apply(*rule ext*)
apply(*coinduction*)
apply(*auto simp add*: *spmf-rel-map generat.rel-map rel-fun-def intro*!: *rel-spmf-reflI generat.rel-refl*)
done

lemma *map-gpv′-comp*: *map-gpv′ f g h* (*map-gpv′ f′ g′ h′ gpv*) = *map-gpv′* (*f* ∘ *f′*) (*g* ∘ *g′*) (*h′* ∘ *h*) *gpv*
by(*coinduction arbitrary*: *gpv*)(*auto simp add*: *spmf.map-comp spmf-rel-map generat.rel-map rel-fun-def intro*!: *rel-spmf-reflI generat.rel-refl*)

functor *gpv*: *map-gpv′* **by**(*simp-all add*: *map-gpv′-comp map-gpv′-id o-def*)

lemma *map-gpv-conv-map-gpv′*: *map-gpv f g* = *map-gpv′ f g id*
apply(*rule ext*)
apply(*coinduction*)
apply(*auto simp add*: *gpv.map-sel spmf-rel-map generat.rel-map rel-fun-def intro*!: *generat.rel-refl-strong rel-spmf-reflI*)
done

coinductive *rel-gpv′′* :: (*′a* ⇒ *′b* ⇒ *bool*) ⇒ (*′out* ⇒ *′out′* ⇒ *bool*) ⇒ (*′ret* ⇒ *′ret′* ⇒ *bool*) ⇒ (*′a*, *′out*, *′ret*) *gpv* ⇒ (*′b*, *′out′*, *′ret′*) *gpv* ⇒ *bool*
for *A C R*
where
rel-spmf (*rel-generat A C* (*R* ===> *rel-gpv′′ A C R*)) (*the-gpv gpv*) (*the-gpv gpv′*)
⟹ *rel-gpv′′ A C R gpv gpv′*

lemma *rel-gpv′′-coinduct* [*consumes 1*, *case-names rel-gpv′′*, *coinduct pred*: *rel-gpv′′*]:
⟦*X gpv gpv′*;
⋀*gpv gpv′*. *X gpv gpv′*
⟹ *rel-spmf* (*rel-generat A C* (*R* ===> (λ*gpv gpv′*. *X gpv gpv′* ∨ *rel-gpv′′ A C R gpv gpv′*)))
(*the-gpv gpv*) (*the-gpv gpv′*) ⟧
⟹ *rel-gpv′′ A C R gpv gpv′*
by(*erule rel-gpv′′.coinduct*) *blast*

lemma *rel-gpv′′D*:

rel-gpv′′ A C R gpv gpv′
⟹ *rel-spmf* (*rel-generat A C* (*R* ===> *rel-gpv′′ A C R*)) (*the-gpv gpv*) (*the-gpv gpv′*)
by(*simp add*: *rel-gpv′′.simps*)

lemma *rel-gpv′′-GPV* [*simp*]:
rel-gpv′′ A C R (*GPV p*) (*GPV q*) ⟷
rel-spmf (*rel-generat A C* (*R* ===> *rel-gpv′′ A C R*)) *p q*
by(*simp add*: *rel-gpv′′.simps*)

lemma *rel-gpv-conv-rel-gpv′′*: *rel-gpv A C* = *rel-gpv′′ A C* (=)
proof(*rule ext iffI*)+
show *rel-gpv A C gpv gpv′* **if** *rel-gpv′′ A C* (=) *gpv gpv′* **for** *gpv* :: (*′a*, *′b*, *′c*) *gpv* **and** *gpv′* :: (*′d*, *′e*, *′c*) *gpv*
using *that* **by**(*coinduct*)(*blast dest*: *rel-gpv′′D*)
show *rel-gpv′′ A C* (=) *gpv gpv′* **if** *rel-gpv A C gpv gpv′* **for** *gpv* :: (*′a*, *′b*, *′c*) *gpv* **and** *gpv′* :: (*′d*, *′e*, *′c*) *gpv*
using *that* **by**(*coinduct*)(*auto elim*!: *gpv.rel-cases rel-spmf-mono generat.rel-mono-strong rel-fun-mono*)
qed

lemma *rel-gpv′′-eq* :
rel-gpv′′ (=) (=) (=) = (=)
by(*simp add*: *rel-gpv-conv-rel-gpv′′*[*symmetric*] *gpv.rel-eq*)

lemma *rel-gpv′′-mono*:
assumes $A \le A'$ $C \le C'$ $R' \le R$
shows *rel-gpv′′ A C R* ≤ *rel-gpv′′ A′ C′ R′*
proof
show *rel-gpv′′ A′ C′ R′ gpv gpv′* **if** *rel-gpv′′ A C R gpv gpv′* **for** *gpv gpv′* **using** *that*
by(*coinduct*)(*auto dest*: *rel-gpv′′D elim*!: *rel-spmf-mono generat.rel-mono-strong rel-fun-mono intro*: *assms*[*THEN predicate2D*])
qed

lemma *rel-gpv′′-conversep*: *rel-gpv′′* A^{-1-1} C^{-1-1} R^{-1-1} = (*rel-gpv′′ A C R*)$^{-1-1}$
proof(*intro ext iffI*; *simp*)
show *rel-gpv′′ A C R gpv gpv′* **if** *rel-gpv′′* A^{-1-1} C^{-1-1} R^{-1-1} *gpv′ gpv*
for *A* :: *′a1* ⇒ *′a2* ⇒ *bool* **and** *C* :: *′c1* ⇒ *′c2* ⇒ *bool* **and** *R* :: *′r1* ⇒ *′r2* ⇒ *bool* **and** *gpv gpv′*
using *that* **apply**(*coinduct*)
apply(*drule rel-gpv′′D*)
apply(*rewrite in* ⌑ *conversep-iff*[*symmetric*])
apply(*subst spmf-rel-conversep*[*symmetric*])
apply(*erule rel-spmf-mono*)
apply(*subst generat.rel-conversep*[*symmetric*])
apply(*erule generat.rel-mono-strong*)
apply(*auto simp add*: *rel-fun-def conversep-iff*[*abs-def*])
done

from *this*[*of* A^{-1-1} C^{-1-1} R^{-1-1}]
show *rel-gpv''* A^{-1-1} C^{-1-1} R^{-1-1} *gpv' gpv* **if** *rel-gpv'' A C R gpv gpv'* **for** *gpv gpv'* **using** *that* **by** *simp*
qed

lemma *rel-gpv''-pos-distr*:
rel-gpv'' A C R OO rel-gpv'' A' C' R' ≤ *rel-gpv''* (*A OO A'*) (*C OO C'*) (*R OO R'*)
proof(*rule predicate2I*; *erule relcomppE*)
show *rel-gpv''* (*A OO A'*) (*C OO C'*) (*R OO R'*) *gpv gpv''*
if *rel-gpv'' A C R gpv gpv' rel-gpv'' A' C' R' gpv' gpv''*
for *gpv gpv' gpv''* **using** *that*
apply(*coinduction arbitrary*: *gpv gpv' gpv''*)
apply(*drule rel-gpv''D*)+
apply(*drule* (*1*) *rel-spmf-pos-distr*[*THEN predicate2D*, *OF relcomppI*])
apply(*erule spmf-rel-mono-strong*)
apply(*subst* (*asm*) *generat.rel-compp*[*symmetric*])
apply(*erule generat.rel-mono-strong*, *assumption*, *assumption*)
apply(*drule pos-fun-distr*[*THEN predicate2D*])
apply(*auto simp add*: *rel-fun-def*)
done
qed

lemma *left-unique-rel-gpv''*:
⟦ *left-unique A*; *left-unique C*; *left-total R* ⟧ ⟹ *left-unique* (*rel-gpv'' A C R*)
unfolding *left-unique-alt-def left-total-alt-def rel-gpv''-conversep*[*symmetric*]
apply(*subst rel-gpv''-eq*[*symmetric*])
apply(*rule order-trans*[*OF rel-gpv''-pos-distr*])
apply(*erule* (*2*) *rel-gpv''-mono*)
done

lemma *right-unique-rel-gpv''*:
⟦ *right-unique A*; *right-unique C*; *right-total R* ⟧ ⟹ *right-unique* (*rel-gpv'' A C R*)
unfolding *right-unique-alt-def right-total-alt-def rel-gpv''-conversep*[*symmetric*]
apply(*subst rel-gpv''-eq*[*symmetric*])
apply(*rule order-trans*[*OF rel-gpv''-pos-distr*])
apply(*erule* (*2*) *rel-gpv''-mono*)
done

lemma *bi-unique-rel-gpv''* [*transfer-rule*]:
⟦ *bi-unique A*; *bi-unique C*; *bi-total R* ⟧ ⟹ *bi-unique* (*rel-gpv'' A C R*)
unfolding *bi-unique-alt-def bi-total-alt-def* **by**(*blast intro*: *left-unique-rel-gpv'' right-unique-rel-gpv''*)

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

show *?rhs* **if** *?lhs* **using** *that*
apply(*coinduction arbitrary*: *gpv gpv′*)
apply(*drule rel-gpv″D*)
apply(*simp add*: *gpv.map-sel spmf-rel-map*)
apply(*erule rel-spmf-mono*)
by(*auto simp add*: *generat.rel-map rel-fun-comp elim*!: *generat.rel-mono-strong rel-fun-mono*)
show *?lhs* **if** *?rhs* **using** *that*
apply(*coinduction arbitrary*: *gpv gpv′*)
apply(*drule rel-gpv″D*)
apply(*simp add*: *gpv.map-sel spmf-rel-map*)
apply(*erule rel-spmf-mono*)
by(*auto simp add*: *generat.rel-map rel-fun-comp elim*!: *generat.rel-mono-strong rel-fun-mono*)
qed

lemma *rel-gpv″-map-gpv2*:
rel-gpv″ A C R gpv (*map-gpv f g gpv′*) = *rel-gpv″* (*λa b. A a* (*f b*)) (*λc d. C c* (*g d*)) *R gpv gpv′*
using *rel-gpv″-map-gpv1*[*of conversep A conversep C conversep R f g gpv′ gpv*]
apply(*rewrite* **in** ⌑ = - *conversep-iff*[*symmetric*])
apply(*rewrite* **in** - = ⌑ *conversep-iff*[*symmetric*])
apply(*simp only*: *rel-gpv″-conversep*)
apply(*simp only*: *rel-gpv″-conversep*[*symmetric*])
apply(*simp only*: *conversep-iff*[*abs-def*])
done

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

lemma *rel-gpv″-map-gpv′* [*simp*]:
shows ⋀*f g h gpv. NO-MATCH id f* ∨ *NO-MATCH id g*
⟹ *rel-gpv″ A C R* (*map-gpv′ f g h gpv*) = *rel-gpv″* (*λa. A* (*f a*)) (*λc. C* (*g c*)) *R* (*map-gpv′ id id h gpv*)
and ⋀*f g h gpv gpv′. NO-MATCH id f* ∨ *NO-MATCH id g*
⟹ *rel-gpv″ A C R gpv* (*map-gpv′ f g h gpv′*) = *rel-gpv″* (*λa b. A a* (*f b*)) (*λc d. C c* (*g d*)) *R gpv* (*map-gpv′ id id h gpv′*)
proof (*goal-cases*)
case (*1 f g h gpv*)
then show *?case* **using** *map-gpv′-comp*[*of f g id id id h gpv*, *symmetric*] **by**(*simp add*: *rel-gpv″-map-gpv*[*unfolded map-gpv-conv-map-gpv′*])
next
case (*2 f g h gpv gpv′*)
then show *?case* **using** *map-gpv′-comp*[*of f g id id id h gpv′*, *symmetric*] **by**(*simp add*: *rel-gpv″-map-gpv*[*unfolded map-gpv-conv-map-gpv′*])
qed

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

definition *rel-witness-gpv* :: (*′a* ⇒ *′d* ⇒ *bool*) ⇒ (*′b* ⇒ *′e* ⇒ *bool*) ⇒ (*′c* ⇒ *′g* ⇒

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

lemma *rel-witness-gpv-sel* [*simp*]:
the-gpv (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) =
map-spmf (*map-generat id id* (λ(*rpv*, *rpv'*). (*rel-witness-gpv A C R R'* ∘ *rel-witness-fun R R'* (*rpv*, *rpv'*))) ∘ *rel-witness-generat*)
(*rel-witness-spmf* (*rel-generat A C* (*rel-fun* (*R OO R'*) (*rel-gpv'' A C* (*R OO R'*)))) (*the-gpv gpv*, *the-gpv gpv'*))
unfolding *rel-witness-gpv-def*
by(*auto simp add*: *spmf.map-comp generat.map-comp o-def intro*!: *map-spmf-cong generat.map-cong*)

lemma assumes *rel-gpv'' A C* (*R OO R'*) *gpv gpv'*
and *R*: *left-unique R right-total R*
and *R'*: *right-unique R' left-total R'*
shows *rel-witness-gpv1*: *rel-gpv''* (λ*a* (*a'*, *b*). *a* = *a'* ∧ *A a' b*) (λ*c* (*c'*, *d*). *c* = *c'* ∧ *C c' d*) *R gpv* (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) (**is** *?thesis1*)
and *rel-witness-gpv2*: *rel-gpv''* (λ(*a*, *b'*) *b*. *b* = *b'* ∧ *A a b'*) (λ(*c*, *d'*) *d*. *d* = *d'* ∧ *C c d'*) *R'* (*rel-witness-gpv A C R R'* (*gpv*, *gpv'*)) *gpv'* (**is** *?thesis2*)
proof −
show *?thesis1* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *gpv gpv'*)
case *rel-gpv''*
from *this*[*THEN rel-gpv''D*] **show** *?case*
by(*auto simp add*: *spmf-rel-map generat.rel-map rel-fun-comp elim*!: *rel-fun-mono*[*OF rel-witness-fun1*[*OF - R R'*]]
rel-spmf-mono[*OF rel-witness-spmf1*] *generat.rel-mono*[*THEN predicate2D*, *rotated −1*, *OF rel-witness-generat1*])
qed
show *?thesis2* **using** *assms*(*1*)
proof(*coinduction arbitrary*: *gpv gpv'*)
case *rel-gpv''*
from *this*[*THEN rel-gpv''D*] **show** *?case*
by(*simp add*: *spmf-rel-map*)
(*erule rel-spmf-mono*[*OF rel-witness-spmf2*]
, *auto simp add*: *generat.rel-map rel-fun-comp elim*!: *rel-fun-mono*[*OF rel-witness-fun2*[*OF - R R'*]]
generat.rel-mono[*THEN predicate2D*, *rotated −1*, *OF rel-witness-generat2*])
qed
qed

lemma *rel-gpv''-neg-distr*:
assumes *R*: *left-unique R right-total R*

and R': *right-unique* R' *left-total* R'
shows *rel-gpv''* (A *OO* A') (C *OO* C') (R *OO* R') $\leq$ *rel-gpv''* A C R *OO* *rel-gpv''* A' C' R'
proof(*rule predicate2I relcomppI*)+
fix *gpv gpv''*
assume $*$: *rel-gpv''* (A *OO* A') (C *OO* C') (R *OO* R') *gpv gpv''*
let *?gpv'* = *map-gpv* (*relcompp-witness* A A') (*relcompp-witness* C C') (*rel-witness-gpv* (A *OO* A') (C *OO* C') R R' (*gpv*, *gpv''*))
show *rel-gpv''* A C R *gpv ?gpv'* **using** *rel-witness-gpv1*[*OF* $*$ R R'] **unfolding** *rel-gpv''-map-gpv*
by(*rule rel-gpv''-mono*[*THEN predicate2D*, *rotated* -1]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)
show *rel-gpv''* A' C' R' *?gpv' gpv''* **using** *rel-witness-gpv2*[*OF* $*$ R R'] **unfolding** *rel-gpv''-map-gpv*
by(*rule rel-gpv''-mono*[*THEN predicate2D*, *rotated* -1]; *clarify del*: *relcomppE elim*!: *relcompp-witness*)
qed

lemma *rel-gpv''-mono'* [*mono*]:
assumes $\bigwedge x\ y.\ A\ x\ y \longrightarrow A'\ x\ y$
and $\bigwedge x\ y.\ C\ x\ y \longrightarrow C'\ x\ y$
and $\bigwedge x\ y.\ R'\ x\ y \longrightarrow R\ x\ y$
shows *rel-gpv''* A C R *gpv gpv'* $\longrightarrow$ *rel-gpv''* A' C' R' *gpv gpv'*
using *rel-gpv''-mono*[*of* A A' C C' R' R] *assms* **by**(*blast*)

lemma *left-total-rel-gpv'*:
⟦ *left-total* A; *left-total* C; *left-unique* R; *right-total* R ⟧ $\Longrightarrow$ *left-total* (*rel-gpv''* A C R)
unfolding *left-unique-alt-def left-total-alt-def rel-gpv''-conversep*[*symmetric*]
apply(*subst rel-gpv''-eq*[*symmetric*])
apply(*rule order-trans*[*rotated*])
apply(*rule rel-gpv''-neg-distr*; *simp add*: *left-unique-alt-def*)
apply(*rule rel-gpv''-mono*; *assumption*)
done

lemma *right-total-rel-gpv'*:
⟦ *right-total* A; *right-total* C; *right-unique* R; *left-total* R ⟧ $\Longrightarrow$ *right-total* (*rel-gpv''* A C R)
unfolding *right-unique-alt-def right-total-alt-def rel-gpv''-conversep*[*symmetric*]
apply(*subst rel-gpv''-eq*[*symmetric*])
apply(*rule order-trans*[*rotated*])
apply(*rule rel-gpv''-neg-distr*; *simp add*: *right-unique-alt-def*)
apply(*rule rel-gpv''-mono*; *assumption*)
done

lemma *bi-total-rel-gpv'* [*transfer-rule*]:
⟦ *bi-total* A; *bi-total* C; *bi-unique* R; *bi-total* R ⟧ $\Longrightarrow$ *bi-total* (*rel-gpv''* A C R)
unfolding *bi-total-alt-def bi-unique-alt-def* **by**(*blast intro*: *left-total-rel-gpv' right-total-rel-gpv'*)

lemma *rel-fun-conversep-grp-grp*:
rel-fun (*conversep* (*BNF-Def.Grp UNIV f*)) (*BNF-Def.Grp B g*) = *BNF-Def.Grp* {*x*. (*x* ∘ *f*) ' *UNIV* ⊆ *B*} (*map-fun f g*)
unfolding *rel-fun-def Grp-def simp-thms fun-eq-iff conversep-iff* **by** *auto*

lemma *Quotient-gpv*:
assumes *Q1*: *Quotient R1 Abs1 Rep1 T1*
and *Q2*: *Quotient R2 Abs2 Rep2 T2*
and *Q3*: *Quotient R3 Abs3 Rep3 T3*
shows *Quotient* (*rel-gpv'' R1 R2 R3*) (*map-gpv' Abs1 Abs2 Rep3*) (*map-gpv' Rep1 Rep2 Abs3*) (*rel-gpv'' T1 T2 T3*)
(**is** *Quotient ?R ?abs ?rep ?T*)
unfolding *Quotient-alt-def2*
proof(*intro conjI strip iffI*; (*elim conjE exE*)?)
note [*simp*] = *spmf-rel-map generat.rel-map*
and [*elim*!] = *rel-spmf-mono generat.rel-mono-strong*
and [*rule del*] = *rel-funI* **and** [*intro*!] = *rel-funI*
have *Abs1* [*simp*]: *Abs1 x* = *y* **if** *T1 x y* **for** *x y* **using** *Q1 that* **by**(*simp add*: *Quotient-alt-def*)
have *Abs2* [*simp*]: *Abs2 x* = *y* **if** *T2 x y* **for** *x y* **using** *Q2 that* **by**(*simp add*: *Quotient-alt-def*)
have *Abs3* [*simp*]: *Abs3 x* = *y* **if** *T3 x y* **for** *x y* **using** *Q3 that* **by**(*simp add*: *Quotient-alt-def*)
have *Rep1*: *T1* (*Rep1 x*) *x* **for** *x* **using** *Q1* **by**(*simp add*: *Quotient-alt-def*)
have *Rep2*: *T2* (*Rep2 x*) *x* **for** *x* **using** *Q2* **by**(*simp add*: *Quotient-alt-def*)
have *Rep3*: *T3* (*Rep3 x*) *x* **for** *x* **using** *Q3* **by**(*simp add*: *Quotient-alt-def*)
have *T1*: *T1 x* (*Abs1 y*) **if** *R1 x y* **for** *x y* **using** *Q1 that* **by**(*simp add*: *Quotient-alt-def2*)
have *T2*: *T2 x* (*Abs2 y*) **if** *R2 x y* **for** *x y* **using** *Q2 that* **by**(*simp add*: *Quotient-alt-def2*)
have *T1'*: *T1 x* (*Abs1 y*) **if** *R1 y x* **for** *x y* **using** *Q1 that* **by**(*simp add*: *Quotient-alt-def2*)
have *T2'*: *T2 x* (*Abs2 y*) **if** *R2 y x* **for** *x y* **using** *Q2 that* **by**(*simp add*: *Quotient-alt-def2*)
have *R3*: *R3 x* (*Rep3 y*) **if** *T3 x y* **for** *x y* **using** *Q3 that* **by**(*simp add*: *Quotient-alt-def2 Abs3*[*OF Rep3*])
have *R3'*: *R3* (*Rep3 y*) *x* **if** *T3 x y* **for** *x y* **using** *Q3 that* **by**(*simp add*: *Quotient-alt-def2 Abs3*[*OF Rep3*])
have *r1*: *R1* = *T1 OO T1*$^{-1-1}$ **using** *Q1* **by**(*simp add*: *Quotient-alt-def4*)
have *r2*: *R2* = *T2 OO T2*$^{-1-1}$ **using** *Q2* **by**(*simp add*: *Quotient-alt-def4*)
have *r3*: *R3* = *T3 OO T3*$^{-1-1}$ **using** *Q3* **by**(*simp add*: *Quotient-alt-def4*)
show *abs*: *?abs gpv* = *gpv'* **if** *?T gpv gpv'* **for** *gpv gpv'* **using** *that*
by(*coinduction arbitrary*: *gpv gpv'*)(*drule rel-gpv''D*; *auto 4 4 intro*: *Rep3 dest*: *rel-funD*)
show *?T* (*?rep gpv*) *gpv* **for** *gpv*
by(*coinduction arbitrary*: *gpv*)(*auto simp add*: *Rep1 Rep2 intro*!: *rel-spmf-reflI generat.rel-refl-strong*)
show *?T gpv* (*?abs gpv'*) **if** *?R gpv gpv'* **for** *gpv gpv'* **using** *that*
by(*coinduction arbitrary*: *gpv gpv'*)(*drule rel-gpv''D*; *auto 4 3 simp add*: *T1 T2*

intro!: *R3 dest*: *rel-funD*)
show *?T gpv* (*?abs gpv'*) **if** *?R gpv' gpv* **for** *gpv gpv'*
proof −
from *that* **have** *rel-gpv'' R1*$^{-1-1}$ *R2*$^{-1-1}$ *R3*$^{-1-1}$ *gpv gpv'* **unfolding** *rel-gpv''-conversep* **by** *simp*
then show *?thesis*
by(*coinduction arbitrary*: *gpv gpv'*)(*drule rel-gpv''D*; *auto 4 3 simp add*: *T1' T2' intro*!: *R3' dest*: *rel-funD*)
qed
show *?R gpv gpv'* **if** *?T gpv* (*?abs gpv'*) *?T gpv'* (*?abs gpv*) **for** *gpv gpv'*
proof −
from *that*[*THEN abs*] **have** *?abs gpv' = ?abs gpv* **by** *simp*
with *that* **have** (*?T OO ?T*$^{-1-1}$) *gpv gpv'* **by**(*auto simp del*: *rel-gpv''-map-gpv'*)
hence *rel-gpv''* (*T1 OO T1*$^{-1-1}$) (*T2 OO T2*$^{-1-1}$) (*T3 OO T3*$^{-1-1}$) *gpv gpv'*
unfolding *rel-gpv''-conversep*[*symmetric*]
by(*rule rel-gpv''-pos-distr*[*THEN predicate2D*])
thus *?thesis* **by**(*simp add*: *r1 r2 r3*)
qed
qed

lemma *the-gpv-parametric'*:
(*rel-gpv'' A C R* ===> *rel-spmf* (*rel-generat A C* (*R* ===> *rel-gpv'' A C R*))) *the-gpv the-gpv*
by(*rule rel-funI*)(*auto elim*: *rel-gpv''.cases*)

lemma *GPV-parametric'*:
(*rel-spmf* (*rel-generat A C* (*R* ===> *rel-gpv'' A C R*)) ===> *rel-gpv'' A C R*) *GPV GPV*
by(*rule rel-funI*)(*auto*)

lemma *corec-gpv-parametric'*:
((*S* ===> *rel-spmf* (*rel-generat A C* (*R* ===> *rel-sum* (*rel-gpv'' A C R*) *S*))) ===> *S* ===> *rel-gpv'' A C R*)
corec-gpv corec-gpv
proof(*rule rel-funI*)+
fix *f g s1 s2*
assume *fg*: (*S* ===> *rel-spmf* (*rel-generat A C* (*R* ===> *rel-sum* (*rel-gpv'' A C R*) *S*))) *f g*
and *s*: *S s1 s2*
from *s* **show** *rel-gpv'' A C R* (*corec-gpv f s1*) (*corec-gpv g s2*)
apply(*coinduction arbitrary*: *s1 s2*)
apply(*drule fg*[*THEN rel-funD*])
apply(*simp add*: *spmf-rel-map*)
apply(*erule rel-spmf-mono*)
apply(*simp add*: *generat.rel-map*)
apply(*erule generat.rel-mono-strong*; *clarsimp simp add*: *o-def*)
apply(*rule rel-funI*)
apply(*drule* (*1*) *rel-funD*)
apply(*auto 4 3 elim*!: *rel-sum.cases*)

done
qed

lemma *map-gpv′-parametric* [*transfer-rule*]:
((*A* ===> *A′*) ===> (*C* ===> *C′*) ===> (*R′* ===> *R*) ===> *rel-gpv″ A C R* ===> *rel-gpv″ A′ C′ R′*) *map-gpv′ map-gpv′*
unfolding *map-gpv′-def*
supply *corec-gpv-parametric′*[*transfer-rule*] *the-gpv-parametric′*[*transfer-rule*]
by(*transfer-prover*)

lemma *map-gpv-parametric′*: ((*A* ===> *A′*) ===> (*C* ===> *C′*) ===> *rel-gpv″ A C R* ===> *rel-gpv″ A′ C′ R*) *map-gpv map-gpv*
unfolding *map-gpv-conv-map-gpv′*[*abs-def*] **by** *transfer-prover*

end

4.4 Simple, derived operations

primcorec *Done* :: *′a* ⇒ (*′a*, *′out*, *′in*) *gpv*
where *the-gpv* (*Done a*) = *return-spmf* (*Pure a*)

primcorec *Pause* :: *′out* ⇒ (*′in* ⇒ (*′a*, *′out*, *′in*) *gpv*) ⇒ (*′a*, *′out*, *′in*) *gpv*
where *the-gpv* (*Pause out c*) = *return-spmf* (*IO out c*)

primcorec *lift-spmf* :: *′a spmf* ⇒ (*′a*, *′out*, *′in*) *gpv*
where *the-gpv* (*lift-spmf p*) = *map-spmf Pure p*

definition *Fail* :: (*′a*, *′out*, *′in*) *gpv*
where *Fail* = *GPV* (*return-pmf None*)

definition *React* :: (*′in* ⇒ *′out* × (*′a*, *′out*, *′in*) *rpv*) ⇒ (*′a*, *′out*, *′in*) *rpv*
where *React f input* = *case-prod Pause* (*f input*)

definition *rFail* :: (*′a*, *′out*, *′in*) *rpv*
where *rFail* = (λ-. *Fail*)

lemma *Done-inject* [*simp*]: *Done x* = *Done y* ⟷ *x* = *y*
by(*simp add*: *Done.ctr*)

lemma *Pause-inject* [*simp*]: *Pause out c* = *Pause out′ c′* ⟷ *out* = *out′* ∧ *c* = *c′*
by(*simp add*: *Pause.ctr*)

lemma [*simp*]:
shows *Done-neq-Pause*: *Done x* ≠ *Pause out c*
and *Pause-neq-Done*: *Pause out c* ≠ *Done x*
by(*simp-all add*: *Done.ctr Pause.ctr*)

lemma *outs′-gpv-Done* [*simp*]: *outs′-gpv* (*Done x*) = {}
by(*auto elim*: *outs′-gpv-cases*)

lemma *results′-gpv-Done* [*simp*]: *results′-gpv* (*Done x*) = {*x*}
by(*auto intro*: *results′-gpvI elim*: *results′-gpv-cases*)

lemma *pred-gpv-Done* [*simp*]: *pred-gpv P Q* (*Done x*) = *P x*
by(*simp add*: *pred-gpv-def*)

lemma *outs′-gpv-Pause* [*simp*]: *outs′-gpv* (*Pause out c*) = *insert out* (⋃*input*. *outs′-gpv* (*c input*))
by(*auto 4 4 intro*: *outs′-gpvI elim*: *outs′-gpv-cases*)

lemma *results′-gpv-Pause* [*simp*]: *results′-gpv* (*Pause out rpv*) = *results′-rpv rpv*
by(*auto 4 4 intro*: *results′-gpvI elim*: *results′-gpv-cases*)

lemma *pred-gpv-Pause* [*simp*]: *pred-gpv P Q* (*Pause x c*) = (*Q x* ∧ *All* (*pred-gpv P Q* ∘ *c*))
by(*auto simp add*: *pred-gpv-def o-def*)

lemma *lift-spmf-return* [*simp*]: *lift-spmf* (*return-spmf x*) = *Done x*
by(*simp add*: *lift-spmf.ctr Done.ctr*)

lemma *lift-spmf-None* [*simp*]: *lift-spmf* (*return-pmf None*) = *Fail*
by(*rule gpv.expand*)(*simp add*: *Fail-def*)

lemma *the-gpv-lift-spmf* [*simp*]: *the-gpv* (*lift-spmf r*) = *map-spmf Pure r*
by(*simp*)

lemma *outs′-gpv-lift-spmf* [*simp*]: *outs′-gpv* (*lift-spmf p*) = {}
by(*auto 4 3 elim*: *outs′-gpv-cases*)

lemma *results′-gpv-lift-spmf* [*simp*]: *results′-gpv* (*lift-spmf p*) = *set-spmf p*
by(*auto 4 3 elim*: *results′-gpv-cases intro*: *results′-gpvI*)

lemma *pred-gpv-lift-spmf* [*simp*]: *pred-gpv P Q* (*lift-spmf p*) = *pred-spmf P p*
by(*simp add*: *pred-gpv-def pred-spmf-def*)

lemma *lift-spmf-inject* [*simp*]: *lift-spmf p* = *lift-spmf q* ⟷ *p* = *q*
by(*auto simp add*: *lift-spmf.code dest*!: *pmf.inj-map-strong*[*rotated*] *option.inj-map-strong*[*rotated*])

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

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

lemma [*simp*]:
shows *Fail-neq-Pause*: *Fail* ≠ *Pause out c*
and *Pause-neq-Fail*: *Pause out c* ≠ *Fail*
and *Fail-neq-Done*: *Fail* ≠ *Done x*

and *Done-neq-Fail*: *Done x* ≠ *Fail*
by(*simp-all add*: *Fail-def Pause.ctr Done.ctr*)

Add *unit* closure to circumvent SML value restriction

definition *Fail′* :: *unit* ⇒ (*′a*, *′out*, *′in*) *gpv*
where [*code del*]: *Fail′* - = *Fail*

lemma *Fail-code* [*code-unfold*]: *Fail* = *Fail′* ()
by(*simp add*: *Fail′-def*)

lemma *Fail′-code* [*code*]:
Fail′ x = *GPV* (*return-pmf None*)
by(*simp add*: *Fail′-def Fail-def*)

lemma *Fail-sel* [*simp*]:
the-gpv Fail = *return-pmf None*
by(*simp add*: *Fail-def*)

lemma *Fail-eq-GPV-iff* [*simp*]: *Fail* = *GPV f* ⟷ *f* = *return-pmf None*
by(*auto simp add*: *Fail-def*)

lemma *outs′-gpv-Fail* [*simp*]: *outs′-gpv Fail* = {}
by(*auto elim*: *outs′-gpv-cases*)

lemma *results′-gpv-Fail* [*simp*]: *results′-gpv Fail* = {}
by(*auto elim*: *results′-gpv-cases*)

lemma *pred-gpv-Fail* [*simp*]: *pred-gpv P Q Fail*
by(*simp add*: *pred-gpv-def*)

lemma *React-inject* [*iff*]: *React f* = *React f′* ⟷ *f* = *f′*
by(*auto simp add*: *React-def fun-eq-iff split-def intro*: *prod.expand*)

lemma *React-apply* [*simp*]: *f input* = (*out*, *c*) ⟹ *React f input* = *Pause out c*
by(*simp add*: *React-def*)

lemma *rFail-apply* [*simp*]: *rFail input* = *Fail*
by(*simp add*: *rFail-def*)

lemma [*simp*]:
shows *rFail-neq-React*: *rFail* ≠ *React f*
and *React-neq-rFail*: *React f* ≠ *rFail*
by(*simp-all add*: *React-def fun-eq-iff split-beta*)

lemma *rel-gpv-FailI* [*simp*]: *rel-gpv A C Fail Fail*
by(*subst gpv.rel-sel*) *simp*

lemma *rel-gpv-Done* [*iff*]: *rel-gpv A C* (*Done x*) (*Done y*) ⟷ *A x y*
by(*subst gpv.rel-sel*) *simp*

lemma *rel-gpv''-Done* [*iff*]: *rel-gpv'' A C R* (*Done x*) (*Done y*) ⟷ *A x y*
by(*subst rel-gpv''.simps*) *simp*

lemma *rel-gpv-Pause* [*iff*]:
rel-gpv A C (*Pause out c*) (*Pause out' c'*) ⟷ *C out out'* ∧ (∀ *x*. *rel-gpv A C* (*c x*) (*c' x*))
by(*subst gpv.rel-sel*)(*simp add*: *rel-fun-def*)

lemma *rel-gpv''-Pause* [*iff*]:
rel-gpv'' A C R (*Pause out c*) (*Pause out' c'*) ⟷ *C out out'* ∧ (∀ *x x'*. *R x x'* ⟶ *rel-gpv'' A C R* (*c x*) (*c' x'*))
by(*subst rel-gpv''.simps*)(*simp add*: *rel-fun-def*)

lemma *rel-gpv-lift-spmf* [*iff*]: *rel-gpv A C* (*lift-spmf p*) (*lift-spmf q*) ⟷ *rel-spmf A p q*
by(*subst gpv.rel-sel*)(*simp add*: *spmf-rel-map*)

lemma *rel-gpv''-lift-spmf* [*iff*]:
rel-gpv'' A C R (*lift-spmf p*) (*lift-spmf q*) ⟷ *rel-spmf A p q*
by(*subst rel-gpv''.simps*)(*simp add*: *spmf-rel-map*)

context includes *lifting-syntax* **begin**
lemmas *Fail-parametric* [*transfer-rule*] = *rel-gpv-FailI*

lemma *Fail-parametric'* [*simp*]: *rel-gpv'' A C R Fail Fail*
unfolding *Fail-def* **by** *simp*

lemma *Done-parametric* [*transfer-rule*]: (*A* ===> *rel-gpv A C*) *Done Done*
by(*rule rel-funI*) *simp*

lemma *Done-parametric'*: (*A* ===> *rel-gpv'' A C R*) *Done Done*
by(*rule rel-funI*) *simp*

lemma *Pause-parametric* [*transfer-rule*]:
(*C* ===> ((=) ===> *rel-gpv A C*) ===> *rel-gpv A C*) *Pause Pause*
by(*simp add*: *rel-fun-def*)

lemma *Pause-parametric'*:
(*C* ===> (*R* ===> *rel-gpv'' A C R*) ===> *rel-gpv'' A C R*) *Pause Pause*
by(*simp add*: *rel-fun-def*)

lemma *lift-spmf-parametric* [*transfer-rule*]:
(*rel-spmf A* ===> *rel-gpv A C*) *lift-spmf lift-spmf*
by(*simp add*: *rel-fun-def*)

lemma *lift-spmf-parametric'*:
(*rel-spmf A* ===> *rel-gpv'' A C R*) *lift-spmf lift-spmf*
by(*simp add*: *rel-fun-def*)

end

lemma *map-gpv-Done* [*simp*]: *map-gpv f g* (*Done x*) = *Done* (*f x*)
by(*simp add*: *Done.code*)

lemma *map-gpv'-Done* [*simp*]: *map-gpv' f g h* (*Done x*) = *Done* (*f x*)
by(*simp add*: *Done.code*)

lemma *map-gpv-Pause* [*simp*]: *map-gpv f g* (*Pause x c*) = *Pause* (*g x*) (*map-gpv f g* ∘ *c*)
by(*simp add*: *Pause.code*)

lemma *map-gpv'-Pause* [*simp*]: *map-gpv' f g h* (*Pause x c*) = *Pause* (*g x*) (*map-gpv' f g h* ∘ *c* ∘ *h*)
by(*simp add*: *Pause.code map-fun-def*)

lemma *map-gpv-Fail* [*simp*]: *map-gpv f g Fail* = *Fail*
by(*simp add*: *Fail-def*)

lemma *map-gpv'-Fail* [*simp*]: *map-gpv' f g h Fail* = *Fail*
by(*simp add*: *Fail-def*)

4.5 Monad structure

primcorec *bind-gpv* :: (*'a*, *'out*, *'in*) *gpv* ⇒ (*'a* ⇒ (*'b*, *'out*, *'in*) *gpv*) ⇒ (*'b*, *'out*, *'in*) *gpv*
where
```
  the-gpv (bind-gpv r f) =
   map-spmf (map-generat id id ((∘) (case-sum id (λr. bind-gpv r f))))
     (the-gpv r ≫=
      (case-generat
        (λx. map-spmf (map-generat id id ((∘) Inl)) (the-gpv (f x)))
        (λout c. return-spmf (IO out (λinput. Inr (c input))))))
```

declare *bind-gpv.sel* [*simp del*]

adhoc-overloading *Monad-Syntax.bind* ⇌ *bind-gpv*

lemma *bind-gpv-unfold* [*code*]:
```
  r ≫= f = GPV (
   do {
     generat ← the-gpv r;
     case generat of Pure x ⇒ the-gpv (f x)
       | IO out c ⇒ return-spmf (IO out (λinput. c input ≫= f))
   })
```
unfolding *bind-gpv-def*
apply(*rule gpv.expand*)
apply(*simp add*: *map-spmf-bind-spmf*)
apply(*rule arg-cong*[**where** *f*=*bind-spmf* (*the-gpv r*)])

apply(*auto split*: *generat.split simp add*: *map-spmf-bind-spmf fun-eq-iff spmf.map-comp o-def generat.map-comp id-def*[*symmetric*] *generat.map-id pmf.map-id option.map-id*)
done

lemma *bind-gpv-code-cong*: $f = f' \Longrightarrow$ *bind-gpv f g* = *bind-gpv f' g* **by** *simp*
setup ‹*Code-Simp.map-ss* (*Simplifier.add-cong* @{*thm bind-gpv-code-cong*})›

lemma *bind-gpv-sel*:
  the-gpv (*r* ≫= *f*) =
   do {
     generat ← *the-gpv r*;
     case generat of Pure x ⇒ *the-gpv* (*f x*)
       | *IO out c* ⇒ *return-spmf* (*IO out* (λ*input*. *bind-gpv* (*c input*) *f*))
   }
by(*subst bind-gpv-unfold*) *simp*

lemma *bind-gpv-sel'* [*simp*]:
  the-gpv (*r* ≫= *f*) =
   do {
     generat ← *the-gpv r*;
     if is-Pure generat then the-gpv (*f* (*result generat*))
     else return-spmf (*IO* (*output generat*) (λ*input*. *bind-gpv* (*continuation generat input*) *f*))
   }
unfolding *bind-gpv-sel*
by(*rule arg-cong*[**where** *f*=*bind-spmf* (*the-gpv r*)])(*simp add*: *fun-eq-iff split*: *generat.split*)

lemma *Done-bind-gpv* [*simp*]: *Done a* ≫= *f* = *f a*
by(*rule gpv.expand*)(*simp*)

lemma *bind-gpv-Done* [*simp*]: *f* ≫= *Done* = *f*
proof(*coinduction arbitrary*: *f rule*: *gpv.coinduct*)
  case (*Eq-gpv f*)
   have ∗: *the-gpv f* ≫= (*case-generat* (λ*x*. *return-spmf* (*Pure x*)) (λ*out c*. *return-spmf* (*IO out* (λ*input*. *Inr* (*c input*))))) =
         map-spmf (*map-generat id id* ((∘) *Inr*)) (*bind-spmf* (*the-gpv f*) *return-spmf*)
    unfolding *map-spmf-bind-spmf*
    by(*rule arg-cong2*[**where** *f*=*bind-spmf*])(*auto simp add*: *fun-eq-iff split*: *generat.split*)
  show *?case*
    by(*auto simp add*: ∗ *bind-gpv.simps pmf.rel-map option.rel-map*[*abs-def*] *generat.rel-map*[*abs-def*] *simp del*: *bind-gpv-sel'* *intro*!: *rel-generatI rel-spmf-reflI*)
qed

lemma *if-distrib-bind-gpv2* [*if-distribs*]:
  bind-gpv gpv (λ*y*. *if b then f y else g y*) = (*if b then bind-gpv gpv f else bind-gpv gpv g*)
by *simp*

lemma *lift-spmf-bind*: *lift-spmf r* ≫= *f* = *GPV* (*r* ≫= *the-gpv* ∘ *f*)
by(*coinduction arbitrary*: *r f rule*: *gpv.coinduct-strong*)(*auto simp add*: *bind-map-spmf o-def intro*: *rel-pmf-reflI rel-optionI rel-generatI*)

lemma *the-gpv-bind-gpv-lift-spmf* [*simp*]:
the-gpv (*bind-gpv* (*lift-spmf p*) *f*) = *bind-spmf p* (*the-gpv* ∘ *f*)
by(*simp add*: *bind-map-spmf o-def*)

lemma *lift-spmf-bind-spmf*: *lift-spmf* (*p* ≫= *f*) = *lift-spmf p* ≫= (λ*x*. *lift-spmf* (*f x*))
by(*rule gpv.expand*)(*simp add*: *lift-spmf-bind o-def map-spmf-bind-spmf*)

lemma *lift-bind-spmf*: *lift-spmf* (*bind-spmf p f*) = *bind-gpv* (*lift-spmf p*) (*lift-spmf* ∘ *f*)
by(*rule gpv.expand*)(*simp add*: *bind-map-spmf map-spmf-bind-spmf o-def*)

lemma *GPV-bind*:
GPV f ≫= *g* =
GPV (*f* ≫= (λ*generat*. *case generat of Pure x* ⇒ *the-gpv* (*g x*) | *IO out c* ⇒ *return-spmf* (*IO out* (λ*input*. *c input* ≫= *g*))))
by(*subst bind-gpv-unfold*) *simp*

lemma *GPV-bind'*:
GPV f ≫= *g* = *GPV* (*f* ≫= (λ*generat*. *if is-Pure generat then the-gpv* (*g* (*result generat*)) *else return-spmf* (*IO* (*output generat*) (λ*input*. *continuation generat input* ≫= *g*))))
unfolding *GPV-bind gpv.inject*
by(*rule arg-cong*[**where** *f*=*bind-spmf f*])(*simp add*: *fun-eq-iff split*: *generat.split*)

lemma *bind-gpv-assoc*:
fixes *f* :: ('*a*, '*out*, '*in*) *gpv*
shows (*f* ≫= *g*) ≫= *h* = *f* ≫= (λ*x*. *g x* ≫= *h*)
proof(*coinduction arbitrary*: *f g h rule*: *gpv.coinduct-strong*)
case (*Eq-gpv f g h*)
show *?case*
apply(*simp cong del*: *if-weak-cong*)
apply(*rule rel-spmf-bindI*[**where** *R*=(=)])
apply(*simp add*: *option.rel-eq pmf.rel-eq*)
apply(*fastforce intro*: *rel-pmf-return-pmfI rel-generatI rel-spmf-reflI*)
done
qed

lemma *map-gpv-bind-gpv*: *map-gpv f g* (*bind-gpv gpv h*) = *bind-gpv* (*map-gpv id g gpv*) (λ*x*. *map-gpv f g* (*h x*))
apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
apply(*simp add*: *bind-gpv.sel gpv.map-sel spmf-rel-map generat.rel-map o-def bind-map-spmf del*: *bind-gpv-sel'*)
apply(*rule rel-spmf-bind-reflI*)

apply(*auto simp add*: *spmf-rel-map generat.rel-map split*: *generat.split del*: *rel-funI intro*!: *rel-spmf-reflI generat.rel-refl rel-funI*)
done

lemma *map-gpv-id-bind-gpv*: *map-gpv f id* (*bind-gpv gpv g*) = *bind-gpv gpv* (*map-gpv f id* ∘ *g*)
by(*simp add*: *map-gpv-bind-gpv gpv.map-id o-def*)

lemma *map-gpv-conv-bind*:
  map-gpv f (λ*x*. *x*) *x* = *bind-gpv x* (λ*x*. *Done* (*f x*))
using *map-gpv-bind-gpv*[*of f* λ*x*. *x x Done*] **by**(*simp add*: *id-def*[*symmetric*] *gpv.map-id*)

lemma *bind-map-gpv*: *bind-gpv* (*map-gpv f id gpv*) *g* = *bind-gpv gpv* (*g* ∘ *f*)
by(*simp add*: *map-gpv-conv-bind id-def bind-gpv-assoc o-def*)

lemma *outs-bind-gpv*:
  outs′-gpv (*bind-gpv x f*) = *outs′-gpv x* ∪ (⋃*x* ∈ *results′-gpv x*. *outs′-gpv* (*f x*))
  (**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
  fix *out*
  assume *out* ∈ *?lhs*
  then show *out* ∈ *?rhs*
  proof(*induction g*≡*x* ⫽= *f arbitrary*: *x*)
    case (*Out generat*)
    then obtain *generat′* **where** ∗: *generat′* ∈ *set-spmf* (*the-gpv x*)
     and ∗∗: *generat* ∈ *set-spmf* (*if is-Pure generat′ then the-gpv* (*f* (*result generat′*))
                         else return-spmf (*IO* (*output generat′*) (λ*input*. *continuation generat′ input* ⫽= *f*)))
      by(*auto*)
    show *?case*
    proof(*cases is-Pure generat′*)
      case *True*
       then have *out* ∈ *outs′-gpv* (*f* (*result generat′*)) **using** *Out*(*2*) ∗∗ **by**(*auto intro*: *outs′-gpvI*)
      moreover have *result generat′* ∈ *results′-gpv x* **using** ∗ *True*
        by(*auto intro*: *results′-gpvI generat.set-sel*)
      ultimately show *?thesis* **by** *blast*
    next
      case *False*
       hence *out* ∈ *outs′-gpv x* **using** ∗ ∗∗ *Out*(*2*) **by**(*auto intro*: *outs′-gpvI generat.set-sel*)
      thus *?thesis* **by** *blast*
    qed
  next
    case (*Cont generat c input*)
    then obtain *generat′* **where** ∗: *generat′* ∈ *set-spmf* (*the-gpv x*)
     and ∗∗: *generat* ∈ *set-spmf* (*if is-Pure generat′ then the-gpv* (*f* (*generat.result generat′*))
                         else return-spmf (*IO* (*generat.output generat′*) (λ*input*.

continuation generat′ input ≫= *f*)))
  by(*auto*)
 show *?case*
 proof(*cases is-Pure generat′*)
  case *True*
  then have *out* ∈ *outs′-gpv* (*f* (*result generat′*)) **using** *Cont*(*2*−*3*) ∗∗ **by**(*auto intro*: *outs′-gpvI*)
  moreover have *result generat′* ∈ *results′-gpv x* **using** ∗ *True*
   by(*auto intro*: *results′-gpvI generat.set-sel*)
  ultimately show *?thesis* **by** *blast*
 next
  case *False*
  then have *generat*: *generat* = *IO* (*output generat′*) (λ*input*. *continuation generat′ input* ≫= *f*)
   using ∗∗ **by** *simp*
  with *Cont*(*2*) **have** *c input* = *continuation generat′ input* ≫= *f* **by** *auto*
   hence *out* ∈ *outs′-gpv* (*continuation generat′ input*) ∪ (⋃*x*∈*results′-gpv* (*continuation generat′ input*). *outs′-gpv* (*f x*))
   by(*rule Cont*)
  thus *?thesis*
  proof
   assume *out* ∈ *outs′-gpv* (*continuation generat′ input*)
    with ∗ ∗∗ *False* **have** *out* ∈ *outs′-gpv x* **by**(*auto intro*: *outs′-gpvI generat.set-sel*)
   thus *?thesis* **..**
  next
   assume *out* ∈ (⋃*x*∈*results′-gpv* (*continuation generat′ input*). *outs′-gpv* (*f x*))
   then obtain *y* **where** *y* ∈ *results′-gpv* (*continuation generat′ input*) *out* ∈ *outs′-gpv* (*f y*) **..**
   from ‹*y* ∈ -› ∗ ∗∗ *False* **have** *y* ∈ *results′-gpv x*
    by(*auto intro*: *results′-gpvI generat.set-sel*)
   with ‹*out* ∈ *outs′-gpv* (*f y*)› **show** *?thesis* **by** *blast*
  qed
 qed
 qed
next
 fix *out*
 assume *out* ∈ *?rhs*
 then show *out* ∈ *?lhs*
 proof
  assume *out* ∈ *outs′-gpv x*
  thus *?thesis*
  proof(*induction*)
   case (*Out generat gpv*)
   then show *?case*
    by(*cases generat*)(*fastforce intro*: *outs′-gpvI rev-bexI*)+
  next
   case (*Cont generat gpv gpv′*)

then show *?case*
by(*cases generat*)(*auto 4 4 intro*: *outs′-gpvI rev-bexI simp add*: *in-set-spmf set-pmf-bind-spmf simp del*: *set-bind-spmf*)
qed
next
assume *out* ∈ (⋃ *x*∈*results′-gpv x*. *outs′-gpv* (*f x*))
then obtain *y* **where** *y* ∈ *results′-gpv x out* ∈ *outs′-gpv* (*f y*) **..**
from ‹*y* ∈ -› **show** *?thesis*
proof(*induction*)
case (*Pure generat gpv*)
thus *?case* **using** ‹*out* ∈ *outs′-gpv* -›
by(*cases generat*)(*auto 4 5 intro*: *outs′-gpvI rev-bexI elim*: *outs′-gpv-cases*)
next
case (*Cont generat gpv gpv′*)
thus *?case*
by(*cases generat*)(*auto 4 4 simp add*: *in-set-spmf simp add*: *set-pmf-bind-spmf intro*: *outs′-gpvI rev-bexI simp del*: *set-bind-spmf*)
qed
qed
qed

lemma *bind-gpv-Fail* [*simp*]: *Fail* ≫= *f* = *Fail*
by(*subst bind-gpv-unfold*)(*simp add*: *Fail-def*)

lemma *bind-gpv-eq-Fail*:
bind-gpv gpv f = *Fail* ⟷ (∀ *x*∈*set-spmf* (*the-gpv gpv*). *is-Pure x*) ∧ (∀ *x*∈*results′-gpv gpv*. *f x* = *Fail*)
(**is** *?lhs* = *?rhs*)
proof(*intro iffI conjI strip*)
show *?lhs* **if** *?rhs* **using** *that*
by(*intro gpv.expand*)(*auto 4 4 simp add*: *bind-eq-return-pmf-None intro*: *results′-gpv-Pure generat.set-sel dest*: *bspec*)

assume *?lhs*
hence ∗: *the-gpv* (*bind-gpv gpv f*) = *return-pmf None* **by** *simp*
from ∗ **show** *is-Pure x* **if** *x* ∈ *set-spmf* (*the-gpv gpv*) **for** *x* **using** *that*
by(*simp add*: *bind-eq-return-pmf-None split*: *if-split-asm*)
show *f x* = *Fail* **if** *x* ∈ *results′-gpv gpv* **for** *x* **using** *that* ∗
by(*cases*)(*auto 4 3 simp add*: *bind-eq-return-pmf-None elim*!: *generat.set-cases intro*: *gpv.expand dest*: *bspec*)
qed

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

lemma *bind-gpv-parametric* [*transfer-rule*]:
(*rel-gpv A C* ===> (*A* ===> *rel-gpv B C*) ===> *rel-gpv B C*) *bind-gpv bind-gpv*
unfolding *bind-gpv-def* **by** *transfer-prover*

lemma *bind-gpv-parametric′*:
  (*rel-gpv″ A C R* ===> (*A* ===> *rel-gpv″ B C R*) ===> *rel-gpv″ B C R*)
bind-gpv bind-gpv
unfolding *bind-gpv-def* **supply** *corec-gpv-parametric′*[*transfer-rule*] *the-gpv-parametric′*[*transfer-rule*]
by(*transfer-prover*)

end

lemma *monad-gpv* [*locale-witness*]: *monad Done bind-gpv*
by(*unfold-locales*)(*simp-all add*: *bind-gpv-assoc*)

lemma *monad-fail-gpv* [*locale-witness*]: *monad-fail Done bind-gpv Fail*
by *unfold-locales auto*

lemma *rel-gpv-bindI*:
  ⟦ *rel-gpv A C gpv gpv′*; ⋀*x y*. *A x y* ⟹ *rel-gpv B C* (*f x*) (*g y*) ⟧
  ⟹ *rel-gpv B C* (*bind-gpv gpv f*) (*bind-gpv gpv′ g*)
by(*fact bind-gpv-parametric*[*THEN rel-funD*, *THEN rel-funD*, *OF* - *rel-funI*])

lemma *bind-gpv-cong*:
  ⟦ *gpv* = *gpv′*; ⋀*x*. *x* ∈ *results′-gpv gpv′* ⟹ *f x* = *g x* ⟧ ⟹ *bind-gpv gpv f* =
bind-gpv gpv′ g
apply(*subst gpv.rel-eq*[*symmetric*])
apply(*rule rel-gpv-bindI*[**where** *A*=*eq-onp* (λ*x*. *x* ∈ *results′-gpv gpv′*)])
 apply(*subst* (*asm*) *gpv.rel-eq*[*symmetric*])
 apply(*erule gpv.rel-mono-strong*)
  apply(*simp add*: *eq-onp-def*)
 apply *simp*
apply(*clarsimp simp add*: *gpv.rel-eq eq-onp-def*)
done

definition *bind-rpv* :: (′*a*, ′*in*, ′*out*) *rpv* ⇒ (′*a* ⇒ (′*b*, ′*in*, ′*out*) *gpv*) ⇒ (′*b*, ′*in*,
′*out*) *rpv*
where *bind-rpv rpv f* = (λ*input*. *bind-gpv* (*rpv input*) *f*)

lemma *bind-rpv-apply* [*simp*]: *bind-rpv rpv f input* = *bind-gpv* (*rpv input*) *f*
by(*simp add*: *bind-rpv-def fun-eq-iff*)

adhoc-overloading *Monad-Syntax.bind* ⇌ *bind-rpv*

lemma *bind-rpv-code-cong*: *rpv* = *rpv′* ⟹ *bind-rpv rpv f* = *bind-rpv rpv′ f* **by**
simp
setup ‹*Code-Simp.map-ss* (*Simplifier.add-cong* @{*thm bind-rpv-code-cong*})›

lemma *bind-rpv-rDone* [*simp*]: *bind-rpv rpv Done* = *rpv*
by(*simp add*: *bind-rpv-def*)

lemma *bind-gpv-Pause* [*simp*]: *bind-gpv* (*Pause out rpv*) *f* = *Pause out* (*bind-rpv*
rpv f)

by(*rule gpv.expand*)(*simp add*: *fun-eq-iff*)

lemma *bind-rpv-React* [*simp*]: *bind-rpv* (*React f*) *g* = *React* (*apsnd* (*λrpv. bind-rpv rpv g*) ∘ *f*)
by(*simp add*: *React-def split-beta fun-eq-iff*)

lemma *bind-rpv-assoc*: *bind-rpv* (*bind-rpv rpv f*) *g* = *bind-rpv rpv* ((*λgpv. bind-gpv gpv g*) ∘ *f*)
by(*simp add*: *fun-eq-iff bind-gpv-assoc o-def*)

lemma *bind-rpv-Done* [*simp*]: *bind-rpv Done f* = *f*
by(*simp add*: *bind-rpv-def*)

lemma *results′-rpv-Done* [*simp*]: *results′-rpv Done* = *UNIV*
by(*auto simp add*: *results′-rpv-def*)

4.6 Embedding *′a spmf* as a monad

lemma *neg-fun-distr3*:
  includes *lifting-syntax*
  assumes *1*: *left-unique R right-total R*
  assumes *2*: *right-unique S left-total S*
  shows (*R OO R′* ===> *S OO S′*) ≤ ((*R* ===> *S*) *OO* (*R′* ===> *S′*))
using *functional-relation*[*OF 2*] *functional-converse-relation*[*OF 1*]
unfolding *rel-fun-def OO-def*
apply *clarify*
apply (*subst all-comm*)
apply (*subst all-conj-distrib*[*symmetric*])
apply (*intro choice*)
by *metis*

locale *spmf-to-gpv* **begin**

The lifting package cannot handle free term variables in the merging of transfer rules, so for the embedding we define a specialised relator *rel-gpv′* which acts only on the returned values.

definition *rel-gpv′* :: (*′a* ⇒ *′b* ⇒ *bool*) ⇒ (*′a*, *′out*, *′in*) *gpv* ⇒ (*′b*, *′out*, *′in*) *gpv* ⇒ *bool*
where *rel-gpv′ A* = *rel-gpv A* (=)

lemma *rel-gpv′-eq* [*relator-eq*]: *rel-gpv′* (=) = (=)
unfolding *rel-gpv′-def gpv.rel-eq* **..**

lemma *rel-gpv′-mono* [*relator-mono*]: *A* ≤ *B* ⟹ *rel-gpv′ A* ≤ *rel-gpv′ B*
unfolding *rel-gpv′-def* **by**(*rule gpv.rel-mono*; *simp*)

lemma *rel-gpv′-distr* [*relator-distr*]: *rel-gpv′ A OO rel-gpv′ B* = *rel-gpv′* (*A OO B*)
unfolding *rel-gpv′-def* **by** (*metis OO-eq gpv.rel-compp*)

lemma *left-unique-rel-gpv′* [*transfer-rule*]: *left-unique A* ⟹ *left-unique* (*rel-gpv′ A*)
unfolding *rel-gpv′-def* **by**(*simp add*: *left-unique-rel-gpv left-unique-eq*)

lemma *right-unique-rel-gpv′* [*transfer-rule*]: *right-unique A* ⟹ *right-unique* (*rel-gpv′ A*)
unfolding *rel-gpv′-def* **by**(*simp add*: *right-unique-rel-gpv right-unique-eq*)

lemma *bi-unique-rel-gpv′* [*transfer-rule*]: *bi-unique A* ⟹ *bi-unique* (*rel-gpv′ A*)
unfolding *rel-gpv′-def* **by**(*simp add*: *bi-unique-rel-gpv bi-unique-eq*)

lemma *left-total-rel-gpv′* [*transfer-rule*]: *left-total A* ⟹ *left-total* (*rel-gpv′ A*)
unfolding *rel-gpv′-def* **by**(*simp add*: *left-total-rel-gpv left-total-eq*)

lemma *right-total-rel-gpv′* [*transfer-rule*]: *right-total A* ⟹ *right-total* (*rel-gpv′ A*)
unfolding *rel-gpv′-def* **by**(*simp add*: *right-total-rel-gpv right-total-eq*)

lemma *bi-total-rel-gpv′* [*transfer-rule*]: *bi-total A* ⟹ *bi-total* (*rel-gpv′ A*)
unfolding *rel-gpv′-def* **by**(*simp add*: *bi-total-rel-gpv bi-total-eq*)

We cannot use *setup-lifting* because (*′a*, *′out*, *′in*) *gpv* contains type variables which do not appear in *′a spmf*.

definition *cr-spmf-gpv* :: *′a spmf* ⇒ (*′a*, *′out*, *′in*) *gpv* ⇒ *bool*
where *cr-spmf-gpv p gpv* ⟷ *gpv* = *lift-spmf p*

definition *spmf-of-gpv* :: (*′a*, *′out*, *′in*) *gpv* ⇒ *′a spmf*
where *spmf-of-gpv gpv* = (*THE p*. *gpv* = *lift-spmf p*)

lemma *spmf-of-gpv-lift-spmf* [*simp*]: *spmf-of-gpv* (*lift-spmf p*) = *p*
unfolding *spmf-of-gpv-def* **by** *auto*

lemma *rel-spmf-setD2*:
⟦ *rel-spmf A p q*; *y* ∈ *set-spmf q* ⟧ ⟹ ∃ *x*∈*set-spmf p*. *A x y*
by(*erule rel-spmfE*) *force*

lemma *rel-gpv-lift-spmf1*: *rel-gpv A B* (*lift-spmf p*) *gpv* ⟷ (∃ *q*. *gpv* = *lift-spmf q* ∧ *rel-spmf A p q*)
apply(*subst gpv.rel-sel*)
apply(*simp add*: *spmf-rel-map rel-generat-Pure1*)
apply *safe*
 apply(*rule exI*[**where** *x*=*map-spmf result* (*the-gpv gpv*)])
 apply(*clarsimp simp add*: *spmf-rel-map*)
 apply(*rule conjI*)
  apply(*rule gpv.expand*)
  apply(*simp add*: *spmf.map-comp*)
  apply(*subst map-spmf-cong*[*OF refl*, **where** *g*=*id*])
   apply(*drule* (*1*) *rel-spmf-setD2*)
   apply *clarsimp*

apply *simp*
apply(*erule rel-spmf-mono*)
apply *clarsimp*
apply(*clarsimp simp add*: *spmf-rel-map*)
done

lemma *rel-gpv-lift-spmf2*: *rel-gpv A B gpv* (*lift-spmf q*) $\longleftrightarrow$ ($\exists$ *p*. *gpv* = *lift-spmf p* $\wedge$ *rel-spmf A p q*)
by(*subst gpv.rel-flip*[*symmetric*])(*simp add*: *rel-gpv-lift-spmf1 pmf.rel-flip option.rel-conversep*)

definition *pcr-spmf-gpv* :: ($'a \Rightarrow 'b \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *spmf* $\Rightarrow$ ($'b$, $'out$, $'in$) *gpv* $\Rightarrow$ *bool*
where *pcr-spmf-gpv A* = *cr-spmf-gpv OO rel-gpv A* (=)

lemma *pcr-cr-eq-spmf-gpv*: *pcr-spmf-gpv* (=) = *cr-spmf-gpv*
by(*simp add*: *pcr-spmf-gpv-def gpv.rel-eq OO-eq*)

lemma *left-unique-cr-spmf-gpv*: *left-unique cr-spmf-gpv*
by(*rule left-uniqueI*)(*simp add*: *cr-spmf-gpv-def*)

lemma *left-unique-pcr-spmf-gpv* [*transfer-rule*]:
left-unique A $\Longrightarrow$ *left-unique* (*pcr-spmf-gpv A*)
unfolding *pcr-spmf-gpv-def* **by**(*intro left-unique-OO left-unique-cr-spmf-gpv left-unique-rel-gpv left-unique-eq*)

lemma *right-unique-cr-spmf-gpv*: *right-unique cr-spmf-gpv*
by(*rule right-uniqueI*)(*simp add*: *cr-spmf-gpv-def*)

lemma *right-unique-pcr-spmf-gpv* [*transfer-rule*]:
right-unique A $\Longrightarrow$ *right-unique* (*pcr-spmf-gpv A*)
unfolding *pcr-spmf-gpv-def* **by**(*intro right-unique-OO right-unique-cr-spmf-gpv right-unique-rel-gpv right-unique-eq*)

lemma *bi-unique-cr-spmf-gpv*: *bi-unique cr-spmf-gpv*
by(*simp add*: *bi-unique-alt-def left-unique-cr-spmf-gpv right-unique-cr-spmf-gpv*)

lemma *bi-unique-pcr-spmf-gpv* [*transfer-rule*]: *bi-unique A* $\Longrightarrow$ *bi-unique* (*pcr-spmf-gpv A*)
by(*simp add*: *bi-unique-alt-def left-unique-pcr-spmf-gpv right-unique-pcr-spmf-gpv*)

lemma *left-total-cr-spmf-gpv*: *left-total cr-spmf-gpv*
by(*rule left-totalI*)(*simp add*: *cr-spmf-gpv-def*)

lemma *left-total-pcr-spmf-gpv* [*transfer-rule*]: *left-total A* ==> *left-total* (*pcr-spmf-gpv A*)
unfolding *pcr-spmf-gpv-def* **by**(*intro left-total-OO left-total-cr-spmf-gpv left-total-rel-gpv left-total-eq*)

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

lemma *return-spmf-gpv-transfer'*:
((=) ===> *cr-spmf-gpv*) *return-spmf Done*
by(*rule rel-funI*)(*simp add*: *cr-spmf-gpv-def*)

lemma *return-spmf-gpv-transfer* [*transfer-rule*]:
(*A* ===> *pcr-spmf-gpv A*) *return-spmf Done*
unfolding *pcr-spmf-gpv-def*
apply(*rewrite* **in** (⌑ ===> -) - - *eq-OO*[*symmetric*])
apply(*rule pos-fun-distr*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule relcomppI*)
apply(*rule return-spmf-gpv-transfer'*)
apply *transfer-prover*
done

lemma *bind-spmf-gpv-transfer'*:
(*cr-spmf-gpv* ===> ((=) ===> *cr-spmf-gpv*) ===> *cr-spmf-gpv*) *bind-spmf bind-gpv*
apply(*clarsimp simp add*: *rel-fun-def cr-spmf-gpv-def*)
apply(*rule gpv.expand*)
apply(*simp add*: *bind-map-spmf map-spmf-bind-spmf o-def*)
done

lemma *bind-spmf-gpv-transfer* [*transfer-rule*]:
(*pcr-spmf-gpv A* ===> (*A* ===> *pcr-spmf-gpv B*) ===> *pcr-spmf-gpv B*) *bind-spmf bind-gpv*
unfolding *pcr-spmf-gpv-def*
apply(*rewrite* **in** (- ===> (⌑ ===> -) ===> -) - - *eq-OO*[*symmetric*])
apply(*rule fun-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule order.refl*)
apply(*rule fun-mono*)
apply(*rule neg-fun-distr3*[*OF left-unique-eq right-total-eq right-unique-cr-spmf-gpv left-total-cr-spmf-gpv*])
apply(*rule order.refl*)
apply(*rule fun-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule order.refl*)
apply(*rule pos-fun-distr*)
apply(*rule pos-fun-distr*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule relcomppI*)
apply(*rule bind-spmf-gpv-transfer'*)
apply *transfer-prover*
done

lemma *lift-spmf-gpv-transfer'*:
((=) ===> *cr-spmf-gpv*) ($\lambda x.\ x$) *lift-spmf*
by(*simp add*: *rel-fun-def cr-spmf-gpv-def*)

lemma *lift-spmf-gpv-transfer* [*transfer-rule*]:
(*rel-spmf A* ===> *pcr-spmf-gpv A*) (λ*x*. *x*) *lift-spmf*
unfolding *pcr-spmf-gpv-def*
apply(*rewrite* **in** (⌑ ===> -) - - *eq-OO*[*symmetric*])
apply(*rule pos-fun-distr*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule relcomppI*)
apply(*rule lift-spmf-gpv-transfer′*)
apply *transfer-prover*
done

lemma *fail-spmf-gpv-transfer′*: *cr-spmf-gpv* (*return-pmf None*) *Fail*
by(*simp add*: *cr-spmf-gpv-def*)

lemma *fail-spmf-gpv-transfer* [*transfer-rule*]: *pcr-spmf-gpv A* (*return-pmf None*) *Fail*
unfolding *pcr-spmf-gpv-def*
apply(*rule relcomppI*)
apply(*rule fail-spmf-gpv-transfer′*)
apply *transfer-prover*
done

lemma *map-spmf-gpv-transfer′*:
((=) ===> *R* ===> *cr-spmf-gpv* ===> *cr-spmf-gpv*) (λ*f g*. *map-spmf f*) *map-gpv*
by(*simp add*: *rel-fun-def cr-spmf-gpv-def map-lift-spmf*)

lemma *map-spmf-gpv-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *R* ===> *pcr-spmf-gpv A* ===> *pcr-spmf-gpv B*) (λ*f g*. *map-spmf f*) *map-gpv*
unfolding *pcr-spmf-gpv-def*
apply(*rewrite* **in** ((⌑ ===> -) ===> -) - - *eq-OO*[*symmetric*])
apply(*rewrite* **in** ((- ===> ⌑) ===> -) - - *eq-OO*[*symmetric*])
apply(*rewrite* **in** (- ===> ⌑ ===> -) - - *OO-eq*[*symmetric*])
apply(*rule fun-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule neg-fun-distr3*[*OF left-unique-eq right-total-eq right-unique-eq left-total-eq*])
apply(*rule fun-mono*[*OF order.refl*])
apply(*rule pos-fun-distr*)
apply(*rule fun-mono*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule order.refl*)
apply(*rule pos-fun-distr*)
apply(*rule pos-fun-distr*[*THEN le-funD*, *THEN le-funD*, *THEN le-boolD*, *THEN mp*])
apply(*rule relcomppI*)
apply(*unfold rel-fun-eq*)
apply(*rule map-spmf-gpv-transfer′*)
apply(*unfold rel-fun-eq*[*symmetric*])
apply *transfer-prover*
done

end

end

4.7 Embedding *'a option* as a monad

locale *option-to-gpv* **begin**

interpretation *option-to-spmf* **.**
interpretation *spmf-to-gpv* **.**

definition *cr-option-gpv* :: *'a option* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *bool*
where *cr-option-gpv x gpv* ⟷ *gpv* = (*lift-spmf* ∘ *return-pmf*) *x*

lemma *cr-option-gpv-conv-OO*:
cr-option-gpv = *cr-spmf-option*$^{-1-1}$ *OO cr-spmf-gpv*
by(*simp add*: *fun-eq-iff relcompp.simps cr-option-gpv-def cr-spmf-gpv-def cr-spmf-option-def*)

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

These transfer rules should follow from merging the transfer rules, but this has not yet been implemented.

lemma *return-option-gpv-transfer* [*transfer-rule*]:
((=) ===> *cr-option-gpv*) *Some Done*
by(*simp add*: *cr-option-gpv-def rel-fun-def*)

lemma *bind-option-gpv-transfer* [*transfer-rule*]:
(*cr-option-gpv* ===> ((=) ===> *cr-option-gpv*) ===> *cr-option-gpv*) *Option.bind bind-gpv*
apply(*clarsimp simp add*: *cr-option-gpv-def rel-fun-def*)
subgoal for *x f g* **by**(*cases x*; *simp*)
done

lemma *fail-option-gpv-transfer* [*transfer-rule*]: *cr-option-gpv None Fail*
by(*simp add*: *cr-option-gpv-def*)

lemma *map-option-gpv-transfer* [*transfer-rule*]:
((=) ===> *R* ===> *cr-option-gpv* ===> *cr-option-gpv*) (λ*f g*. *map-option f*) *map-gpv*
unfolding *rel-fun-eq* **by**(*simp add*: *rel-fun-def cr-option-gpv-def map-lift-spmf*)

end

end

locale *option-le-gpv* **begin**

interpretation *option-le-spmf* **.**

interpretation *spmf-to-gpv* **.**

definition *cr-option-le-gpv* :: *'a option* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *bool*
where *cr-option-le-gpv x gpv* ⟷ *gpv* = (*lift-spmf* ∘ *return-pmf*) *x* ∨ *x* = *None*

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

lemma *return-option-le-gpv-transfer* [*transfer-rule*]:
((=) ===> *cr-option-le-gpv*) *Some Done*
by(*simp add*: *cr-option-le-gpv-def rel-fun-def*)

lemma *bind-option-gpv-transfer* [*transfer-rule*]:
(*cr-option-le-gpv* ===> ((=) ===> *cr-option-le-gpv*) ===> *cr-option-le-gpv*) *Option.bind bind-gpv*
apply(*clarsimp simp add*: *cr-option-le-gpv-def rel-fun-def bind-eq-Some-conv*)
subgoal for *f g x y* **by**(*erule allE*[**where** *x*=*y*]) *auto*
done

lemma *fail-option-gpv-transfer* [*transfer-rule*]:
cr-option-le-gpv None Fail
by(*simp add*: *cr-option-le-gpv-def*)

lemma *map-option-gpv-transfer* [*transfer-rule*]:
(((=) ===> (=)) ===> *cr-option-le-gpv* ===> *cr-option-le-gpv*) *map-option* (λ*f*. *map-gpv f id*)
unfolding *rel-fun-eq* **by**(*simp add*: *rel-fun-def cr-option-le-gpv-def map-lift-spmf*)

end

end

4.8 Embedding resumptions

primcorec *lift-resumption* :: (*'a*, *'out*, *'in*) *resumption* ⇒ (*'a*, *'out*, *'in*) *gpv*
where
the-gpv (*lift-resumption r*) =
(*case r of resumption.Done None* ⇒ *return-pmf None*
| *resumption.Done* (*Some x'*) => *return-spmf* (*Pure x'*)
| *resumption.Pause out c* => *map-spmf* (*map-generat id id* ((∘) *lift-resumption*)) (*return-spmf* (*IO out c*)))

lemma *the-gpv-lift-resumption*:
the-gpv (*lift-resumption r*) =
(*if is-Done r then if Option.is-none* (*resumption.result r*) *then return-pmf None else return-spmf* (*Pure* (*the* (*resumption.result r*)))
else return-spmf (*IO* (*resumption.output r*) (*lift-resumption* ∘ *resume r*)))
by(*simp split*: *option.split resumption.split*)

declare *lift-resumption.simps* [*simp del*]

lemma *lift-resumption-Done* [*code*]:
lift-resumption (*resumption.Done x*) = (*case x of None* ⇒ *Fail* | *Some x'* ⇒ *Done x'*)
by(*rule gpv.expand*)(*simp add*: *the-gpv-lift-resumption split*: *option.split*)

lemma *lift-resumption-DONE* [*simp*]:
lift-resumption (*DONE x*) = *Done x*
by(*simp add*: *DONE-def lift-resumption-Done*)

lemma *lift-resumption-ABORT* [*simp*]:
lift-resumption ABORT = *Fail*
by(*simp add*: *ABORT-def lift-resumption-Done*)

lemma *lift-resumption-Pause* [*simp*, *code*]:
lift-resumption (*resumption.Pause out c*) = *Pause out* (*lift-resumption* ∘ *c*)
by(*rule gpv.expand*)(*simp add*: *the-gpv-lift-resumption*)

lemma *lift-resumption-Done-Some* [*simp*]: *lift-resumption* (*resumption.Done* (*Some x*)) = *Done x*
using *lift-resumption-DONE* **unfolding** *DONE-def* **by** *simp*

lemma *results'-gpv-lift-resumption* [*simp*]:
results'-gpv (*lift-resumption r*) = *results r* (**is** *?lhs* = *?rhs*)
proof(*rule set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv*≡*lift-resumption r arbitrary*: *r*)
(*auto intro*: *resumption.set-sel simp add*: *lift-resumption.sel split*: *resumption.split-asm option.split-asm*)
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that* **by** *induction*(*auto simp add*: *lift-resumption.sel*)
qed

lemma *outs'-gpv-lift-resumption* [*simp*]:
outs'-gpv (*lift-resumption r*) = *outputs r* (**is** *?lhs* = *?rhs*)
proof(*rule set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv*≡*lift-resumption r arbitrary*: *r*)
(*auto simp add*: *lift-resumption.sel split*: *resumption.split-asm option.split-asm*)
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that* **by** *induction auto*
qed

lemma *pred-gpv-lift-resumption* [*simp*]:
⋀*A*. *pred-gpv A C* (*lift-resumption r*) = *pred-resumption A C r*
by(*simp add*: *pred-gpv-def pred-resumption-def*)

lemma *lift-resumption-bind*: *lift-resumption* (*r* ≫= *f*) = *lift-resumption r* ≫= *lift-resumption* ∘ *f*
by(*coinduction arbitrary*: *r rule*: *gpv.coinduct-strong*)

(*auto simp add*: *lift-resumption.sel Done-bind split*: *resumption.split option.split del*: *rel-funI intro*!: *rel-funI*)

4.9 Assertions

definition *assert-gpv* :: *bool* ⇒ (*unit*, *'out*, *'in*) *gpv*
where *assert-gpv b* = (*if b then Done* () *else Fail*)

lemma *assert-gpv-simps* [*simp*]:
  assert-gpv True = *Done* ()
  assert-gpv False = *Fail*
by(*simp-all add*: *assert-gpv-def*)

lemma [*simp*]:
  shows *assert-gpv-eq-Done*: *assert-gpv b* = *Done x* ⟷ *b*
  and *Done-eq-assert-gpv*: *Done x* = *assert-gpv b* ⟷ *b*
  and *Pause-neq-assert-gpv*: *Pause out rpv* ≠ *assert-gpv b*
  and *assert-gpv-neq-Pause*: *assert-gpv b* ≠ *Pause out rpv*
  and *assert-gpv-eq-Fail*: *assert-gpv b* = *Fail* ⟷ ¬ *b*
  and *Fail-eq-assert-gpv*: *Fail* = *assert-gpv b* ⟷ ¬ *b*
by(*simp-all add*: *assert-gpv-def*)

lemma *assert-gpv-inject* [*simp*]: *assert-gpv b* = *assert-gpv b'* ⟷ *b* = *b'*
by(*simp add*: *assert-gpv-def*)

lemma *assert-gpv-sel* [*simp*]:
  the-gpv (*assert-gpv b*) = *map-spmf Pure* (*assert-spmf b*)
by(*simp add*: *assert-gpv-def*)

lemma *the-gpv-bind-assert* [*simp*]:
  the-gpv (*bind-gpv* (*assert-gpv b*) *f*) =
   bind-spmf (*assert-spmf b*) (*the-gpv* ∘ *f*)
by(*cases b*) *simp-all*

lemma *pred-gpv-assert* [*simp*]: *pred-gpv P Q* (*assert-gpv b*) = (*b* ⟶ *P* ())
by(*cases b*) *simp-all*

primcorec *try-gpv* :: (*'a*, *'call*, *'ret*) *gpv* ⇒ (*'a*, *'call*, *'ret*) *gpv* ⇒ (*'a*, *'call*, *'ret*) *gpv* (‹*TRY - ELSE -*› [*0*,*60*] *59*)
where
  the-gpv (*TRY gpv ELSE gpv'*) =
   map-spmf (*map-generat id id* (*λc input*. *case c input of Inl gpv* ⇒ *try-gpv gpv gpv'* | *Inr gpv'* ⇒ *gpv'*))
     (*try-spmf* (*map-spmf* (*map-generat id id* (*map-fun id Inl*)) (*the-gpv gpv*))
         (*map-spmf* (*map-generat id id* (*map-fun id Inr*)) (*the-gpv gpv'*)))

lemma *try-gpv-sel*:
  the-gpv (*TRY gpv ELSE gpv'*) =
  TRY map-spmf (*map-generat id id* (*λc input*. *TRY c input ELSE gpv'*)) (*the-gpv*

gpv) *ELSE the-gpv gpv′*
by(*simp add*: *try-gpv-def map-try-spmf spmf.map-comp o-def generat.map-comp generat.map-ident id-def*)

lemma *try-gpv-Done* [*simp*]: *TRY Done x ELSE gpv′* = *Done x*
by(*rule gpv.expand*)(*simp*)

lemma *try-gpv-Fail* [*simp*]: *TRY Fail ELSE gpv′* = *gpv′*
by(*rule gpv.expand*)(*simp add*: *spmf.map-comp o-def generat.map-comp generat.map-ident*)

lemma *try-gpv-Pause* [*simp*]: *TRY Pause out c ELSE gpv′* = *Pause out* (*λinput. TRY c input ELSE gpv′*)
by(*rule gpv.expand*) *simp*

lemma *try-gpv-Fail2* [*simp*]: *TRY gpv ELSE Fail* = *gpv*
by(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
(*auto simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI generat.rel-refl*)

lemma *lift-try-spmf*: *lift-spmf* (*TRY p ELSE q*) = *TRY lift-spmf p ELSE lift-spmf q*
by(*rule gpv.expand*)(*simp add*: *map-try-spmf spmf.map-comp o-def*)

lemma *try-assert-gpv*: *TRY assert-gpv b ELSE gpv′* = (*if b then Done* () *else gpv′*)
by(*simp*)

context includes *lifting-syntax* **begin**
lemma *try-gpv-parametric* [*transfer-rule*]:
(*rel-gpv A C* ===> *rel-gpv A C* ===> *rel-gpv A C*) *try-gpv try-gpv*
unfolding *try-gpv-def* **by** *transfer-prover*

lemma *try-gpv-parametric′*:
(*rel-gpv″ A C R* ===> *rel-gpv″ A C R* ===> *rel-gpv″ A C R*) *try-gpv try-gpv*
unfolding *try-gpv-def*
supply *corec-gpv-parametric′*[*transfer-rule*] *the-gpv-parametric′*[*transfer-rule*]
by *transfer-prover*
end

lemma *map-try-gpv*: *map-gpv f g* (*TRY gpv ELSE gpv′*) = *TRY map-gpv f g gpv ELSE map-gpv f g gpv′*
by(*simp add*: *gpv.rel-map try-gpv-parametric*[*THEN rel-funD*, *THEN rel-funD*] *gpv.rel-refl gpv.rel-eq*[*symmetric*])

lemma *map′-try-gpv*: *map-gpv′ f g h* (*TRY gpv ELSE gpv′*) = *TRY map-gpv′ f g h gpv ELSE map-gpv′ f g h gpv′*
by(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI generat.rel-refl rel-funI rel-spmf-try-spmf*)

lemma *try-bind-assert-gpv*:

TRY (*assert-gpv b* ≫= *f*) *ELSE gpv* = (*if b then TRY* (*f* ()) *ELSE gpv else gpv*)
by(*simp*)

4.10 Order for ('*a*, '*out*, '*in*) *gpv*

coinductive *ord-gpv* :: ('*a*, '*out*, '*in*) *gpv* ⇒ ('*a*, '*out*, '*in*) *gpv* ⇒ *bool*
where
ord-spmf (*rel-generat* (=) (=) (*rel-fun* (=) *ord-gpv*)) *f g* ⟹ *ord-gpv* (*GPV f*) (*GPV g*)

inductive-simps *ord-gpv-simps* [*simp*]:
ord-gpv (*GPV f*) (*GPV g*)

lemma *ord-gpv-coinduct* [*consumes 1*, *case-names ord-gpv*, *coinduct pred*: *ord-gpv*]:
assumes *X f g*
and *step*: ⋀*f g*. *X f g* ⟹ *ord-spmf* (*rel-generat* (=) (=) (*rel-fun* (=) *X*)) (*the-gpv f*) (*the-gpv g*)
shows *ord-gpv f g*
using ‹*X f g*›
by(*coinduct*)(*auto dest*: *step simp add*: *eq-GPV-iff intro*: *ord-spmf-mono rel-generat-mono rel-fun-mono*)

lemma *ord-gpv-the-gpvD*:
ord-gpv f g ⟹ *ord-spmf* (*rel-generat* (=) (=) (*rel-fun* (=) *ord-gpv*)) (*the-gpv f*) (*the-gpv g*)
by(*erule ord-gpv.cases*) *simp*

lemma *reflp-equality*: *reflp* (=)
by(*simp add*: *reflp-def*)

lemma *ord-gpv-reflI* [*simp*]: *ord-gpv f f*
by(*coinduction arbitrary*: *f*)(*auto intro*: *ord-spmf-reflI simp add*: *rel-generat-same rel-fun-def*)

lemma *reflp-ord-gpv*: *reflp ord-gpv*
by(*rule reflpI*)(*rule ord-gpv-reflI*)

lemma *ord-gpv-trans*:
assumes *ord-gpv f g ord-gpv g h*
shows *ord-gpv f h*
using *assms*
proof(*coinduction arbitrary*: *f g h*)
case (*ord-gpv f g h*)
have ∗: *ord-spmf* (*rel-generat* (=) (=) (*rel-fun* (=) (λ*f h*. ∃ *g*. *ord-gpv f g* ∧ *ord-gpv g h*))) (*the-gpv f*) (*the-gpv h*) =
ord-spmf (*rel-generat* ((=) *OO* (=)) ((=) *OO* (=)) (*rel-fun* (=) (*ord-gpv OO ord-gpv*))) (*the-gpv f*) (*the-gpv h*)
by(*simp add*: *relcompp.simps*[*abs-def*])
then show *?case* **using** *ord-gpv*

by(*auto elim*!: *ord-gpv.cases simp add*: *generat.rel-compp ord-spmf-compp fun.rel-compp*)
qed

lemma *ord-gpv-compp*: (*ord-gpv OO ord-gpv*) = *ord-gpv*
by(*auto simp add*: *fun-eq-iff intro*: *ord-gpv-trans*)

lemma *transp-ord-gpv* [*simp*]: *transp ord-gpv*
by(*blast intro*: *transpI ord-gpv-trans*)

lemma *ord-gpv-antisym*:
⟦ *ord-gpv f g*; *ord-gpv g f* ⟧ ⟹ *f* = *g*
proof(*coinduction arbitrary*: *f g*)
case (*Eq-gpv f g*)
let *?R* = *rel-generat* (=) (=) (*rel-fun* (=) *ord-gpv*)
from ‹*ord-gpv f g*› **have** *ord-spmf ?R* (*the-gpv f*) (*the-gpv g*) **by** *cases simp*
moreover
from ‹*ord-gpv g f*› **have** *ord-spmf ?R* (*the-gpv g*) (*the-gpv f*) **by** *cases simp*
ultimately have *rel-spmf* (*inf ?R ?R*$^{-1-1}$) (*the-gpv f*) (*the-gpv g*)
by(*rule rel-spmf-inf*)(*auto 4 3 intro*: *transp-rel-generatI transp-ord-gpv reflp-ord-gpv reflp-equality reflp-fun1 is-equality-eq transp-rel-fun*)
also have *inf ?R ?R*$^{-1-1}$ = *rel-generat* (*inf* (=) (=)) (*inf* (=) (=)) (*rel-fun* (=) (*inf ord-gpv ord-gpv*$^{-1-1}$))
unfolding *rel-generat-inf*[*symmetric*] *rel-fun-inf*[*symmetric*]
by(*simp add*: *generat.rel-conversep*[*symmetric*] *fun.rel-conversep*)
finally show *?case* **by**(*simp add*: *inf-fun-def*)
qed

lemma *RFail-least* [*simp*]: *ord-gpv Fail f*
by(*coinduction arbitrary*: *f*)(*simp add*: *eq-GPV-iff*)

4.11 Bounds on interaction

context
fixes *consider* :: *'out* ⇒ *bool*
notes *monotone-SUP*[*partial-function-mono*] [[*function-internals*]]
begin
declaration ‹*Partial-Function.init lfp-strong* @{*term lfp.fixp-fun*} @{*term lfp.mono-body*} @{*thm lfp.fixp-rule-uc*} @{*thm lfp.fixp-induct-strong2-uc*} *NONE*›

partial-function (*lfp-strong*) *interaction-bound* :: (*'a*, *'out*, *'in*) *gpv* ⇒ *enat*
where
interaction-bound gpv =
(*SUP generat*∈*set-spmf* (*the-gpv gpv*). *case generat of Pure* - ⇒ *0*
| *IO out c* ⇒ *if consider out then eSuc* (*SUP input*. *interaction-bound* (*c input*)) *else* (*SUP input*. *interaction-bound* (*c input*)))

lemma *interaction-bound-fixp-induct* [*case-names adm bottom step*]:
⟦ *ccpo.admissible* (*fun-lub Sup*) (*fun-ord* (≤)) *P*;
P (λ-. *0*);

⋀*interaction-bound′*.
⟦ *P interaction-bound′*;
⋀*gpv*. *interaction-bound′ gpv* ≤ *interaction-bound gpv*;
⋀*gpv*. *interaction-bound′ gpv* ≤ (*SUP generat*∈*set-spmf* (*the-gpv gpv*). *case generat of Pure* - ⇒ *0*
| *IO out c* ⇒ *if consider out then eSuc* (*SUP input*. *interaction-bound′* (*c input*)) *else* (*SUP input*. *interaction-bound′* (*c input*)))
⟧
⟹ *P* (λ*gpv*. ⨆*generat*∈*set-spmf* (*the-gpv gpv*). *case generat of Pure x* ⇒ *0*
| *IO out c* ⇒ *if consider out then eSuc* (⨆*input*. *interaction-bound′* (*c input*)) *else* (⨆*input*. *interaction-bound′* (*c input*))) ⟧
⟹ *P interaction-bound*
by(*erule interaction-bound.fixp-induct*)(*simp-all add*: *bot-enat-def fun-ord-def*)

lemma *interaction-bound-IO*:
IO out c ∈ *set-spmf* (*the-gpv gpv*)
⟹ (*if consider out then eSuc* (*interaction-bound* (*c input*)) *else interaction-bound* (*c input*)) ≤ *interaction-bound gpv*
by(*rewrite* **in** - ≤ ⌑ *interaction-bound.simps*)(*auto intro*!: *SUP-upper2*)

lemma *interaction-bound-IO-consider*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *consider out* ⟧
⟹ *eSuc* (*interaction-bound* (*c input*)) ≤ *interaction-bound gpv*
by(*drule interaction-bound-IO*) *simp*

lemma *interaction-bound-IO-ignore*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); ¬ *consider out* ⟧
⟹ *interaction-bound* (*c input*) ≤ *interaction-bound gpv*
by(*drule interaction-bound-IO*) *simp*

lemma *interaction-bound-Done* [*simp*]: *interaction-bound* (*Done x*) = *0*
by(*simp add*: *interaction-bound.simps*)

lemma *interaction-bound-Fail* [*simp*]: *interaction-bound Fail* = *0*
by(*simp add*: *interaction-bound.simps bot-enat-def*)

lemma *interaction-bound-Pause* [*simp*]:
interaction-bound (*Pause out c*) =
(*if consider out then eSuc* (*SUP input*. *interaction-bound* (*c input*)) *else* (*SUP input*. *interaction-bound* (*c input*)))
by(*simp add*: *interaction-bound.simps*)

lemma *interaction-bound-lift-spmf* [*simp*]: *interaction-bound* (*lift-spmf p*) = *0*
by(*simp add*: *interaction-bound.simps SUP-constant bot-enat-def*)

lemma *interaction-bound-assert-gpv* [*simp*]: *interaction-bound* (*assert-gpv b*) = *0*
by(*cases b*) *simp-all*

lemma *interaction-bound-bind-step*:

```
  assumes IH: ⋀p. interaction-bound' (p ≫= f) ≤ interaction-bound p + (⨆x∈results'-gpv
p. interaction-bound' (f x))
  and unfold: ⋀gpv. interaction-bound' gpv ≤ (⨆generat∈set-spmf (the-gpv gpv).
case generat of Pure x ⇒ 0
            | IO out c ⇒ if consider out then eSuc (⨆input. interaction-bound' (c
input)) else ⨆input. interaction-bound' (c input))
  shows (⨆generat∈set-spmf (the-gpv (p ≫= f)).
            case generat of Pure x ⇒ 0
            | IO out c ⇒
                if consider out then eSuc (⨆input. interaction-bound' (c input))
                else ⨆input. interaction-bound' (c input))
         ≤ interaction-bound p +
           (⨆x∈results'-gpv p.
               ⨆generat∈set-spmf (the-gpv (f x)).
                  case generat of Pure x ⇒ 0
                  | IO out c ⇒
                     if consider out then eSuc (⨆input. interaction-bound' (c input))
                      else ⨆input. interaction-bound' (c input))
    (is (SUP generat'∈?bind. ?g generat') ≤ ?p + ?f)
proof(rule SUP-least)
  fix generat'
  assume generat' ∈ ?bind
  then obtain generat where generat: generat ∈ set-spmf (the-gpv p)
    and *: case generat of Pure x ⇒ generat' ∈ set-spmf (the-gpv (f x))
        | IO out c ⇒ generat' = IO out (λinput. c input ≫= f)
    by(clarsimp simp add: bind-gpv.sel simp del: bind-gpv-sel')
       (clarsimp split: generat.split-asm simp add: generat.map-comp o-def gen-
erat.map-id[unfolded id-def])
  show ?g generat' ≤ ?p + ?f
  proof(cases generat)
    case (Pure x)
    have ?g generat' ≤ (SUP generat'∈set-spmf (the-gpv (f x)). (case generat' of
Pure x ⇒ 0 | IO out c ⇒ if consider out then eSuc (⨆input. interaction-bound'
(c input)) else ⨆input. interaction-bound' (c input)))
      using * Pure by(auto intro: SUP-upper)
    also have ... ≤ 0 + ?f using generat Pure
      by(auto 4 3 intro: SUP-upper results'-gpv-Pure)
    also have ... ≤ ?p + ?f by simp
    finally show ?thesis .
  next
    case (IO out c)
     with * have ?g generat' = (if consider out then eSuc (SUP input. interac-
tion-bound' (c input ≫= f)) else (SUP input. interaction-bound' (c input ≫= f)))
by simp
     also have ... ≤ (if consider out then eSuc (SUP input. interaction-bound (c
input) + (⨆x∈results'-gpv (c input). interaction-bound' (f x))) else (SUP input.
interaction-bound (c input) + (⨆x∈results'-gpv (c input). interaction-bound' (f
x))))
      by(auto intro: SUP-mono IH)
```

also have ... ≤ (*case IO out c of Pure (x :: 'a) ⇒ 0 | IO out c ⇒ if consider out then eSuc (SUP input. interaction-bound (c input)) else (SUP input. interaction-bound (c input))) + (SUP input. SUP x∈results'-gpv (c input). interaction-bound' (f x)*)
by(*simp add*: *iadd-Suc SUP-le-iff*)(*meson SUP-upper2 UNIV-I add-mono order-refl*)
also have ... ≤ *?p* + *?f*
apply(*rewrite* **in** - ≤ ⌑ *interaction-bound.simps*)
apply(*rule add-mono SUP-least SUP-upper generat*[*unfolded IO*])+
apply(*rule order-trans*[*OF unfold*])
apply(*auto 4 3 intro*: *results'-gpv-Cont*[*OF generat*] *SUP-upper simp add*: *IO*)
done
finally show *?thesis* **.**
qed
qed

lemma *interaction-bound-bind*:
defines *ib1* ≡ *interaction-bound*
shows *interaction-bound* (*p* ⤜ *f*) ≤ *ib1 p* + (*SUP x∈results'-gpv p. interaction-bound* (*f x*))
proof(*induction arbitrary*: *p rule*: *interaction-bound-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step interaction-bound'*) **then show** *?case* **unfolding** *ib1-def* **by** −(*rule interaction-bound-bind-step*)
qed

lemma *interaction-bound-bind-lift-spmf* [*simp*]:
interaction-bound (*lift-spmf p* ⤜ *f*) = (*SUP x∈set-spmf p. interaction-bound* (*f x*))
by(*subst* (*1 2*) *interaction-bound.simps*)(*simp add*: *bind-UNION SUP-UNION*)

end

lemma *interaction-bound-map-gpv'*:
assumes *surj h*
shows *interaction-bound consider* (*map-gpv' f g h gpv*) = *interaction-bound* (*consider* ∘ *g*) *gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF lattice-partial-function-definition lattice-partial-function-definition interaction-bound.mono interaction-bound.mono interaction-bound-def interaction-bound-def*, *case-names adm bottom step*])
case (*step interaction-bound' interaction-bound'' gpv*)
have ∗: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *x* ∈ *UNIV* ⟹ *interaction-bound''* (*c x*) ≤ (⨆*x. interaction-bound''* (*c* (*h x*))) **for** *out c x*
using *assms*[*THEN surjD*, *of x*] **by** (*clarsimp intro*!: *SUP-upper*)

show *?case*
by (*auto simp add*: ∗ *step.IH image-comp split*: *generat.split*
intro!: *SUP-cong* [*OF refl*] *antisym SUP-upper SUP-least*)

qed *simp-all*

abbreviation *interaction-any-bound* :: ('*a*, '*out*, '*in*) *gpv* ⇒ *enat*
where *interaction-any-bound* ≡ *interaction-bound* (λ-. *True*)

lemma *interaction-any-bound-coinduct* [*consumes 1*, *case-names interaction-bound*]:
  assumes *X*: *X gpv n*
  and ∗: ⋀*gpv n out c input*. ⟦ *X gpv n*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟧
    ⟹ ∃ *n*'. (*X* (*c input*) *n*' ∨ *interaction-any-bound* (*c input*) ≤ *n*') ∧ *eSuc n*' ≤ *n*
  shows *interaction-any-bound gpv* ≤ *n*
using *X*
proof(*induction arbitrary*: *gpv n rule*: *interaction-bound-fixp-induct*)
  case *adm* **show** *?case* **by**(*intro cont-intro*)
  case *bottom* **show** *?case* **by** *simp*
next
  case (*step interaction-bound*')
  { fix *out c*
    assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
    from ∗[*OF step.prems IO*] **obtain** *n*' **where** *n*: *n* = *eSuc n*'
      by(*cases n rule*: *co.enat.exhaust*) *auto*
    moreover
    { fix *input*
      have ∃ *n*''. (*X* (*c input*) *n*'' ∨ *interaction-any-bound* (*c input*) ≤ *n*'') ∧ *eSuc n*'' ≤ *n*
        using *step.prems IO* ‹*n* = *eSuc n*'› **by**(*auto 4 3 dest*: ∗)
      then have *interaction-bound*' (*c input*) ≤ *n*' **using** *n*
        by(*auto dest*: *step.IH intro*: *step.hyps*[*THEN order-trans*] *elim*!: *order-trans simp add*: *neq-zero-conv-eSuc*) **}**
    ultimately have *eSuc* (⨆*input*. *interaction-bound*' (*c input*)) ≤ *n*
      by(*auto intro*: *SUP-least*) **}**
  then show *?case* **by**(*auto intro*!: *SUP-least split*: *generat.split*)
qed

context includes *lifting-syntax* **begin**
lemma *interaction-bound-parametric*':
  assumes [*transfer-rule*]: *bi-total R*
  shows ((*C* ===> (=)) ===> *rel-gpv*'' *A C R* ===> (=)) *interaction-bound interaction-bound*
unfolding *interaction-bound-def*[*abs-def*]
apply(*rule rel-funI*)
apply(*rule fixp-lfp-parametric-eq*[*OF interaction-bound.mono interaction-bound.mono*])
subgoal premises [*transfer-rule*]
  supply *the-gpv-parametric*'[*transfer-rule*] *rel-gpv*''*-eq*[*relator-eq*]
  by *transfer-prover*
done

lemma *interaction-bound-parametric* [*transfer-rule*]:
  ((*C* ===> (=)) ===> *rel-gpv A C* ===> (=)) *interaction-bound interac-*

tion-bound
unfolding *rel-gpv-conv-rel-gpv''* **by**(*rule interaction-bound-parametric'*)(*rule bi-total-eq*)
end

There is no nice *interaction-bound* equation for (⪢=), as it computes an exact bound, but we only need an upper bound. As *enat* is hard to work with (and ∞ does not constrain a gpv in any way), we work with *nat*.

inductive *interaction-bounded-by* :: (*'out* ⇒ *bool*) ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *enat* ⇒ *bool*
for *consider gpv n* **where**
interaction-bounded-by: ⟦ *interaction-bound consider gpv* ≤ *n* ⟧ ⟹ *interaction-bounded-by consider gpv n*

lemmas *interaction-bounded-byI* = *interaction-bounded-by*
hide-fact (**open**) *interaction-bounded-by*

context includes *lifting-syntax* **begin**
lemma *interaction-bounded-by-parametric* [*transfer-rule*]:
((*C* ===> (=)) ===> *rel-gpv A C* ===> (=) ===> (=)) *interaction-bounded-by interaction-bounded-by*
unfolding *interaction-bounded-by.simps*[*abs-def*] **by** *transfer-prover*

lemma *interaction-bounded-by-parametric'*:
notes *interaction-bound-parametric'*[*transfer-rule*]
assumes [*transfer-rule*]: *bi-total R*
shows ((*C* ===> (=)) ===> *rel-gpv'' A C R* ===> (=) ===> (=))
interaction-bounded-by interaction-bounded-by
unfolding *interaction-bounded-by.simps*[*abs-def*] **by** *transfer-prover*
end

lemma *interaction-bounded-by-mono*:
⟦ *interaction-bounded-by consider gpv n*; *n* ≤ *m* ⟧ ⟹ *interaction-bounded-by consider gpv m*
unfolding *interaction-bounded-by.simps* **by**(*erule order-trans*) *simp*

lemma *interaction-bounded-by-contD*:
⟦ *interaction-bounded-by consider gpv n*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *consider out* ⟧
⟹ *n* > *0* ∧ *interaction-bounded-by consider* (*c input*) (*n* − *1*)
unfolding *interaction-bounded-by.simps*
by(*subst* (*asm*) *interaction-bound.simps*)(*auto simp add*: *SUP-le-iff eSuc-le-iff enat-eSuc-iff dest*!: *bspec*)

lemma *interaction-bounded-by-contD-ignore*:
⟦ *interaction-bounded-by consider gpv n*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟧
⟹ *interaction-bounded-by consider* (*c input*) *n*
unfolding *interaction-bounded-by.simps*
by(*subst* (*asm*) *interaction-bound.simps*)(*auto 4 4 simp add*: *SUP-le-iff eSuc-le-iff enat-eSuc-iff dest*!: *bspec split*: *if-split-asm elim*: *order-trans*)

lemma *interaction-bounded-byI-epred*:
assumes $\bigwedge$*out c*. ⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *consider out* ⟧ ⟹ *n* ≠ *0* ∧ (∀ *input*. *interaction-bounded-by consider* (*c input*) (*n* − *1*))
and $\bigwedge$*out c input*. ⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); ¬ *consider out* ⟧ ⟹ *interaction-bounded-by consider* (*c input*) *n*
shows *interaction-bounded-by consider gpv n*
unfolding *interaction-bounded-by.simps*
by(*subst interaction-bound.simps*)(*auto 4 5 intro*!: *SUP-least split*: *generat.split dest*: *assms simp add*: *eSuc-le-iff enat-eSuc-iff gr0-conv-Suc neq-zero-conv-eSuc interaction-bounded-by.simps*)

lemma *interaction-bounded-by-IO*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *interaction-bounded-by consider gpv n*; *consider out* ⟧
⟹ *n* ≠ *0* ∧ *interaction-bounded-by consider* (*c input*) (*n* − *1*)
by(*drule interaction-bound-IO*[**where** *input*=*input* **and** *?consider*=*consider*])(*auto simp add*: *interaction-bounded-by.simps epred-conv-minus eSuc-le-iff enat-eSuc-iff*)

lemma *interaction-bounded-by-0*: *interaction-bounded-by consider gpv 0* ⟷ *interaction-bound consider gpv* = *0*
by(*simp add*: *interaction-bounded-by.simps zero-enat-def* [*symmetric*])

abbreviation *interaction-bounded-by′* :: (′*out* ⇒ *bool*) ⇒ (′*a*, ′*out*, ′*in*) *gpv* ⇒ *nat* ⇒ *bool*
where *interaction-bounded-by′ consider gpv n* ≡ *interaction-bounded-by consider gpv* (*enat n*)

named-theorems *interaction-bound*

lemmas *interaction-bounded-by-start* = *interaction-bounded-by-mono*

method *interaction-bound-start* = (*rule interaction-bounded-by-start*)
method *interaction-bound-step* **uses** *add simp* =
((*match* **conclusion in** *interaction-bounded-by* - - - ⇒ *fail* | - ⇒ ‹*solves* ‹*clarsimp simp add*: *simp*››) | *rule add interaction-bound*)
method *interaction-bound-rec* **uses** *add simp* =
(*interaction-bound-step add*: *add simp*: *simp*; (*interaction-bound-rec add*: *add simp*: *simp*)*?*)
method *interaction-bound* **uses** *add simp* =
(*interaction-bound-start*, *interaction-bound-rec add*: *add simp*: *simp*)

lemma *interaction-bounded-by-Done* [*simp*]: *interaction-bounded-by consider* (*Done x*) *n*
by(*simp add*: *interaction-bounded-by.simps*)

lemma *interaction-bounded-by-DoneI* [*interaction-bound*]:
interaction-bounded-by consider (*Done x*) *0*
by *simp*

lemma *interaction-bounded-by-Fail* [*simp*]: *interaction-bounded-by consider Fail n*
by(*simp add*: *interaction-bounded-by.simps*)

lemma *interaction-bounded-by-FailI* [*interaction-bound*]: *interaction-bounded-by consider Fail 0*
by *simp*

lemma *interaction-bounded-by-lift-spmf* [*simp*]: *interaction-bounded-by consider* (*lift-spmf p*) *n*
by(*simp add*: *interaction-bounded-by.simps*)

lemma *interaction-bounded-by-lift-spmfI* [*interaction-bound*]:
interaction-bounded-by consider (*lift-spmf p*) *0*
by *simp*

lemma *interaction-bounded-by-assert-gpv* [*simp*]: *interaction-bounded-by consider* (*assert-gpv b*) *n*
by(*cases b*) *simp-all*

lemma *interaction-bounded-by-assert-gpvI* [*interaction-bound*]:
interaction-bounded-by consider (*assert-gpv b*) *0*
by *simp*

lemma *interaction-bounded-by-Pause* [*simp*]:
interaction-bounded-by consider (*Pause out c*) *n* $\longleftrightarrow$
(*if consider out then* $0 < n \wedge (\forall$ *input. interaction-bounded-by consider* (*c input*) $(n - 1))$ *else* ($\forall$ *input. interaction-bounded-by consider* (*c input*) *n*))
by(*cases n rule*: *co.enat.exhaust*)
(*auto 4 3 simp add*: *interaction-bounded-by.simps eSuc-le-iff enat-eSuc-iff gr0-conv-Suc intro*: *SUP-least dest*: *order-trans*[*OF SUP-upper*, *rotated*])

lemma *interaction-bounded-by-PauseI* [*interaction-bound*]:
($\bigwedge$*input. interaction-bounded-by consider* (*c input*) (*n input*))
$\Longrightarrow$ *interaction-bounded-by consider* (*Pause out c*) (*if consider out then 1* + (*SUP input. n input*) *else* (*SUP input. n input*))
by(*auto simp add*: *iadd-is-0 enat-add-sub-same intro*: *interaction-bounded-by-mono SUP-upper*)

lemma *interaction-bounded-by-bindI* [*interaction-bound*]:
$[\![$ *interaction-bounded-by consider gpv n*; $\bigwedge x.\ x \in$ *results'-gpv gpv* $\Longrightarrow$ *interaction-bounded-by consider* (*f x*) (*m x*) $]\!]$
$\Longrightarrow$ *interaction-bounded-by consider* (*gpv* $\gg=$ *f*) (*n* + (*SUP x*$\in$*results'-gpv gpv. m x*))
unfolding *interaction-bounded-by.simps plus-enat-simps*(*1*)[*symmetric*]
by(*rule interaction-bound-bind*[*THEN order-trans*])(*auto intro*: *add-mono SUP-mono*)

lemma *interaction-bounded-by-bind-PauseI* [*interaction-bound*]:
($\bigwedge$*input. interaction-bounded-by consider* (*c input* $\gg=$ *f*) (*n input*))

⟹ *interaction-bounded-by consider* (*Pause out c* ⤜ *f*) (*if consider out then SUP input. n input* + *1 else SUP input. n input*)
by(*auto 4 3 simp add*: *interaction-bounded-by.simps SUP-enat-add-left eSuc-plus-1 intro*: *SUP-least SUP-upper2*)

lemma *interaction-bounded-by-bind-lift-spmf* [*simp*]:
interaction-bounded-by consider (*lift-spmf p* ⤜ *f*) *n* ⟷ (∀ *x*∈*set-spmf p*. *interaction-bounded-by consider* (*f x*) *n*)
by(*simp add*: *interaction-bounded-by.simps SUP-le-iff*)

lemma *interaction-bounded-by-bind-lift-spmfI* [*interaction-bound*]:
(⋀*x*. *x* ∈ *set-spmf p* ⟹ *interaction-bounded-by consider* (*f x*) (*n x*))
⟹ *interaction-bounded-by consider* (*lift-spmf p* ⤜ *f*) (*SUP x*∈*set-spmf p*. *n x*)
by(*auto intro*: *interaction-bounded-by-mono SUP-upper*)

lemma *interaction-bounded-by-bind-DoneI* [*interaction-bound*]:
interaction-bounded-by consider (*f x*) *n* ⟹ *interaction-bounded-by consider* (*Done x* ⤜ *f*) *n*
by(*simp*)

lemma *interaction-bounded-by-if* [*interaction-bound*]:
⟦ *b* ⟹ *interaction-bounded-by consider gpv1 n*; ¬ *b* ⟹ *interaction-bounded-by consider gpv2 m* ⟧
⟹ *interaction-bounded-by consider* (*if b then gpv1 else gpv2*) (*if b then n else m*)
by(*auto 4 3 simp add*: *max-def not-le elim*: *interaction-bounded-by-mono*)

lemma *interaction-bounded-by-case-bool* [*interaction-bound*]:
⟦ *b* ⟹ *interaction-bounded-by consider t bt*; ¬ *b* ⟹ *interaction-bounded-by consider f bf* ⟧
⟹ *interaction-bounded-by consider* (*case-bool t f b*) (*if b then bt else bf*)
by(*cases b*)(*auto*)

lemma *interaction-bounded-by-case-sum* [*interaction-bound*]:
⟦ ⋀*y*. *x* = *Inl y* ⟹ *interaction-bounded-by consider* (*l y*) (*bl y*);
⋀*y*. *x* = *Inr y* ⟹ *interaction-bounded-by consider* (*r y*) (*br y*) ⟧
⟹ *interaction-bounded-by consider* (*case-sum l r x*) (*case-sum bl br x*)
by(*cases x*)(*auto*)

lemma *interaction-bounded-by-case-prod* [*interaction-bound*]:
(⋀*a b*. *x* = (*a*, *b*) ⟹ *interaction-bounded-by consider* (*f a b*) (*n a b*))
⟹ *interaction-bounded-by consider* (*case-prod f x*) (*case-prod n x*)
by(*simp split*: *prod.split*)

lemma *interaction-bounded-by-let* [*interaction-bound*]: — This rule unfolds let's
interaction-bounded-by consider (*f t*) *m* ⟹ *interaction-bounded-by consider* (*Let t f*) *m*
by(*simp add*: *Let-def*)

lemma *interaction-bounded-by-map-gpv-id* [*interaction-bound*]:
assumes [*interaction-bound*]: *interaction-bounded-by P gpv n*
shows *interaction-bounded-by P* (*map-gpv f id gpv*) *n*
unfolding *id-def map-gpv-conv-bind* **by** *interaction-bound simp*

abbreviation *interaction-any-bounded-by* :: (*'a*, *'out*, *'in*) *gpv* ⇒ *enat* ⇒ *bool*
where *interaction-any-bounded-by* ≡ *interaction-bounded-by* (λ-. *True*)

lemma *interaction-any-bounded-by-map-gpv'*:
assumes *interaction-any-bounded-by gpv n*
and *surj h*
shows *interaction-any-bounded-by* (*map-gpv' f g h gpv*) *n*
using *assms* **by**(*simp add*: *interaction-bounded-by.simps interaction-bound-map-gpv' o-def*)

4.12 Typing

4.12.1 Interface between gpvs and rpvs / callees

lemma *is-empty-parametric* [*transfer-rule*]: *rel-fun* (*rel-set A*) (=) *Set.is-empty Set.is-empty*
by(*auto simp add*: *rel-fun-def Set.is-empty-def dest*: *rel-setD1 rel-setD2*)

typedef (*'call*, *'ret*) $\mathcal{I}$ = *UNIV* :: (*'call* ⇒ *'ret set*) *set* **..**

setup-lifting *type-definition-*$\mathcal{I}$

lemma *outs-*$\mathcal{I}$*-tparametric*:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *rel-set B*) ===> *rel-set A*) (λ*resps*. {*out*. *resps out* ≠ {}}) (λ*resps*. {*out*. *resps out* ≠ {}})
by(*fold Set.is-empty-def*) *transfer-prover*

lift-definition *outs-*$\mathcal{I}$ :: (*'call*, *'ret*) $\mathcal{I}$ ⇒ *'call set* **is** λ*resps*. {*out*. *resps out* ≠ {}}
parametric *outs-*$\mathcal{I}$*-tparametric* **.**
lift-definition *responses-*$\mathcal{I}$ :: (*'call*, *'ret*) $\mathcal{I}$ ⇒ *'call* ⇒ *'ret set* **is** λ*x*. *x* **parametric** *id-transfer*[*unfolded id-def*] **.**

lift-definition *rel-*$\mathcal{I}$ :: (*'call* ⇒ *'call'* ⇒ *bool*) ⇒ (*'ret* ⇒ *'ret'* ⇒ *bool*) ⇒ (*'call*, *'ret*) $\mathcal{I}$ ⇒ (*'call'*, *'ret'*) $\mathcal{I}$ ⇒ *bool*
is λ*C R resp1 resp2*. *rel-set C* {*out*. *resp1 out* ≠ {}} {*out*. *resp2 out* ≠ {}} ∧ *rel-fun C* (*rel-set R*) *resp1 resp2*
.

lemma *rel-*$\mathcal{I}$*I* [*intro?*]:
⟦ *rel-set C* (*outs-*$\mathcal{I}$ $\mathcal{I}$*1*) (*outs-*$\mathcal{I}$ $\mathcal{I}$*2*); ⋀*x y*. *C x y* ⟹ *rel-set R* (*responses-*$\mathcal{I}$ $\mathcal{I}$*1 x*) (*responses-*$\mathcal{I}$ $\mathcal{I}$*2 y*) ⟧
⟹ *rel-*$\mathcal{I}$ *C R* $\mathcal{I}$*1* $\mathcal{I}$*2*
by *transfer*(*auto simp add*: *rel-fun-def*)

lemma *rel-ℐ-eq* [*relator-eq*]: *rel-ℐ* (=) (=) = (=)
unfolding *fun-eq-iff* **by** *transfer*(*auto simp add*: *relator-eq*)

lemma *rel-ℐ-conversep* [*simp*]: *rel-ℐ* C^{-1-1} R^{-1-1} = (*rel-ℐ C R*)$^{-1-1}$
unfolding *fun-eq-iff conversep-iff*
apply *transfer*
apply(*rewrite* **in** *rel-fun* ⌑ *conversep-iff*[*symmetric*])
apply(*rewrite* **in** *rel-set* ⌑ *conversep-iff*[*symmetric*])
apply(*rewrite* **in** *rel-fun* - ⌑ *conversep-iff*[*symmetric*])
apply(*simp del*: *conversep-iff add*: *rel-fun-conversep*)
apply(*simp*)
done

lemma *rel-ℐ-conversep1-eq* [*simp*]: *rel-ℐ* C^{-1-1} (=) = (*rel-ℐ C* (=))$^{-1-1}$
by(*rewrite* **in** ⌑ = - *conversep-eq*[*symmetric*])(*simp del*: *conversep-eq*)

lemma *rel-ℐ-conversep2-eq* [*simp*]: *rel-ℐ* (=) R^{-1-1} = (*rel-ℐ* (=) *R*)$^{-1-1}$
by(*rewrite* **in** ⌑ = - *conversep-eq*[*symmetric*])(*simp del*: *conversep-eq*)

lemma *responses-ℐ-empty-iff*: *responses-ℐ ℐ out* = {} ⟷ *out* ∉ *outs-ℐ ℐ*
including *ℐ.lifting* **by** *transfer auto*

lemma *in-outs-ℐ-iff-responses-ℐ*: *out* ∈ *outs-ℐ ℐ* ⟷ *responses-ℐ ℐ out* ≠ {}
by(*simp add*: *responses-ℐ-empty-iff*)

lift-definition *ℐ-full* :: ('*call*, '*ret*) *ℐ* **is** λ-. *UNIV* **.**

lemma *ℐ-full-sel* [*simp*]:
  shows *outs-ℐ-full*: *outs-ℐ ℐ-full* = *UNIV*
  and *responses-ℐ-full*: *responses-ℐ ℐ-full x* = *UNIV*
by(*transfer*; *simp*; *fail*)+

context includes *lifting-syntax* **begin**
lemma *outs-ℐ-parametric* [*transfer-rule*]: (*rel-ℐ C R* ===> *rel-set C*) *outs-ℐ outs-ℐ*
unfolding *rel-fun-def* **by** *transfer simp*

lemma *responses-ℐ-parametric* [*transfer-rule*]:
  (*rel-ℐ C R* ===> *C* ===> *rel-set R*) *responses-ℐ responses-ℐ*
unfolding *rel-fun-def* **by** *transfer*(*auto dest*: *rel-funD*)

end

definition *ℐ-trivial* :: ('*out*, '*in*) *ℐ* ⇒ *bool*
where *ℐ-trivial ℐ* ⟷ *outs-ℐ ℐ* = *UNIV*

lemma *ℐ-trivialI* [*intro?*]: (⋀*x*. *x* ∈ *outs-ℐ ℐ*) ⟹ *ℐ-trivial ℐ*
by(*auto simp add*: *ℐ-trivial-def*)

lemma *ℐ-trivialD*: *ℐ-trivial ℐ* ⟹ *outs-ℐ ℐ* = *UNIV*
by(*simp add*: *ℐ-trivial-def*)

lemma *ℐ-trivial-ℐ-full* [*simp*]: *ℐ-trivial ℐ-full*
by(*simp add*: *ℐ-trivial-def*)

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

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

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

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

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

lemma *ℐ-uniform-UNIV* [*simp*]: *ℐ-uniform UNIV UNIV* = *ℐ-full*
  by *transfer simp*

lift-definition *map-ℐ* :: (*'out'* ⇒ *'out*) ⇒ (*'in* ⇒ *'in'*) ⇒ (*'out*, *'in*) *ℐ* ⇒ (*'out'*, *'in'*) *ℐ*
  is *λf g resp x. g ‘ resp* (*f x*) **.**

lemma *outs-ℐ-map-ℐ* [*simp*]:
  outs-ℐ (*map-ℐ f g ℐ*) = *f* −‘ *outs-ℐ ℐ*
  by *transfer simp*

lemma *responses-ℐ-map-ℐ* [*simp*]:
  responses-ℐ (*map-ℐ f g ℐ*) *x* = *g* ‘ *responses-ℐ ℐ* (*f x*)
  by *transfer simp*

lemma *map-ℐ-ℐ-uniform* [*simp*]:
  map-ℐ f g (*ℐ-uniform A B*) = *ℐ-uniform* (*f* −‘ *A*) (*g* ‘ *B*)
  by *transfer*(*auto simp add*: *fun-eq-iff*)

lemma *map-ℐ-id* [*simp*]: *map-ℐ id id ℐ* = *ℐ*
  by *transfer simp*

lemma *map-ℐ-id0*: *map-ℐ id id* = *id*
  by(*simp add*: *fun-eq-iff*)

lemma *map-ℐ-comp* [*simp*]: *map-ℐ f g* (*map-ℐ f' g' ℐ*) = *map-ℐ* (*f'* ∘ *f*) (*g* ∘ *g'*)

$\mathcal{I}$
by *transfer auto*

lemma *map-$\mathcal{I}$-cong*: *map-$\mathcal{I}$ f g $\mathcal{I}$ = map-$\mathcal{I}$ f′ g′ $\mathcal{I}$′*
if $\mathcal{I} = \mathcal{I}'$ **and** *f*: $f = f'$ **and** $\bigwedge x\ y.\ \llbracket\ x \in$ *outs-$\mathcal{I}$ $\mathcal{I}'$*; $y \in$ *responses-$\mathcal{I}$ $\mathcal{I}'$ x* $\rrbracket \Longrightarrow$ $g\ y = g'\ y$
unfolding *that*(*1*,*2*) **using** *that*(*3*−)
by *transfer*(*auto simp add*: *fun-eq-iff intro*!: *image-cong*)

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

functor *map-$\mathcal{I}$* **by**(*simp-all add*: *fun-eq-iff*)

lemma *$\mathcal{I}$-eqI*: $\llbracket$ *outs-$\mathcal{I}$ $\mathcal{I}$ = outs-$\mathcal{I}$ $\mathcal{I}'$*; $\bigwedge x.\ x \in$ *outs-$\mathcal{I}$ $\mathcal{I}'$* $\Longrightarrow$ *responses-$\mathcal{I}$ $\mathcal{I}$ x* = *responses-$\mathcal{I}$ $\mathcal{I}'$ x* $\rrbracket \Longrightarrow \mathcal{I} = \mathcal{I}'$
including *$\mathcal{I}$.lifting* **by** *transfer auto*

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

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

definition *less-$\mathcal{I}$* :: (′*a*, ′*b*) $\mathcal{I} \Rightarrow$ (′*a*, ′*b*) $\mathcal{I} \Rightarrow$ *bool*
where *less-$\mathcal{I}$* = *mk-less* ($\leq$)

instance
proof
show $\mathcal{I} < \mathcal{I}' \longleftrightarrow \mathcal{I} \leq \mathcal{I}' \wedge \neg\, \mathcal{I}' \leq \mathcal{I}$ **for** $\mathcal{I}\ \mathcal{I}'$ :: (′*a*, ′*b*) $\mathcal{I}$ **by**(*simp add*: *less-$\mathcal{I}$-def mk-less-def*)
show $\mathcal{I} \leq \mathcal{I}$ **for** $\mathcal{I}$ :: (′*a*, ′*b*) $\mathcal{I}$ **by**(*simp add*: *le-$\mathcal{I}$-def*)
show $\mathcal{I} \leq \mathcal{I}''$ **if** $\mathcal{I} \leq \mathcal{I}'\ \mathcal{I}' \leq \mathcal{I}''$ **for** $\mathcal{I}\ \mathcal{I}'\ \mathcal{I}''$ :: (′*a*, ′*b*) $\mathcal{I}$ **using** *that*
by(*fastforce simp add*: *le-$\mathcal{I}$-def*)
show $\mathcal{I} = \mathcal{I}'$ **if** $\mathcal{I} \leq \mathcal{I}'\ \mathcal{I}' \leq \mathcal{I}$ **for** $\mathcal{I}\ \mathcal{I}'$ :: (′*a*, ′*b*) $\mathcal{I}$ **using** *that*
by(*auto simp add*: *le-$\mathcal{I}$-def intro*!: *$\mathcal{I}$-eqI*)
qed
end

instantiation $\mathcal{I}$ :: (*type*, *type*) *order-bot* **begin**
definition *bot-$\mathcal{I}$* :: (′*a*, ′*b*) $\mathcal{I}$ **where** *bot-$\mathcal{I}$* = *$\mathcal{I}$-uniform* {} *UNIV*
instance by *standard*(*auto simp add*: *bot-$\mathcal{I}$-def le-$\mathcal{I}$-def*)
end

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

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

by(*simp add*: *bot-$\mathcal{I}$-def*)

lemma *outs-$\mathcal{I}$-mono*: $\mathcal{I} \leq \mathcal{I}' \Longrightarrow$ *outs-$\mathcal{I}$* $\mathcal{I} \subseteq$ *outs-$\mathcal{I}$* $\mathcal{I}'$
by(*simp add*: *le-$\mathcal{I}$-def*)

lemma *responses-$\mathcal{I}$-mono*: $[\![\mathcal{I} \leq \mathcal{I}'; x \in$ *outs-$\mathcal{I}$* $\mathcal{I}]\!] \Longrightarrow$ *responses-$\mathcal{I}$* $\mathcal{I}'$ $x \subseteq$ *responses-$\mathcal{I}$* $\mathcal{I}$ x
by(*simp add*: *le-$\mathcal{I}$-def*)

lemma *$\mathcal{I}$-uniform-empty* [*simp*]: *$\mathcal{I}$-uniform* {} *A* = *bot*
unfolding *bot-$\mathcal{I}$-def* **including** *$\mathcal{I}$.lifting* **by** *transfer simp*

lemma *$\mathcal{I}$-uniform-mono*:
$\mathcal{I}$-uniform A B $\leq$ *$\mathcal{I}$-uniform C D* **if** $A \subseteq C$ $D \subseteq B$ $D = \{\} \longrightarrow B = \{\}$
unfolding *le-$\mathcal{I}$-def* **using** *that* **by** *auto*

context begin
qualified inductive *resultsp-gpv* :: (*'out*, *'in*) $\mathcal{I} \Rightarrow$ *'a* $\Rightarrow$ (*'a*, *'out*, *'in*) *gpv* $\Rightarrow$ *bool*
for Γ *x*
where
Pure: *Pure x* $\in$ *set-spmf* (*the-gpv gpv*) $\Longrightarrow$ *resultsp-gpv* Γ *x gpv*
| *IO*:
$[\![$ *IO out c* $\in$ *set-spmf* (*the-gpv gpv*); *input* $\in$ *responses-$\mathcal{I}$* Γ *out*; *resultsp-gpv* Γ *x* (*c input*) $]\!]$
$\Longrightarrow$ *resultsp-gpv* Γ *x gpv*

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

lemma *resultsp-gpv-results-gpv-eq* [*pred-set-conv*]: *resultsp-gpv* Γ *x gpv* $\longleftrightarrow$ *x* $\in$ *results-gpv* Γ *gpv*
by(*simp add*: *results-gpv-def*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path results-gpv*)›

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

inductive-simps *results-gpv-GPV* [*to-set*, *simp*]: *resultsp-gpv* Γ *x* (*GPV gpv*)

end

lemma *results-gpv-Done* [*iff*]: *results-gpv* Γ (*Done x*) = {*x*}
by(*auto simp add*: *Done.ctr*)

lemma *results-gpv-Fail* [*iff*]: *results-gpv* Γ *Fail* = {}
by(*auto simp add*: *Fail-def*)

lemma *results-gpv-Pause* [*simp*]:
  results-gpv Γ (*Pause out c*) = (⋃ *input*∈*responses-ℐ* Γ *out*. *results-gpv* Γ (*c input*))
by(*auto simp add*: *Pause.ctr*)

lemma *results-gpv-lift-spmf* [*iff*]: *results-gpv* Γ (*lift-spmf p*) = *set-spmf p*
by(*auto simp add*: *lift-spmf.ctr*)

lemma *results-gpv-assert-gpv* [*simp*]: *results-gpv* Γ (*assert-gpv b*) = (*if b then* {()}
else {})
by *auto*

lemma *results-gpv-bind-gpv* [*simp*]:
  results-gpv Γ (*gpv* ≫= *f*) = (⋃ *x*∈*results-gpv* Γ *gpv*. *results-gpv* Γ (*f x*))
  (**is** *?lhs* = *?rhs*)
proof(*intro set-eqI iffI*)
  fix *x*
  assume *x* ∈ *?lhs*
  then show *x* ∈ *?rhs*
  proof(*induction gpv*′≡*gpv* ≫= *f arbitrary*: *gpv*)
    case *Pure* **thus** *?case*
      by(*auto 4 3 split*: *if-split-asm intro*: *results-gpv.intros rev-bexI*)
  next
    case (*IO out c input*)
    from ‹*IO out c* ∈ -›
    obtain *generat* **where** *generat*: *generat* ∈ *set-spmf* (*the-gpv gpv*)
     and ∗: *IO out c* ∈ *set-spmf* (*if is-Pure generat then the-gpv* (*f* (*result generat*))
                                     else return-spmf (*IO* (*output generat*) (λ*input*.
continuation generat input ≫= *f*)))
      by(*auto*)
    thus *?case*
    proof(*cases generat*)
      case (*Pure y*)
      with *generat* **have** *y* ∈ *results-gpv* Γ *gpv* **by**(*auto intro*: *results-gpv.intros*)
      thus *?thesis* **using** ∗ *Pure* ‹*input* ∈ *responses-ℐ* Γ *out*› ‹*x* ∈ *results-gpv* Γ (*c*
input)›
        by(*auto intro*: *results-gpv.IO*)
    next
      case (*IO out*′ *c*′)
      hence [*simp*]: *out*′ = *out*
        and *c*: ⋀*input*. *c input* = *bind-gpv* (*c*′ *input*) *f* **using** ∗ **by** *simp-all*
      from *IO.hyps*(*4*)[*OF c*] **obtain** *y* **where** *y*: *y* ∈ *results-gpv* Γ (*c*′ *input*)

and *x* ∈ *results-gpv* Γ (*f y*) **by** *blast*
from *y IO generat* **have** *y* ∈ *results-gpv* Γ *gpv* **using** ‹*input* ∈ *responses-ℐ* Γ *out*›
by(*auto intro*: *results-gpv.IO*)
with ‹*x* ∈ *results-gpv* Γ (*f y*)› **show** *?thesis* **by** *blast*
qed
qed
next
fix *x*
assume *x* ∈ *?rhs*
then obtain *y* **where** *y*: *y* ∈ *results-gpv* Γ *gpv*
and *x*: *x* ∈ *results-gpv* Γ (*f y*) **by** *blast*
from *y* **show** *x* ∈ *?lhs*
proof(*induction*)
case (*Pure gpv*)
with *x* **show** *?case*
by *cases*(*auto 4 4 intro*: *results-gpv.intros rev-bexI*)
qed(*auto 4 4 intro*: *rev-bexI results-gpv.IO*)
qed

lemma *results-gpv-ℐ-full*: *results-gpv ℐ-full* = *results′-gpv*
proof(*intro ext set-eqI iffI*)
show *x* ∈ *results′-gpv gpv* **if** *x* ∈ *results-gpv ℐ-full gpv* **for** *x gpv*
using *that* **by** *induction*(*auto intro*: *results′-gpvI*)
show *x* ∈ *results-gpv ℐ-full gpv* **if** *x* ∈ *results′-gpv gpv* **for** *x gpv*
using *that* **by** *induction*(*auto intro*: *results-gpv.intros elim*!: *generat.set-cases*)
qed

lemma *results′-bind-gpv* [*simp*]:
results′-gpv (*bind-gpv gpv f*) = (⋃*x*∈*results′-gpv gpv*. *results′-gpv* (*f x*))
unfolding *results-gpv-ℐ-full*[*symmetric*] **by** *simp*

lemma *results-gpv-map-gpv-id* [*simp*]: *results-gpv ℐ* (*map-gpv f id gpv*) = *f* ‘ *results-gpv ℐ gpv*
by(*auto simp add*: *map-gpv-conv-bind id-def*)

lemma *results-gpv-map-gpv-id′* [*simp*]: *results-gpv ℐ* (*map-gpv f* (λ*x*. *x*) *gpv*) = *f* ‘ *results-gpv ℐ gpv*
by(*auto simp add*: *map-gpv-conv-bind id-def*)

lemma *pred-gpv-bind* [*simp*]: *pred-gpv P Q* (*bind-gpv gpv f*) = *pred-gpv* (*pred-gpv P Q* ∘ *f*) *Q gpv*
by(*auto simp add*: *pred-gpv-def outs-bind-gpv*)

lemma *results′-gpv-bind-option* [*simp*]:
results′-gpv (*monad.bind-option Fail x f*) = (⋃*y*∈*set-option x*. *results′-gpv* (*f y*))
by(*cases x*) *simp-all*

lemma *results′-gpv-map-gpv′*:

assumes *surj h*
shows *results′-gpv* (*map-gpv′ f g h gpv*) = *f* ‘ *results′-gpv gpv* (**is** *?lhs* = *?rhs*)
proof −
have ∗:*IO z c* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *x* ∈ *results′-gpv* (*c input*) ⟹
f x ∈ *results′-gpv* (*map-gpv′ f g h* (*c input*)) ⟹ *f x* ∈ *results′-gpv* (*map-gpv′ f g h gpv*) **for** *x z gpv c input*
using *surjD*[*OF assms*, *of input*] **by**(*fastforce intro*: *results′-gpvI elim*!: *generat.set-cases intro*: *rev-image-eqI simp add*: *map-fun-def o-def*)

show *?thesis*
proof(*intro Set.set-eqI iffI*; (*elim imageE*; *hypsubst*)*?*)
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv′≡map-gpv′ f g h gpv arbitrary*: *gpv*)(*fastforce elim*!: *generat.set-cases intro*: *results′-gpvI*)+
show *f x* ∈ *?lhs* **if** *x* ∈ *results′-gpv gpv* **for** *x* **using** *that*
by *induction* (*fastforce intro*: *results′-gpvI elim*!: *generat.set-cases intro*: *rev-image-eqI simp add*: *map-fun-def o-def*
, *clarsimp simp add*: ∗ *elim*!: *generat.set-cases*)
qed
qed

lemma *bind-gpv-bind-option-assoc*:
bind-gpv (*monad.bind-option Fail x f*) *g* = *monad.bind-option Fail x* (λ*x*. *bind-gpv* (*f x*) *g*)
by(*cases x*) *simp-all*

context begin
qualified inductive *outsp-gpv* :: (′*out*, ′*in*) ℐ ⇒ ′*out* ⇒ (′*a*, ′*out*, ′*in*) *gpv* ⇒ *bool*
for ℐ *x* **where**
IO: *IO x c* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *outsp-gpv* ℐ *x gpv*
| *Cont*: ⟦ *IO out rpv* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*ℐ ℐ *out*; *outsp-gpv* ℐ *x* (*rpv input*) ⟧
⟹ *outsp-gpv* ℐ *x gpv*

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

lemma *outsp-gpv-outs-gpv-eq* [*pred-set-conv*]: *outsp-gpv* ℐ *x* = (λ*gpv*. *x* ∈ *outs-gpv* ℐ *gpv*)
by(*simp add*: *outs-gpv-def*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path outs-gpv*)›

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

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

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

end

lemma *outs-gpv-Done* [*iff*]: *outs-gpv* $\mathcal{I}$ (*Done x*) = {}
by(*auto simp add*: *Done.ctr*)

lemma *outs-gpv-Fail* [*iff*]: *outs-gpv* $\mathcal{I}$ *Fail* = {}
by(*auto simp add*: *Fail-def*)

lemma *outs-gpv-Pause* [*simp*]:
outs-gpv $\mathcal{I}$ (*Pause out c*) = *insert out* ($\bigcup$ *input*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *outs-gpv* $\mathcal{I}$ (*c input*))
by(*auto simp add*: *Pause.ctr*)

lemma *outs-gpv-lift-spmf* [*iff*]: *outs-gpv* $\mathcal{I}$ (*lift-spmf p*) = {}
by(*auto simp add*: *lift-spmf.ctr*)

lemma *outs-gpv-assert-gpv* [*simp*]: *outs-gpv* $\mathcal{I}$ (*assert-gpv b*) = {}
by(*cases b*)*auto*

lemma *outs-gpv-bind-gpv* [*simp*]:
outs-gpv $\mathcal{I}$ (*gpv* ≫= *f*) = *outs-gpv* $\mathcal{I}$ *gpv* ∪ ($\bigcup$ *x*∈*results-gpv* $\mathcal{I}$ *gpv*. *outs-gpv* $\mathcal{I}$ (*f x*))
(**is** *?lhs* = *?rhs*)
proof(*intro Set.set-eqI iffI*)
fix *x*
assume *x* ∈ *?lhs*
then show *x* ∈ *?rhs*
proof(*induction gpv*′≡*gpv* ≫= *f arbitrary*: *gpv*)
case *IO* **thus** *?case*
proof(*clarsimp split*: *if-split-asm elim*!: *is-PureE not-is-PureE*, *goal-cases*)
case (*1 generat*)
then show *?case* **by**(*cases generat*)(*auto intro*: *results-gpv.Pure outs-gpv.intros*)
qed
next
case (*Cont out rpv input*)
thus *?case*
proof(*clarsimp split*: *if-split-asm*, *goal-cases*)
case (*1 generat*)
then show *?case* **by**(*cases generat*)(*auto 4 3 split*: *if-split-asm intro*: *results-gpv.intros outs-gpv.intros*)
qed
qed
next

fix *x*
assume $x \in$ *?rhs*
then consider (*out*) $x \in$ *outs-gpv* $\mathcal{I}$ *gpv* | (*result*) *y* **where** $y \in$ *results-gpv* $\mathcal{I}$ *gpv* $x \in$ *outs-gpv* $\mathcal{I}$ (*f y*) **by** *auto*
then show $x \in$ *?lhs*
proof *cases*
case *out* **then show** *?thesis*
by(*induction*) (*auto 4 4 intro*: *outs-gpv.IO outs-gpv.Cont rev-bexI*)
next
case *result* **then show** *?thesis*
by *induction* ((*erule outs-gpv.cases* | *rule outs-gpv.Cont*),
auto 4 4 intro: *outs-gpv.intros rev-bexI elim*: *outs-gpv.cases*)+
qed
qed

lemma *outs-gpv-*$\mathcal{I}$*-full*: *outs-gpv* $\mathcal{I}$*-full* = *outs*′*-gpv*
proof(*intro ext Set.set-eqI iffI*)
show $x \in$ *outs*′*-gpv gpv* **if** $x \in$ *outs-gpv* $\mathcal{I}$*-full gpv* **for** *x gpv*
using *that* **by** *induction*(*auto intro*: *outs*′*-gpvI*)
show $x \in$ *outs-gpv* $\mathcal{I}$*-full gpv* **if** $x \in$ *outs*′*-gpv gpv* **for** *x gpv*
using *that* **by** *induction*(*auto intro*: *outs-gpv.intros elim*!: *generat.set-cases*)
qed

lemma *outs*′*-bind-gpv* [*simp*]:
outs′*-gpv* (*bind-gpv gpv f*) = *outs*′*-gpv gpv* ∪ ($\bigcup$ *x*∈*results*′*-gpv gpv*. *outs*′*-gpv* (*f x*))
unfolding *outs-gpv-*$\mathcal{I}$*-full*[*symmetric*] *results-gpv-*$\mathcal{I}$*-full*[*symmetric*] **by** *simp*

lemma *outs-gpv-map-gpv-id* [*simp*]: *outs-gpv* $\mathcal{I}$ (*map-gpv f id gpv*) = *outs-gpv* $\mathcal{I}$ *gpv*
by(*auto simp add*: *map-gpv-conv-bind id-def*)

lemma *outs-gpv-map-gpv-id*′ [*simp*]: *outs-gpv* $\mathcal{I}$ (*map-gpv f* (λ*x*. *x*) *gpv*) = *outs-gpv* $\mathcal{I}$ *gpv*
by(*auto simp add*: *map-gpv-conv-bind id-def*)

lemma *outs*′*-gpv-bind-option* [*simp*]:
outs′*-gpv* (*monad.bind-option Fail x f*) = ($\bigcup$ *y*∈*set-option x*. *outs*′*-gpv* (*f y*))
by(*cases x*) *simp-all*

lemma *rel-gpv*′′*-Grp*: **includes** *lifting-syntax* **shows**
rel-gpv′′ (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$ =
BNF-Def.Grp {*x*. *results-gpv* ($\mathcal{I}$*-uniform UNIV* (*range h*)) $x \subseteq A \wedge$ *outs-gpv* ($\mathcal{I}$*-uniform UNIV* (*range h*)) $x \subseteq B$} (*map-gpv*′ *f g h*)
(**is** *?lhs* = *?rhs*)
proof(*intro ext GrpI iffI CollectI conjI subsetI*)
let *?*$\mathcal{I}$ = $\mathcal{I}$*-uniform UNIV* (*range h*)
fix *gpv gpv*′

assume ∗: *?lhs gpv gpv′*
then show *map-gpv′ f g h gpv* = *gpv′*
by(*coinduction arbitrary*: *gpv gpv′*)
(*drule rel-gpv′′D*
, *auto 4 5 simp add*: *spmf-rel-map generat.rel-map elim*!: *rel-spmf-mono generat.rel-mono-strong GrpE intro*!: *GrpI dest*: *rel-funD*)
show $x \in A$ **if** $x \in$ *results-gpv ?I gpv* **for** *x* **using** *that* ∗
proof(*induction arbitrary*: *gpv′*)
case (*Pure gpv*)
have *pred-spmf* (*Domainp* (*rel-generat* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) ((*BNF-Def.Grp UNIV h*)$^{-1-1}$ ===> *rel-gpv′′* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$))) (*the-gpv gpv*)
using *Pure.prems*[*THEN rel-gpv′′D*] **unfolding** *spmf-Domainp-rel*[*symmetric*] **..**
with *Pure.hyps* **show** *?case* **by**(*simp add*: *generat.Domainp-rel pred-spmf-def pred-generat-def Domainp-Grp*)
next
case (*IO out c gpv input*)
have *pred-spmf* (*Domainp* (*rel-generat* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) ((*BNF-Def.Grp UNIV h*)$^{-1-1}$ ===> *rel-gpv′′* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$))) (*the-gpv gpv*)
using *IO.prems*[*THEN rel-gpv′′D*] **unfolding** *spmf-Domainp-rel*[*symmetric*] **by**(*rule DomainPI*)
with *IO.hyps* **show** *?case*
by(*auto simp add*: *generat.Domainp-rel pred-spmf-def pred-generat-def Grp-iff dest*: *rel-funD intro*: *IO.IH dest*!: *bspec*)
qed
show $x \in B$ **if** $x \in$ *outs-gpv ?I gpv* **for** *x* **using** *that* ∗
proof(*induction arbitrary*: *gpv′*)
case (*IO c gpv*)
have *pred-spmf* (*Domainp* (*rel-generat* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) ((*BNF-Def.Grp UNIV h*)$^{-1-1}$ ===> *rel-gpv′′* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$))) (*the-gpv gpv*)
using *IO.prems*[*THEN rel-gpv′′D*] **unfolding** *spmf-Domainp-rel*[*symmetric*] **by**(*rule DomainPI*)
with *IO.hyps* **show** *?case* **by**(*simp add*: *generat.Domainp-rel pred-spmf-def pred-generat-def Domainp-Grp*)
next
case (*Cont out rpv gpv input*)
have *pred-spmf* (*Domainp* (*rel-generat* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) ((*BNF-Def.Grp UNIV h*)$^{-1-1}$ ===> *rel-gpv′′* (*BNF-Def.Grp A f*) (*BNF-Def.Grp B g*) (*BNF-Def.Grp UNIV h*)$^{-1-1}$))) (*the-gpv gpv*)
using *Cont.prems*[*THEN rel-gpv′′D*] **unfolding** *spmf-Domainp-rel*[*symmetric*] **by**(*rule DomainPI*)
with *Cont.hyps* **show** *?case*
by(*auto simp add*: *generat.Domainp-rel pred-spmf-def pred-generat-def Grp-iff dest*: *rel-funD intro*: *Cont.IH dest*!: *bspec*)
qed
next

fix *gpv gpv'*
assume *?rhs gpv gpv'*
then have *gpv'*: *gpv' = map-gpv' f g h gpv*
and ∗: *results-gpv* (*ℐ-uniform UNIV* (*range h*)) *gpv* ⊆ *A outs-gpv* (*ℐ-uniform UNIV* (*range h*)) *gpv* ⊆ *B* **by**(*auto simp add*: *Grp-iff*)
show *?lhs gpv gpv'* **using** ∗ **unfolding** *gpv'*
by(*coinduction arbitrary*: *gpv*)
(*fastforce simp add*: *spmf-rel-map generat.rel-map Grp-iff intro*!: *rel-spmf-reflI generat.rel-refl-strong rel-funI elim*!: *generat.set-cases intro*: *results-gpv.intros outs-gpv.intros*)
qed

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

lemma *pred-gpv-conv-pred-gpv'*: *pred-gpv P Q* = *pred-gpv' P Q UNIV*
by(*auto simp add*: *fun-eq-iff pred-gpv-def pred-gpv'.simps results-gpv-ℐ-full outs-gpv-ℐ-full*)

lemma *rel-gpv''-map-gpv'1*:
rel-gpv'' A C (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *gpv gpv'* ⟹ *rel-gpv'' A C* (=) (*map-gpv' id id h gpv*) *gpv'*
apply(*coinduction arbitrary*: *gpv gpv'*)
apply(*drule rel-gpv''D*)
apply(*simp add*: *spmf-rel-map*)
apply(*erule rel-spmf-mono*)
apply(*simp add*: *generat.rel-map*)
apply(*erule generat.rel-mono-strong*; *simp?*)
apply(*subst map-fun2-id*)
by(*auto simp add*: *rel-fun-comp intro*!: *rel-fun-map-fun1 elim*: *rel-fun-mono*)

lemma *rel-gpv''-map-gpv'2*:
rel-gpv'' A C (*eq-on* (*range h*)) *gpv gpv'* ⟹ *rel-gpv'' A C* (*BNF-Def.Grp UNIV h*)$^{-1-1}$ *gpv* (*map-gpv' id id h gpv'*)
apply(*coinduction arbitrary*: *gpv gpv'*)
apply(*drule rel-gpv''D*)
apply(*simp add*: *spmf-rel-map*)
apply(*erule rel-spmf-mono-strong*)
apply(*simp add*: *generat.rel-map*)
apply(*erule generat.rel-mono-strong*; *simp?*)
apply(*subst map-fun-id2-in*)
apply(*rule rel-fun-map-fun2*)
by (*auto simp add*: *rel-fun-comp elim*: *rel-fun-mono*)

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

begin

private lemma *f11*: *Pure x* ∈ *set-spmf* (*the-gpv gpv*) ⟹
Domainp (*rel-generat A C* (*rel-fun R* (*rel-gpv'' A C R*))) (*Pure x*) ⟹ *Domainp A x*
by (*auto simp add*: *pred-generat-def elim*:*bspec dest*: *generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])

private lemma *f21*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟹
rel-generat A C (*rel-fun R* (*rel-gpv'' A C R*)) (*IO out c*) *ba* ⟹ *Domainp C out*
by (*auto simp add*: *pred-generat-def elim*:*bspec dest*: *generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])

private lemma *f12*:
assumes *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *input* ∈ *responses-ℐ* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) *out*
and *x* ∈ *results-gpv* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) (*c input*)
and *Domainp* (*rel-gpv'' A C R*) *gpv*
shows *Domainp* (*rel-gpv'' A C R*) (*c input*)
proof −
obtain *b1* **where** *o1*:*rel-gpv'' A C R gpv b1* **using** *assms*(*4*) **by** *clarsimp*
obtain *b2* **where** *o2*:*rel-generat A C* (*rel-fun R* (*rel-gpv'' A C R*)) (*IO out c*) *b2*
using *assms*(*1*) *o1*[*THEN rel-gpv''D*, *THEN spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-spmf-def* **by** − (*drule* (*1*) *bspec*, *auto*)

have *Ball* (*generat-conts* (*IO out c*)) (*Domainp* (*rel-fun R* (*rel-gpv'' A C R*)))
using *o2*[*THEN generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-generat-def* **by** *simp*

with *assms*(*2*) **show** *?thesis*
apply −
apply(*drule bspec*)
apply *simp*
apply *clarify*
apply(*drule Domainp-rel-fun-le*[*THEN predicate1D*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
by *simp*
qed

private lemma *f22*:
assumes *IO out' rpv* ∈ *set-spmf* (*the-gpv gpv*)
and *input* ∈ *responses-ℐ* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) *out'*
and *out* ∈ *outs-gpv* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) (*rpv input*)
and *Domainp* (*rel-gpv'' A C R*) *gpv*
shows *Domainp* (*rel-gpv'' A C R*) (*rpv input*)
proof −
obtain *b1* **where** *o1*:*rel-gpv'' A C R gpv b1* **using** *assms*(*4*) **by** *auto*

obtain *b2* **where** *o2*:*rel-generat A C* (*rel-fun R* (*rel-gpv'' A C R*)) (*IO out' rpv*) *b2*
using *assms*(*1*) *o1*[*THEN rel-gpv''D*, *THEN spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-spmf-def* **by** − (*drule* (*1*) *bspec*, *auto*)

have *Ball* (*generat-conts* (*IO out' rpv*)) (*Domainp* (*rel-fun R* (*rel-gpv'' A C R*)))
using *o2*[*THEN generat.Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*]]
unfolding *pred-generat-def* **by** *simp*

with *assms*(*2*) **show** *?thesis*
apply −
apply(*drule bspec*)
apply *simp*
apply *clarify*
apply(*drule Domainp-rel-fun-le*[*THEN predicate1D*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
by *simp*
qed

lemma *Domainp-rel-gpv''-le*:
Domainp (*rel-gpv'' A C R*) ≤ *pred-gpv'* (*Domainp A*) (*Domainp C*) {*x*. *Domainp R x*}
proof(*rule predicate1I pred-gpv'.intros*)+
show *Domainp A x* **if** *x* ∈ *results-gpv* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) *gpv Domainp* (*rel-gpv'' A C R*) *gpv* **for** *x gpv* **using** *that*
proof(*induction*)
case (*Pure gpv*)
then show *?case*
by (*clarify*) (*drule rel-gpv''D*
, *auto simp add*: *f11 pred-spmf-def dest*: *spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
qed (*simp add*: *f12*)
show *Domainp C out* **if** *out* ∈ *outs-gpv* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) *gpv Domainp* (*rel-gpv'' A C R*) *gpv* **for** *out gpv* **using** *that*
proof(*induction*)
case (*IO c gpv*)
then show *?case*
by (*clarify*) (*drule rel-gpv''D*
, *auto simp add*: *f21 pred-spmf-def dest*!: *bspec spmf-Domainp-rel*[*THEN fun-cong*, *THEN iffD1*, *OF Domainp-iff*[*THEN iffD2*], *OF exI*])
qed (*simp add*: *f22*)
qed

end

lemma *map-gpv'-id12*: *map-gpv' f g h gpv* = *map-gpv' id id h* (*map-gpv f g gpv*)
unfolding *map-gpv-conv-map-gpv' map-gpv'-comp* **by** *simp*

lemma *rel-gpv''-refl*: ⟦ (=) ≤ A; (=) ≤ C; R ≤ (=) ⟧ ⟹ (=) ≤ *rel-gpv''* A C R
by(*subst rel-gpv''-eq*[*symmetric*])(*rule rel-gpv''-mono*)

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

begin

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

private lemma *f1*: *foo results-gpv results-gpv gpv gpv'* A A' ⟹
x ∈ *set-spmf* (*the-gpv gpv*) ⟹ y ∈ *set-spmf* (*the-gpv gpv'*) ⟹
a ∈ *generat-conts* x ⟹ b ∈ *generat-conts* y ⟹ R' a' α ⟹ R' β b' ⟹
foo results-gpv results-gpv (a a') (b b') A A'
by (*fastforce elim*: *generat.set-cases intro*: *results-gpv.IO*)

private lemma *f2*: *foo outs-gpv outs-gpv gpv gpv'* C C' ⟹
x ∈ *set-spmf* (*the-gpv gpv*) ⟹ y ∈ *set-spmf* (*the-gpv gpv'*) ⟹
a ∈ *generat-conts* x ⟹ b ∈ *generat-conts* y ⟹ R' a' α ⟹ R' β b' ⟹
foo outs-gpv outs-gpv (a a') (b b') C C'
by (*fastforce elim*: *generat.set-cases intro*: *outs-gpv.Cont*)

lemma *rel-gpv''-mono-strong*:
⟦ *rel-gpv''* A C R *gpv gpv'*;
⋀x y. ⟦ x ∈ *results-gpv* (*ℐ-uniform UNIV* {x. *Domainp* R' x}) *gpv*; y ∈ *results-gpv* (*ℐ-uniform UNIV* {x. *Rangep* R' x}) *gpv'*; A x y ⟧ ⟹ A' x y;
⋀x y. ⟦ x ∈ *outs-gpv* (*ℐ-uniform UNIV* {x. *Domainp* R' x}) *gpv*; y ∈ *outs-gpv* (*ℐ-uniform UNIV* {x. *Rangep* R' x}) *gpv'*; C x y ⟧ ⟹ C' x y;
R' ≤ R ⟧
⟹ *rel-gpv''* A' C' R' *gpv gpv'*
apply(*coinduction arbitrary*: *gpv gpv'*)
apply(*drule rel-gpv''D*)
apply(*erule rel-spmf-mono-strong*)
apply(*erule generat.rel-mono-strong*)
apply(*erule generat.set-cases*)+
apply(*erule allE*, *rotate-tac* −*1*)
apply(*erule allE*)
apply(*erule impE*)
apply(*rule results-gpv.Pure*)
apply *simp*
apply(*erule impE*)

apply(*rule results-gpv.Pure*)
apply *simp*
apply *simp*
apply(*erule generat.set-cases*)+
apply(*rotate-tac 1*)
apply(*erule allE*, *rotate-tac −1*)
apply(*erule allE*)
apply(*erule impE*)
apply(*rule outs-gpv.IO*)
apply *simp*
apply(*erule impE*)
apply(*rule outs-gpv.IO*)
apply *simp*
apply *simp*
apply(*erule* (*1*) *rel-fun-mono-strong*)
by (*fastforce simp add*: *f1*[*simplified*] *f2*[*simplified*])

end

lemma *rel-gpv′′-refl-strong*:
assumes ⋀*x*. *x* ∈ *results-gpv* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) *gpv* ⟹ *A x x*
and ⋀*x*. *x* ∈ *outs-gpv* (*ℐ-uniform UNIV* {*x*. *Domainp R x*}) *gpv* ⟹ *C x x*
and *R* ≤ (=)
shows *rel-gpv′′ A C R gpv gpv*
proof −
have *rel-gpv′′* (=) (=) (=) *gpv gpv* **unfolding** *rel-gpv′′-eq* **by** *simp*
then show *?thesis* **using** - - *assms*(*3*) **by**(*rule rel-gpv′′-mono-strong*)(*auto intro*: *assms*(*1*−*2*))
qed

lemma *rel-gpv′′-refl-eq-on*:
⟦ ⋀*x*. *x* ∈ *results-gpv* (*ℐ-uniform UNIV X*) *gpv* ⟹ *A x x*; ⋀*out*. *out* ∈ *outs-gpv* (*ℐ-uniform UNIV X*) *gpv* ⟹ *B out out* ⟧
⟹ *rel-gpv′′ A B* (*eq-on X*) *gpv gpv*
by(*rule rel-gpv′′-refl-strong*) (*auto elim*: *eq-onE*)

lemma *pred-gpv′-mono′* [*mono*]:
pred-gpv′ A C R gpv ⟶ *pred-gpv′ A′ C′ R gpv*
if ⋀*x*. *A x* ⟶ *A′ x* ⋀*x*. *C x* ⟶ *C′ x*
using *that* **unfolding** *pred-gpv′.simps*
by *auto*

4.12.2 Type judgements

coinductive *WT-gpv* :: (*′out*, *′in*) *ℐ* ⇒ (*′a*, *′out*, *′in*) *gpv* ⇒ *bool* (‹((-)/ ⊢*g* (-) √)› [*100*, *0*] *99*)
for Γ
where

($\bigwedge$*out c. IO out c* $\in$ *set-spmf gpv* $\Longrightarrow$ *out* $\in$ *outs-$\mathcal{I}$* Γ $\wedge$ ($\forall$ *input*$\in$*responses-$\mathcal{I}$* Γ *out*. Γ $\vdash$*g c input* $\surd$))
$\Longrightarrow$ Γ $\vdash$*g GPV gpv* $\surd$

lemma *WT-gpv-coinduct* [*consumes 1*, *case-names WT-gpv*, *case-conclusion WT-gpv out cont*, *coinduct pred*: *WT-gpv*]:
assumes $*$: *X gpv*
and *step*: $\bigwedge$*gpv out c*.
⟦ *X gpv*; *IO out c* $\in$ *set-spmf* (*the-gpv gpv*) ⟧
$\Longrightarrow$ *out* $\in$ *outs-$\mathcal{I}$* Γ $\wedge$ ($\forall$ *input* $\in$ *responses-$\mathcal{I}$* Γ *out*. *X* (*c input*) $\vee$ Γ $\vdash$*g c input* $\surd$)
shows Γ $\vdash$*g gpv* $\surd$
using $*$ **by**(*coinduct*)(*auto dest*: *step simp add*: *eq-GPV-iff*)

lemma *WT-gpv-simps*:
Γ $\vdash$*g GPV gpv* $\surd$ $\longleftrightarrow$
($\forall$ *out c*. *IO out c* $\in$ *set-spmf gpv* $\longrightarrow$ *out* $\in$ *outs-$\mathcal{I}$* Γ $\wedge$ ($\forall$ *input*$\in$*responses-$\mathcal{I}$* Γ *out*. Γ $\vdash$*g c input* $\surd$))
by(*subst WT-gpv.simps*) *simp*

lemma *WT-gpvI*:
($\bigwedge$*out c*. *IO out c* $\in$ *set-spmf* (*the-gpv gpv*) $\Longrightarrow$ *out* $\in$ *outs-$\mathcal{I}$* Γ $\wedge$ ($\forall$ *input*$\in$*responses-$\mathcal{I}$* Γ *out*. Γ $\vdash$*g c input* $\surd$))
$\Longrightarrow$ Γ $\vdash$*g gpv* $\surd$
by(*cases gpv*)(*simp add*: *WT-gpv-simps*)

lemma *WT-gpvD*:
assumes Γ $\vdash$*g gpv* $\surd$
shows *WT-gpv-OutD*: *IO out c* $\in$ *set-spmf* (*the-gpv gpv*) $\Longrightarrow$ *out* $\in$ *outs-$\mathcal{I}$* Γ
and *WT-gpv-ContD*: ⟦ *IO out c* $\in$ *set-spmf* (*the-gpv gpv*); *input* $\in$ *responses-$\mathcal{I}$* Γ *out* ⟧ $\Longrightarrow$ Γ $\vdash$*g c input* $\surd$
using *assms* **by**(*cases*, *fastforce*)+

lemma *WT-gpv-mono*:
assumes *WT*: *$\mathcal{I}$1* $\vdash$*g gpv* $\surd$
and *outs*: *outs-$\mathcal{I}$ $\mathcal{I}$1* $\subseteq$ *outs-$\mathcal{I}$ $\mathcal{I}$2*
and *responses*: $\bigwedge$*x*. *x* $\in$ *outs-$\mathcal{I}$ $\mathcal{I}$1* $\Longrightarrow$ *responses-$\mathcal{I}$ $\mathcal{I}$2 x* $\subseteq$ *responses-$\mathcal{I}$ $\mathcal{I}$1 x*
shows *$\mathcal{I}$2* $\vdash$*g gpv* $\surd$
using *WT*
proof *coinduct*
case (*WT-gpv gpv out c*)
with *outs* **show** *?case* **by**(*auto 6 4 dest*: *responses WT-gpvD*)
qed

lemma *WT-gpv-Done* [*iff*]: Γ $\vdash$*g Done x* $\surd$
by(*rule WT-gpvI*) *simp-all*

lemma *WT-gpv-Fail* [*iff*]: Γ $\vdash$*g Fail* $\surd$
by(*rule WT-gpvI*) *simp-all*

lemma *WT-gpv-PauseI*:
  ⟦ *out* ∈ *outs-ℐ* Γ; ⋀*input*. *input* ∈ *responses-ℐ* Γ *out* ⟹ Γ ⊢*g c input* √ ⟧
  ⟹ Γ ⊢*g Pause out c* √
by(*rule WT-gpvI*) *simp-all*

lemma *WT-gpv-Pause* [*iff*]:
  Γ ⊢*g Pause out c* √ ⟷ *out* ∈ *outs-ℐ* Γ ∧ (∀ *input* ∈ *responses-ℐ* Γ *out*. Γ ⊢*g c input* √)
by(*auto intro*: *WT-gpv-PauseI dest*: *WT-gpvD*)

lemma *WT-gpv-bindI*:
  ⟦ Γ ⊢*g gpv* √; ⋀*x*. *x* ∈ *results-gpv* Γ *gpv* ⟹ Γ ⊢*g f x* √ ⟧
  ⟹ Γ ⊢*g gpv* ≫= *f* √
proof(*coinduction arbitrary*: *gpv*)
  case [*rule-format*]: (*WT-gpv out c gpv*)
  from ‹*IO out c* ∈ -›
  obtain *generat* **where** *generat*: *generat* ∈ *set-spmf* (*the-gpv gpv*)
    and ∗: *IO out c* ∈ *set-spmf* (*if is-Pure generat then the-gpv* (*f* (*result generat*))
                          else return-spmf (*IO* (*output generat*) (λ*input*. *continuation generat input* ≫= *f*)))
    by(*auto*)
  show *?case*
  proof(*cases generat*)
    case (*Pure y*)
    with *generat* **have** *y* ∈ *results-gpv* Γ *gpv* **by**(*auto intro*: *results-gpv.Pure*)
    hence Γ ⊢*g f y* √ **by**(*rule WT-gpv*)
    with ∗ *Pure* **show** *?thesis* **by**(*auto dest*: *WT-gpvD*)
  next
    case (*IO out′ c′*)
    hence [*simp*]: *out′* = *out*
      and *c*: ⋀*input*. *c input* = *bind-gpv* (*c′ input*) *f* **using** ∗ **by** *simp-all*
    from *generat IO* **have** ∗∗: *IO out c′* ∈ *set-spmf* (*the-gpv gpv*) **by** *simp*
    with ‹Γ ⊢*g gpv* √› **have** *?out* **by**(*auto dest*: *WT-gpvD*)
    moreover **{**
      fix *input*
      assume *input*: *input* ∈ *responses-ℐ* Γ *out*
      with ‹Γ ⊢*g gpv* √› ∗∗ **have** Γ ⊢*g c′ input* √ **by**(*rule WT-gpvD*)
      moreover **{**
        fix *y*
        assume *y* ∈ *results-gpv* Γ (*c′ input*)
        with ∗∗ *input* **have** *y* ∈ *results-gpv* Γ *gpv* **by**(*rule results-gpv.IO*)
        hence Γ ⊢*g f y* √ **by**(*rule WT-gpv*) **}**
      moreover note *calculation* **}**
    hence *?cont* **using** *c* **by** *blast*
    ultimately show *?thesis* **..**
  qed
qed

lemma *WT-gpv-bindD2*:
assumes *WT*: Γ ⊢g *gpv* ≫= *f* √
and *x*: *x* ∈ *results-gpv* Γ *gpv*
shows Γ ⊢g *f x* √
using *x WT*
proof *induction*
case (*Pure gpv*)
show *?case*
proof(*rule WT-gpvI*)
fix *out c*
assume *IO out c* ∈ *set-spmf* (*the-gpv* (*f x*))
with *Pure* **have** *IO out c* ∈ *set-spmf* (*the-gpv* (*gpv* ≫= *f*)) **by**(*auto intro*: *rev-bexI*)
with ‹Γ ⊢g *gpv* ≫= *f* √› **show** *out* ∈ *outs-ℐ* Γ ∧ (∀ *input*∈*responses-ℐ* Γ *out*. Γ ⊢g *c input* √)
by(*auto dest*: *WT-gpvD simp del*: *set-bind-spmf*)
qed
next
case (*IO out c gpv input*)
from ‹*IO out c* ∈ -›
have *IO out* (λ*input*. *bind-gpv* (*c input*) *f*) ∈ *set-spmf* (*the-gpv* (*gpv* ≫= *f*))
by(*auto intro*: *rev-bexI*)
with *IO.prems* **have** Γ ⊢g *c input* ≫= *f* √ **using** ‹*input* ∈ -› **by**(*rule WT-gpv-ContD*)
thus *?case* **by**(*rule IO.IH*)
qed

lemma *WT-gpv-bindD1*: Γ ⊢g *gpv* ≫= *f* √ ⟹ Γ ⊢g *gpv* √
proof(*coinduction arbitrary*: *gpv*)
case (*WT-gpv out c gpv*)
from ‹*IO out c* ∈ -›
have *IO out* (λ*input*. *bind-gpv* (*c input*) *f*) ∈ *set-spmf* (*the-gpv* (*gpv* ≫= *f*))
by(*auto intro*: *rev-bexI*)
with ‹Γ ⊢g *gpv* ≫= *f* √› **show** *?case*
by(*auto simp del*: *bind-gpv-sel′ dest*: *WT-gpvD*)
qed

lemma *WT-gpv-bind* [*simp*]: Γ ⊢g *gpv* ≫= *f* √ ⟷ Γ ⊢g *gpv* √ ∧ (∀ *x*∈*results-gpv* Γ *gpv*. Γ ⊢g *f x* √)
by(*blast intro*: *WT-gpv-bindI dest*: *WT-gpv-bindD1 WT-gpv-bindD2*)

lemma *WT-gpv-full* [*simp*, *intro*!]: *ℐ-full* ⊢g *gpv* √
by(*coinduction arbitrary*: *gpv*)(*auto*)

lemma *WT-gpv-lift-spmf* [*simp*, *intro*!]: ℐ ⊢g *lift-spmf p* √
by(*rule WT-gpvI*) *auto*

lemma *WT-gpv-coinduct-bind* [*consumes 1*, *case-names WT-gpv*, *case-conclusion WT-gpv out cont*]:
assumes ∗: *X gpv*

and *step*: $\bigwedge$*gpv out c*. ⟦ *X gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟧
⟹ *out* ∈ *outs-ℐ ℐ* ∧ (∀ *input*∈*responses-ℐ ℐ out*.
X (*c input*) ∨
ℐ ⊢*g c input* √ ∨
(∃ (*gpv'* :: ('*b*, '*call*, '*ret*) *gpv*) *f*. *c input* = *gpv'* ≫= *f* ∧ *ℐ* ⊢*g gpv'* √ ∧ (∀ *x*∈*results-gpv ℐ gpv'*. *X* (*f x*))))
shows *ℐ* ⊢*g gpv* √
proof −
fix *x*
define *gpv'* :: ('*b*, '*call*, '*ret*) *gpv* **and** *f* :: '*b* ⇒ ('*a*, '*call*, '*ret*) *gpv*
where *gpv'* = *Done x* **and** *f* = (λ-. *gpv*)
with ∗ **have** *ℐ* ⊢*g gpv'* √ **and** $\bigwedge$*x*. *x* ∈ *results-gpv ℐ gpv'* ⟹ *X* (*f x*) **by** *simp-all*
then have *ℐ* ⊢*g gpv'* ≫= *f* √
proof(*coinduction arbitrary*: *gpv' f rule*: *WT-gpv-coinduct*)
case [*rule-format*]: (*WT-gpv out c gpv'*)
from ‹*IO out c* ∈ -›
obtain *generat* **where** *generat*: *generat* ∈ *set-spmf* (*the-gpv gpv'*)
and ∗: *IO out c* ∈ *set-spmf* (*if is-Pure generat*
then the-gpv (*f* (*result generat*))
else return-spmf (*IO* (*output generat*) (λ*input*. *continuation generat input* ≫= *f*)))
by(*clarsimp*)
show *?case*
proof(*cases generat*)
case (*Pure x*)
from *Pure* ∗ **have** *IO*: *IO out c* ∈ *set-spmf* (*the-gpv* (*f x*)) **by** *simp*
from *generat Pure* **have** *x* ∈ *results-gpv ℐ gpv'* **by** (*simp add*: *results-gpv.Pure*)
then have *X* (*f x*) **by**(*rule WT-gpv*)
from *step*[*OF this IO*] **show** *?thesis* **by**(*auto 4 4 intro*: *exI*[**where** *x*=*Done* -])
next
case (*IO out' c'*)
with ∗ **have** [*simp*]: *out'* = *out*
and *c*: *c* = (λ*input*. *c' input* ≫= *f*) **by** *simp-all*
from *IO generat* **have** *IO*: *IO out c'* ∈ *set-spmf* (*the-gpv gpv'*) **by** *simp*
then have $\bigwedge$*input*. *input* ∈ *responses-ℐ ℐ out* ⟹ *results-gpv ℐ* (*c' input*) ⊆ *results-gpv ℐ gpv'*
by(*auto intro*: *results-gpv.IO*)
with *WT-gpvD*[*OF* ‹*ℐ* ⊢*g gpv'* √› *IO*] **show** *?thesis* **unfolding** *c* **using** *WT-gpv*(*2*) **by** *blast*
qed
qed
then show *?thesis* **unfolding** *gpv'-def f-def* **by** *simp*
qed

lemma *ℐ-trivial-WT-gpvD* [*simp*]: *ℐ-trivial ℐ* ⟹ *ℐ* ⊢*g gpv* √
using *WT-gpv-full* **by**(*rule WT-gpv-mono*)(*simp-all add*: *ℐ-trivial-def*)

lemma *ℐ-trivial-WT-gpvI*:

assumes $\bigwedge$*gpv* :: (*'a*, *'out*, *'in*) *gpv*. $\mathcal{I}$ ⊢*g gpv* √
shows $\mathcal{I}$*-trivial* $\mathcal{I}$
proof
fix *x*
have $\mathcal{I}$ ⊢*g Pause x* (λ-. *Fail* :: (*'a*, *'out*, *'in*) *gpv*) √ **by**(*rule assms*)
thus *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*simp*)
qed

lemma *WT-gpv-*$\mathcal{I}$*-mono*: ⟦ $\mathcal{I}$ ⊢*g gpv* √; $\mathcal{I}$ ≤ $\mathcal{I}'$ ⟧ ⟹ $\mathcal{I}'$ ⊢*g gpv* √
by(*erule WT-gpv-mono*; *rule outs-*$\mathcal{I}$*-mono responses-*$\mathcal{I}$*-mono*)

lemma *results-gpv-mono*:
assumes *le*: $\mathcal{I}'$ ≤ $\mathcal{I}$ **and** *WT*: $\mathcal{I}'$ ⊢*g gpv* √
shows *results-gpv* $\mathcal{I}$ *gpv* ⊆ *results-gpv* $\mathcal{I}'$ *gpv*
proof(*rule subsetI*, *goal-cases*)
case (*1 x*)
show *?case* **using** *1 WT* **by**(*induction*)(*auto 4 3 intro*: *results-gpv.intros responses-*$\mathcal{I}$*-mono*[*OF le*, *THEN subsetD*] *intro*: *WT-gpvD*)
qed

lemma *WT-gpv-outs-gpv*:
assumes $\mathcal{I}$ ⊢*g gpv* √
shows *outs-gpv* $\mathcal{I}$ *gpv* ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}$
proof
show *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **if** *x* ∈ *outs-gpv* $\mathcal{I}$ *gpv* **for** *x* **using** *that assms*
by(*induction*)(*blast intro*: *WT-gpv-OutD WT-gpv-ContD*)+
qed

lemma *WT-gpv-map-gpv'*: $\mathcal{I}$ ⊢*g map-gpv' f g h gpv* √ **if** *map-*$\mathcal{I}$ *g h* $\mathcal{I}$ ⊢*g gpv* √
using *that* **by**(*coinduction arbitrary*: *gpv*)(*auto 4 4 dest*: *WT-gpvD*)

lemma *WT-gpv-map-gpv*: $\mathcal{I}$ ⊢*g map-gpv f g gpv* √ **if** *map-*$\mathcal{I}$ *g id* $\mathcal{I}$ ⊢*g gpv* √
unfolding *map-gpv-conv-map-gpv'* **using** *that* **by**(*rule WT-gpv-map-gpv'*)

lemma *results-gpv-map-gpv'* [*simp*]:
results-gpv $\mathcal{I}$ (*map-gpv' f g h gpv*) = *f* ' (*results-gpv* (*map-*$\mathcal{I}$ *g h* $\mathcal{I}$) *gpv*)
proof(*intro Set.set-eqI iffI*; (*elim imageE*; *hypsubst*)*?*)
show *x* ∈ *f* ' *results-gpv* (*map-*$\mathcal{I}$ *g h* $\mathcal{I}$) *gpv* **if** *x* ∈ *results-gpv* $\mathcal{I}$ (*map-gpv' f g h gpv*) **for** *x* **using** *that*
by(*induction gpv'≡map-gpv' f g h gpv arbitrary*: *gpv*)(*fastforce intro*: *results-gpv.intros rev-image-eqI*)+
show *f x* ∈ *results-gpv* $\mathcal{I}$ (*map-gpv' f g h gpv*) **if** *x* ∈ *results-gpv* (*map-*$\mathcal{I}$ *g h* $\mathcal{I}$) *gpv* **for** *x* **using** *that*
by(*induction*)(*fastforce intro*: *results-gpv.intros*)+
qed

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

note [*transfer-rule*] = *the-gpv-parametric′*
show ∗: *ℐ* ⊢*g gpv* √ **if** [*transfer-rule*]: *rel-ℐ C R ℐ ℐ′ bi-unique C*
and ∗: *ℐ′* ⊢*g gpv′* √ *rel-gpv″ A C R gpv gpv′* **for** *ℐ ℐ′ gpv gpv′ A C R*
using ∗
proof(*coinduction arbitrary*: *gpv gpv′*)
case (*WT-gpv out c gpv gpv′*)
note [*transfer-rule*] = *WT-gpv*(*2*)
have *rel-set* (*rel-generat A C* (*R* ===> *rel-gpv″ A C R*)) (*set-spmf* (*the-gpv gpv*)) (*set-spmf* (*the-gpv gpv′*))
by *transfer-prover*
from *rel-setD1*[*OF this WT-gpv*(*3*)] **obtain** *out′ c′*
where [*transfer-rule*]: *C out out′* (*R* ===> *rel-gpv″ A C R*) *c c′*
and *out′*: *IO out′ c′* ∈ *set-spmf* (*the-gpv gpv′*)
by(*auto elim*: *generat.rel-cases*)
have *out* ∈ *outs-ℐ ℐ* ⟷ *out′* ∈ *outs-ℐ ℐ′* **by** *transfer-prover*
with *WT-gpvD*(*1*)[*OF WT-gpv*(*1*) *out′*] **have** *?out* **by** *simp*
moreover have *?cont*
proof(*standard*; *goal-cases cont*)
case (*cont input*)
have *rel-set R* (*responses-ℐ ℐ out*) (*responses-ℐ ℐ′ out′*) **by** *transfer-prover*
from *rel-setD1*[*OF this cont*] **obtain** *input′* **where** [*transfer-rule*]: *R input input′*
and *input′*: *input′* ∈ *responses-ℐ ℐ′ out′* **by** *blast*
have *rel-gpv″ A C R* (*c input*) (*c′ input′*) **by** *transfer-prover*
with *WT-gpvD*(*2*)[*OF WT-gpv*(*1*) *out′ input′*] **show** *?case* **by** *auto*
qed
ultimately show *?case* **..**
qed

show *ℐ′* ⊢*g gpv′* √ **if** *rel-ℐ C R ℐ ℐ′ bi-unique C ℐ* ⊢*g gpv* √ *rel-gpv″ A C R gpv gpv′*
for *ℐ ℐ′ gpv gpv′*
using ∗[*of conversep C conversep R ℐ′ ℐ gpv conversep A gpv′*] *that*
by(*simp add*: *rel-gpv″-conversep*)
qed

lemma *WT-gpv-map-gpv-id* [*simp*]: *ℐ* ⊢*g map-gpv f id gpv* √ ⟷ *ℐ* ⊢*g gpv* √
using *WT-gpv-parametric′*[*of BNF-Def.Grp UNIV id* (=) *BNF-Def.Grp UNIV f*, *folded rel-gpv-conv-rel-gpv″*]
unfolding *gpv.rel-Grp* **unfolding** *eq-alt*[*symmetric*] *relator-eq*
by(*auto simp add*: *rel-fun-def Grp-def bi-unique-eq*)

lemma *WT-gpv-outs-gpvI*:
assumes *outs-gpv ℐ gpv* ⊆ *outs-ℐ ℐ*
shows *ℐ* ⊢*g gpv* √
using *assms* **by**(*coinduction arbitrary*: *gpv*)(*auto intro*: *outs-gpv.intros*)

lemma *WT-gpv-iff-outs-gpv*:
ℐ ⊢*g gpv* √ ⟷ *outs-gpv ℐ gpv* ⊆ *outs-ℐ ℐ*

by(*blast intro*: *WT-gpv-outs-gpvI dest*: *WT-gpv-outs-gpv*)

4.13 Sub-gpvs

context begin
qualified inductive *sub-gpvsp* :: ($'out$, $'in$) $\mathcal{I}$ ⇒ ($'a$, $'out$, $'in$) *gpv* ⇒ ($'a$, $'out$, $'in$) *gpv* ⇒ *bool*
for $\mathcal{I}$ *x*
where
base:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*; *x* = *c input* ⟧
⟹ *sub-gpvsp* $\mathcal{I}$ *x gpv*
| *cont*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*; *sub-gpvsp* $\mathcal{I}$ *x* (*c input*) ⟧
⟹ *sub-gpvsp* $\mathcal{I}$ *x gpv*

qualified lemma *sub-gpvsp-base*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* ⟧
⟹ *sub-gpvsp* $\mathcal{I}$ (*c input*) *gpv*
by(*rule base*) *simp-all*

definition *sub-gpvs* :: ($'out$, $'in$) $\mathcal{I}$ ⇒ ($'a$, $'out$, $'in$) *gpv* ⇒ ($'a$, $'out$, $'in$) *gpv set*
where *sub-gpvs* $\mathcal{I}$ *gpv* ≡ {*x*. *sub-gpvsp* $\mathcal{I}$ *x gpv*}

lemma *sub-gpvsp-sub-gpvs-eq* [*pred-set-conv*]: *sub-gpvsp* $\mathcal{I}$ *x gpv* ⟷ *x* ∈ *sub-gpvs* $\mathcal{I}$ *gpv*
by(*simp add*: *sub-gpvs-def*)

context begin
local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path sub-gpvs*)›

lemmas *intros* [*intro?*] = *sub-gpvsp.intros*[*to-set*]
and *base* = *sub-gpvsp-base*[*to-set*]
and *cont* = *cont*[*to-set*]
and *induct* [*consumes 1*, *case-names Pure IO*, *induct set*: *sub-gpvs*] = *sub-gpvsp.induct*[*to-set*]
and *cases* [*consumes 1*, *case-names Pure IO*, *cases set*: *sub-gpvs*] = *sub-gpvsp.cases*[*to-set*]
and *simps* = *sub-gpvsp.simps*[*to-set*]
end
end

lemma *WT-sub-gpvsD*:
assumes $\mathcal{I}$ ⊢*g gpv* √ **and** *gpv'* ∈ *sub-gpvs* $\mathcal{I}$ *gpv*
shows $\mathcal{I}$ ⊢*g gpv'* √
using *assms*(*2*,*1*) **by**(*induction*)(*auto dest*: *WT-gpvD*)

lemma *WT-sub-gpvsI*:
⟦ ⋀*out c*. *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *out* ∈ *outs-*$\mathcal{I}$ Γ;

⋀*gpv′*. *gpv′* ∈ *sub-gpvs* Γ *gpv* ⟹ Γ ⊢*g gpv′* √ ⟧
⟹ Γ ⊢*g gpv* √
by(*rule WT-gpvI*)(*auto intro*: *sub-gpvs.base*)

4.14 Losslessness

A gpv is lossless iff we are guaranteed to get a result after a finite number of interactions that respect the interface. It is colossless if the interactions may go on for ever, but there is no non-termination.

We define both notions of losslessness simultaneously by mimicking what the (co)inductive package would do internally. Thus, we get a constant which is parametrised by the choice of the fixpoint, i.e., for non-recursive gpvs, we can state and prove both versions of losslessness in one go.

context
fixes *co* :: *bool* **and** ℐ :: (*′out*, *′in*) ℐ
and *F* :: ((*′a*, *′out*, *′in*) *gpv* ⇒ *bool*) ⇒ ((*′a*, *′out*, *′in*) *gpv* ⇒ *bool*)
and *co′* :: *bool*
defines *F* ≡ λ*gen-lossless-gpv gpv*. ∃ *pa*. *gpv* = *GPV pa* ∧
lossless-spmf pa ∧ (∀ *out c input*. *IO out c* ∈ *set-spmf pa* ⟶ *input* ∈ *responses-ℐ* ℐ *out* ⟶ *gen-lossless-gpv* (*c input*))
and *co′* ≡ *co* — We use a copy of *co* such that we can do case distinctions on *co′* without the simplifier rewriting the *co* in the local abbreviations for the constants.
begin

lemma *gen-lossless-gpv-mono*: *mono F*
unfolding *F-def*
apply(*rule monoI le-funI le-boolI′*)+
apply(*tactic* ‹*REPEAT* (*resolve-tac* @{*context*} (*Inductive.get-monos* @{*context*}) *1*)›)
apply(*erule le-funE*)
apply(*erule le-boolD*)
done

definition *gen-lossless-gpv* :: (*′a*, *′out*, *′in*) *gpv* ⇒ *bool*
where *gen-lossless-gpv* = (*if co′ then gfp else lfp*) *F*

lemma *gen-lossless-gpv-unfold*: *gen-lossless-gpv* = *F gen-lossless-gpv*
by(*simp add*: *gen-lossless-gpv-def gfp-unfold*[*OF gen-lossless-gpv-mono*, *symmetric*] *lfp-unfold*[*OF gen-lossless-gpv-mono*, *symmetric*])

lemma *gen-lossless-gpv-True*: *co′* = *True* ⟹ *gen-lossless-gpv* ≡ *gfp F*
and *gen-lossless-gpv-False*: *co′* = *False* ⟹ *gen-lossless-gpv* ≡ *lfp F*
by(*simp-all add*: *gen-lossless-gpv-def*)

lemma *gen-lossless-gpv-cases* [*elim?*, *cases pred*]:
assumes *gen-lossless-gpv gpv*
obtains (*gen-lossless-gpv*) *p* **where** *gpv* = *GPV p lossless-spmf p*

⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *input* ∈ *responses-ℐ ℐ out*⟧ ⟹ *gen-lossless-gpv* (*c input*)
proof −
from *assms* **show** *?thesis*
by(*rewrite* **in** *asm gen-lossless-gpv-unfold*)(*auto simp add*: *F-def intro*: *that*)
qed

lemma *gen-lossless-gpvD*:
assumes *gen-lossless-gpv gpv*
shows *gen-lossless-gpv-lossless-spmfD*: *lossless-spmf* (*the-gpv gpv*)
and *gen-lossless-gpv-continuationD*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-ℐ ℐ out* ⟧ ⟹ *gen-lossless-gpv* (*c input*)
using *assms* **by**(*auto elim*: *gen-lossless-gpv-cases*)

lemma *gen-lossless-gpv-intros*:
⟦ *lossless-spmf p*;
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *input* ∈ *responses-ℐ ℐ out* ⟧ ⟹ *gen-lossless-gpv* (*c input*) ⟧
⟹ *gen-lossless-gpv* (*GPV p*)
by(*rewrite gen-lossless-gpv-unfold*)(*simp add*: *F-def*)

lemma *gen-lossless-gpvI* [*intro?*]:
⟦ *lossless-spmf* (*the-gpv gpv*);
⋀*out c input*. ⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-ℐ ℐ out* ⟧
⟹ *gen-lossless-gpv* (*c input*) ⟧
⟹ *gen-lossless-gpv gpv*
by(*cases gpv*)(*auto intro*: *gen-lossless-gpv-intros*)

lemma *gen-lossless-gpv-simps*:
gen-lossless-gpv gpv ⟷
(∃ *p*. *gpv* = *GPV p* ∧ *lossless-spmf p* ∧ (∀ *out c input*.
IO out c ∈ *set-spmf p* ⟶ *input* ∈ *responses-ℐ ℐ out* ⟶ *gen-lossless-gpv* (*c input*)))
by(*rewrite gen-lossless-gpv-unfold*)(*simp add*: *F-def*)

lemma *gen-lossless-gpv-Done* [*iff*]: *gen-lossless-gpv* (*Done x*)
by(*rule gen-lossless-gpvI*) *auto*

lemma *gen-lossless-gpv-Fail* [*iff*]: ¬ *gen-lossless-gpv Fail*
by(*auto dest*: *gen-lossless-gpvD*)

lemma *gen-lossless-gpv-Pause* [*simp*]:
gen-lossless-gpv (*Pause out c*) ⟷ (∀ *input* ∈ *responses-ℐ ℐ out*. *gen-lossless-gpv* (*c input*))
by(*auto dest*: *gen-lossless-gpvD intro*: *gen-lossless-gpvI*)

lemma *gen-lossless-gpv-lift-spmf* [*iff*]: *gen-lossless-gpv* (*lift-spmf p*) ⟷ *lossless-spmf p*

by(*auto dest*: *gen-lossless-gpvD intro*: *gen-lossless-gpvI*)

end

lemma *gen-lossless-gpv-assert-gpv* [*iff*]: *gen-lossless-gpv co* $\mathcal{I}$ (*assert-gpv b*) $\longleftrightarrow$ *b*
by(*cases b*) *simp-all*

abbreviation *lossless-gpv* :: ('*out*, '*in*) $\mathcal{I}$ $\Rightarrow$ ('*a*, '*out*, '*in*) *gpv* $\Rightarrow$ *bool*
where *lossless-gpv* $\equiv$ *gen-lossless-gpv False*

abbreviation *colossless-gpv* :: ('*out*, '*in*) $\mathcal{I}$ $\Rightarrow$ ('*a*, '*out*, '*in*) *gpv* $\Rightarrow$ *bool*
where *colossless-gpv* $\equiv$ *gen-lossless-gpv True*

lemma *lossless-gpv-induct* [*consumes 1*, *case-names lossless-gpv*, *induct pred*]:
assumes $*$: *lossless-gpv* $\mathcal{I}$ *gpv*
and *step*: $\bigwedge$*p*. ⟦ *lossless-spmf p*;
$\bigwedge$*out c input*. ⟦*IO out c* $\in$ *set-spmf p*; *input* $\in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*⟧ $\Longrightarrow$ *lossless-gpv* $\mathcal{I}$ (*c input*);
$\bigwedge$*out c input*. ⟦*IO out c* $\in$ *set-spmf p*; *input* $\in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*⟧ $\Longrightarrow$ *P* (*c input*) ⟧
$\Longrightarrow$ *P* (*GPV p*)
shows *P gpv*
proof −
have *lossless-gpv* $\mathcal{I}$ $\leq$ *P*
by(*rule def-lfp-induct*[*OF gen-lossless-gpv-False gen-lossless-gpv-mono*])(*auto intro*!: *le-funI step*)
then show *?thesis* **using** $*$ **by** *auto*
qed

lemma *colossless-gpv-coinduct*
[*consumes 1*, *case-names colossless-gpv*, *case-conclusion colossless-gpv lossless-spmf continuation*, *coinduct pred*]:
assumes $*$: *X gpv*
and *step*: $\bigwedge$*gpv*. *X gpv* $\Longrightarrow$ *lossless-spmf* (*the-gpv gpv*) $\wedge$ ($\forall$ *out c input*.
IO out c $\in$ *set-spmf* (*the-gpv gpv*) $\longrightarrow$ *input* $\in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* $\longrightarrow$ *X* (*c input*) $\vee$ *colossless-gpv* $\mathcal{I}$ (*c input*))
shows *colossless-gpv* $\mathcal{I}$ *gpv*
proof −
have *X* $\leq$ *colossless-gpv* $\mathcal{I}$
by(*rule def-coinduct*[*OF gen-lossless-gpv-True gen-lossless-gpv-mono*])
(*auto 4 4 intro*!: *le-funI dest*!: *step intro*: *exI*[**where** *x*=*the-gpv* -])
then show *?thesis* **using** $*$ **by** *auto*
qed

lemmas *lossless-gpvI* = *gen-lossless-gpvI*[**where** *co*=*False*]
and *lossless-gpvD* = *gen-lossless-gpvD*[**where** *co*=*False*]
and *lossless-gpv-lossless-spmfD* = *gen-lossless-gpv-lossless-spmfD*[**where** *co*=*False*]
and *lossless-gpv-continuationD* = *gen-lossless-gpv-continuationD*[**where** *co*=*False*]

lemmas *colossless-gpvI* = *gen-lossless-gpvI*[**where** *co*=*True*]
and *colossless-gpvD* = *gen-lossless-gpvD*[**where** *co*=*True*]
and *colossless-gpv-lossless-spmfD* = *gen-lossless-gpv-lossless-spmfD*[**where** *co*=*True*]
and *colossless-gpv-continuationD* = *gen-lossless-gpv-continuationD*[**where** *co*=*True*]

lemma *gen-lossless-bind-gpvI*:
assumes *gen-lossless-gpv co* $\mathcal{I}$ *gpv* $\bigwedge x.\ x \in$ *results-gpv* $\mathcal{I}$ *gpv* $\Longrightarrow$ *gen-lossless-gpv co* $\mathcal{I}$ (*f x*)
shows *gen-lossless-gpv co* $\mathcal{I}$ (*gpv* $\gg=$ *f*)
proof(*cases co*)
case *False*
hence *eq*: *co* = *False* **by** *simp*
show *?thesis* **using** *assms* **unfolding** *eq*
proof(*induction*)
case (*lossless-gpv p*)
{ fix *x*
assume *Pure x* $\in$ *set-spmf p*
hence *x* $\in$ *results-gpv* $\mathcal{I}$ (*GPV p*) **by** *simp*
hence *lossless-gpv* $\mathcal{I}$ (*f x*) **by**(*rule lossless-gpv.prems*) **}**
with ‹*lossless-spmf p*› **show** *?case* **unfolding** *GPV-bind*
apply(*intro gen-lossless-gpv-intros*)
apply(*fastforce dest*: *lossless-gpvD split*: *generat.split*)
apply(*clarsimp*; *split generat.split-asm*)
apply(*auto dest*: *lossless-gpvD intro*!: *lossless-gpv*)
done
qed
next
case *True*
hence *eq*: *co* = *True* **by** *simp*
show *?thesis* **using** *assms* **unfolding** *eq*
proof(*coinduction arbitrary*: *gpv rule*: *colossless-gpv-coinduct*)
case ∗ [*rule-format*]: (*colossless-gpv gpv*)
from ∗(*1*) **have** *?lossless-spmf*
by(*auto 4 3 dest*: *colossless-gpv-lossless-spmfD elim*!: *is-PureE intro*: ∗(*2*)[*THEN colossless-gpv-lossless-spmfD*] *results-gpv.Pure*)
moreover have *?continuation*
proof(*intro strip*)
fix *out c input*
assume *IO*: *IO out c* $\in$ *set-spmf* (*the-gpv* (*gpv* $\gg=$ *f*))
and *input*: *input* $\in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*
from *IO* **obtain** *generat* **where** *generat*: *generat* $\in$ *set-spmf* (*the-gpv gpv*)
and *IO*: *IO out c* $\in$ *set-spmf* (*if is-Pure generat then the-gpv* (*f* (*result generat*))
else return-spmf (*IO* (*output generat*) (λ*input. continuation generat input* $\gg=$ *f*)))
by(*auto*)
show ($\exists$ *gpv. c input* = *gpv* $\gg=$ *f* $\wedge$ *colossless-gpv* $\mathcal{I}$ *gpv* $\wedge$ ($\forall$ *x. x* $\in$ *results-gpv* $\mathcal{I}$ *gpv* $\longrightarrow$ *colossless-gpv* $\mathcal{I}$ (*f x*))) $\vee$
colossless-gpv $\mathcal{I}$ (*c input*)

```
proof(cases generat)
  case (Pure x)
  hence x ∈ results-gpv ℐ gpv using generat by(auto intro: results-gpv.Pure)
  from *(2)[OF this] have colossless-gpv ℐ (c input)
    using IO Pure input by(auto intro: colossless-gpv-continuationD)
  thus ?thesis ..
next
  case **: (IO out' c')
  with input generat IO have colossless-gpv ℐ (f x) if x ∈ results-gpv ℐ (c' input) for x
    using that by(auto intro: * results-gpv.IO)
  then show ?thesis using IO input ** *(1) generat by(auto dest: colossless-gpv-continuationD)
qed
qed
ultimately show ?case ..
qed
qed

lemmas lossless-bind-gpvI = gen-lossless-bind-gpvI[where co=False]
  and colossless-bind-gpvI = gen-lossless-bind-gpvI[where co=True]

lemma gen-lossless-bind-gpvD1:
  assumes gen-lossless-gpv co ℐ (gpv ≫= f)
  shows gen-lossless-gpv co ℐ gpv
proof(cases co)
  case False
  hence eq: co = False by simp
  show ?thesis using assms unfolding eq
  proof(induction gpv'≡gpv ≫= f arbitrary: gpv)
    case (lossless-gpv p)
    obtain p' where gpv: gpv = GPV p' by(cases gpv)
    from lossless-gpv.hyps gpv have lossless-spmf p' by(simp add: GPV-bind)
    then show ?case unfolding gpv
    proof(rule gen-lossless-gpv-intros)
      fix out c input
      assume IO out c ∈ set-spmf p' input ∈ responses-ℐ ℐ out
      hence IO out (λinput. c input ≫= f) ∈ set-spmf p using lossless-gpv.hyps gpv
        by(auto simp add: GPV-bind intro: rev-bexI)
      thus lossless-gpv ℐ (c input) using ‹input ∈ -› by(rule lossless-gpv.hyps) simp
    qed
  qed
next
  case True
  hence eq: co = True by simp
  show ?thesis using assms unfolding eq
    by(coinduction arbitrary: gpv)(auto 4 3 intro: rev-bexI elim!: colossless-gpv-continuationD
```

dest: *colossless-gpv-lossless-spmfD*)
qed

lemmas *lossless-bind-gpvD1* = *gen-lossless-bind-gpvD1*[**where** *co*=*False*]
  and *colossless-bind-gpvD1* = *gen-lossless-bind-gpvD1*[**where** *co*=*True*]

lemma *gen-lossless-bind-gpvD2*:
  assumes *gen-lossless-gpv co* $\mathcal{I}$ (*gpv* ≫= *f*)
  and *x* ∈ *results-gpv* $\mathcal{I}$ *gpv*
  shows *gen-lossless-gpv co* $\mathcal{I}$ (*f x*)
using *assms*(*2*,*1*)
proof(*induction*)
  case (*Pure gpv*)
  thus *?case*
    by −(*rule gen-lossless-gpvI*, *auto 4 4 dest*: *gen-lossless-gpvD intro*: *rev-bexI*)
qed(*auto 4 4 dest*: *gen-lossless-gpvD intro*: *rev-bexI*)

lemmas *lossless-bind-gpvD2* = *gen-lossless-bind-gpvD2*[**where** *co*=*False*]
  and *colossless-bind-gpvD2* = *gen-lossless-bind-gpvD2*[**where** *co*=*True*]

lemma *gen-lossless-bind-gpv* [*simp*]:
  gen-lossless-gpv co $\mathcal{I}$ (*gpv* ≫= *f*) ⟷ *gen-lossless-gpv co* $\mathcal{I}$ *gpv* ∧ (∀ *x*∈*results-gpv* $\mathcal{I}$ *gpv*. *gen-lossless-gpv co* $\mathcal{I}$ (*f x*))
by(*blast intro*: *gen-lossless-bind-gpvI dest*: *gen-lossless-bind-gpvD1 gen-lossless-bind-gpvD2*)

lemmas *lossless-bind-gpv* = *gen-lossless-bind-gpv*[**where** *co*=*False*]
  and *colossless-bind-gpv* = *gen-lossless-bind-gpv*[**where** *co*=*True*]

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

lemma *rel-gpv′′-lossless-gpvD1*:
  assumes *rel*: *rel-gpv′′ A C R gpv gpv′*
  and *gpv*: *lossless-gpv* $\mathcal{I}$ *gpv*
  and [*transfer-rule*]: *rel-*$\mathcal{I}$ *C R* $\mathcal{I}$ $\mathcal{I}'$
  shows *lossless-gpv* $\mathcal{I}'$ *gpv′*
using *gpv rel*
proof(*induction arbitrary*: *gpv′*)
  case (*lossless-gpv p*)
  from *lossless-gpv.prems* **obtain** *q* **where** *q*: *gpv′* = *GPV q*
    and [*transfer-rule*]: *rel-spmf* (*rel-generat A C* (*R* ===> *rel-gpv′′ A C R*)) *p q*
    by(*cases gpv′*) *auto*
  show *?case*
  proof(*rule lossless-gpvI*)
    have *lossless-spmf p* = *lossless-spmf q* **by** *transfer-prover*
    with *lossless-gpv.hyps*(*1*) *q* **show** *lossless-spmf* (*the-gpv gpv′*) **by** *simp*

    fix *out′ c′ input′*
    assume *IO′*: *IO out′ c′* ∈ *set-spmf* (*the-gpv gpv′*)
      and *input′*: *input′* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out′*

have *rel-set* (*rel-generat A C* (*R* ===> *rel-gpv'' A C R*)) (*set-spmf p*) (*set-spmf q*)
by *transfer-prover*
with *IO' q* **obtain** *out c* **where** *IO*: *IO out c* ∈ *set-spmf p*
and [*transfer-rule*]: *C out out'* (*R* ===> *rel-gpv'' A C R*) *c c'*
by(*auto dest*!: *rel-setD2 elim*: *generat.rel-cases*)
have *rel-set R* (*responses-ℐ ℐ out*) (*responses-ℐ ℐ' out'*) **by** *transfer-prover*
moreover
from *this*[*THEN rel-setD2*, *OF input'*] **obtain** *input*
where [*transfer-rule*]: *R input input'* **and** *input*: *input* ∈ *responses-ℐ ℐ out* **by** *blast*
have *rel-gpv'' A C R* (*c input*) (*c' input'*) **by** *transfer-prover*
ultimately show *lossless-gpv ℐ'* (*c' input'*) **using** *input IO* **by**(*auto intro*: *lossless-gpv.IH*)
qed
qed

lemma *rel-gpv''-lossless-gpvD2*:
⟦ *rel-gpv'' A C R gpv gpv'*; *lossless-gpv ℐ' gpv'*; *rel-ℐ C R ℐ ℐ'* ⟧
⟹ *lossless-gpv ℐ gpv*
using *rel-gpv''-lossless-gpvD1*[*of* A^{-1-1} C^{-1-1} R^{-1-1} *gpv' gpv ℐ' ℐ*]
by(*simp add*: *rel-gpv''-conversep prod.rel-conversep rel-fun-eq-conversep*)

lemma *rel-gpv-lossless-gpvD1*:
⟦ *rel-gpv A C gpv gpv'*; *lossless-gpv ℐ gpv*; *rel-ℐ C* (=) *ℐ ℐ'* ⟧ ⟹ *lossless-gpv ℐ' gpv'*
using *rel-gpv''-lossless-gpvD1*[*of A C* (=) *gpv gpv' ℐ ℐ'*] **by**(*simp add*: *rel-gpv-conv-rel-gpv''*)

lemma *rel-gpv-lossless-gpvD2*:
⟦ *rel-gpv A C gpv gpv'*; *lossless-gpv ℐ' gpv'*; *rel-ℐ C* (=) *ℐ ℐ'* ⟧
⟹ *lossless-gpv ℐ gpv*
using *rel-gpv-lossless-gpvD1*[*of* A^{-1-1} C^{-1-1} *gpv' gpv ℐ' ℐ*]
by(*simp add*: *gpv.rel-conversep prod.rel-conversep rel-fun-eq-conversep*)

lemma *rel-gpv''-colossless-gpvD1*:
assumes *rel*: *rel-gpv'' A C R gpv gpv'*
and *gpv*: *colossless-gpv ℐ gpv*
and [*transfer-rule*]: *rel-ℐ C R ℐ ℐ'*
shows *colossless-gpv ℐ' gpv'*
using *gpv rel*
proof(*coinduction arbitrary*: *gpv gpv'*)
case (*colossless-gpv gpv gpv'*)
note [*transfer-rule*] = ‹*rel-gpv'' A C R gpv gpv'*› *the-gpv-parametric'*
and *co* = ‹*colossless-gpv ℐ gpv*›
have *lossless-spmf* (*the-gpv gpv*) = *lossless-spmf* (*the-gpv gpv'*) **by** *transfer-prover*
with *co* **have** *?lossless-spmf* **by**(*auto dest*: *colossless-gpv-lossless-spmfD*)
moreover have *?continuation*
proof(*intro strip disjI1*)
fix *out' c' input'*

assume *IO′*: *IO out′ c′* ∈ *set-spmf* (*the-gpv gpv′*)
and *input′*: *input′* ∈ *responses-𝓘 𝓘′ out′*
have *rel-set* (*rel-generat A C* (*R* ===> *rel-gpv′′ A C R*)) (*set-spmf* (*the-gpv gpv*)) (*set-spmf* (*the-gpv gpv′*))
by *transfer-prover*
with *IO′* **obtain** *out c* **where** *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and [*transfer-rule*]: *C out out′* (*R* ===> *rel-gpv′′ A C R*) *c c′*
by(*auto dest*!: *rel-setD2 elim*: *generat.rel-cases*)
have *rel-set R* (*responses-𝓘 𝓘 out*) (*responses-𝓘 𝓘′ out′*) **by** *transfer-prover*
moreover
from *this*[*THEN rel-setD2*, *OF input′*] **obtain** *input*
where [*transfer-rule*]: *R input input′* **and** *input*: *input* ∈ *responses-𝓘 𝓘 out* **by** *blast*
have *rel-gpv′′ A C R* (*c input*) (*c′ input′*) **by** *transfer-prover*
ultimately show ∃ *gpv gpv′*. *c′ input′* = *gpv′* ∧ *colossless-gpv 𝓘 gpv* ∧ *rel-gpv′′ A C R gpv gpv′*
using *input IO co* **by**(*auto dest*: *colossless-gpv-continuationD*)
qed
ultimately show *?case* ..
qed

lemma *rel-gpv′′-colossless-gpvD2*:
⟦ *rel-gpv′′ A C R gpv gpv′*; *colossless-gpv 𝓘′ gpv′*; *rel-𝓘 C R 𝓘 𝓘′* ⟧
⟹ *colossless-gpv 𝓘 gpv*
using *rel-gpv′′-colossless-gpvD1*[*of* A^{-1-1} C^{-1-1} R^{-1-1} *gpv′ gpv 𝓘′ 𝓘*]
by(*simp add*: *rel-gpv′′-conversep prod.rel-conversep rel-fun-eq-conversep*)

lemma *rel-gpv-colossless-gpvD1*:
⟦ *rel-gpv A C gpv gpv′*; *colossless-gpv 𝓘 gpv*; *rel-𝓘 C* (=) *𝓘 𝓘′* ⟧ ⟹ *colossless-gpv 𝓘′ gpv′*
using *rel-gpv′′-colossless-gpvD1*[*of A C* (=) *gpv gpv′ 𝓘 𝓘′*] **by**(*simp add*: *rel-gpv-conv-rel-gpv′′*)

lemma *rel-gpv-colossless-gpvD2*:
⟦ *rel-gpv A C gpv gpv′*; *colossless-gpv 𝓘′ gpv′*; *rel-𝓘 C* (=) *𝓘 𝓘′* ⟧
⟹ *colossless-gpv 𝓘 gpv*
using *rel-gpv-colossless-gpvD1*[*of* A^{-1-1} C^{-1-1} *gpv′ gpv 𝓘′ 𝓘*]
by(*simp add*: *gpv.rel-conversep prod.rel-conversep rel-fun-eq-conversep*)

lemma *gen-lossless-gpv-parametric′*:
((=) ===> *rel-𝓘 C R* ===> *rel-gpv′′ A C R* ===> (=))
gen-lossless-gpv gen-lossless-gpv
proof(*rule rel-funI*; *hypsubst*)
show (*rel-𝓘 C R* ===> *rel-gpv′′ A C R* ===> (=)) (*gen-lossless-gpv b*) (*gen-lossless-gpv b*) **for** *b*
by(*cases b*)(*auto intro*!: *rel-funI dest*: *rel-gpv′′-colossless-gpvD1 rel-gpv′′-colossless-gpvD2 rel-gpv′′-lossless-gpvD1 rel-gpv′′-lossless-gpvD2*)
qed

lemma *gen-lossless-gpv-parametric* [*transfer-rule*]:

((=) ===> *rel-ℐ C* (=) ===> *rel-gpv A C* ===> (=))
gen-lossless-gpv gen-lossless-gpv
proof(*rule rel-funI*; *hypsubst*)
show (*rel-ℐ C* (=) ===> *rel-gpv A C* ===> (=)) (*gen-lossless-gpv b*) (*gen-lossless-gpv b*) **for** *b*
by(*cases b*)(*auto intro*!: *rel-funI dest*: *rel-gpv-colossless-gpvD1 rel-gpv-colossless-gpvD2 rel-gpv-lossless-gpvD1 rel-gpv-lossless-gpvD2*)
qed

end

lemma *gen-lossless-gpv-map-full* [*simp*]:
gen-lossless-gpv b ℐ-full (*map-gpv f g gpv*) = *gen-lossless-gpv b ℐ-full gpv*
(**is** *?lhs* = *?rhs*)
proof(*cases b* = *True*)
case *True*
show *?lhs* = *?rhs*
proof
show *?rhs* **if** *?lhs* **using** *that* **unfolding** *True*
by(*coinduction arbitrary*: *gpv*)(*auto 4 3 dest*: *colossless-gpvD simp add*: *gpv.map-sel intro*!: *rev-image-eqI*)
show *?lhs* **if** *?rhs* **using** *that* **unfolding** *True*
by(*coinduction arbitrary*: *gpv*)(*auto 4 4 dest*: *colossless-gpvD simp add*: *gpv.map-sel intro*!: *rev-image-eqI*)
qed
next
case *False*
hence *False*: *b* = *False* **by** *simp*
show *?lhs* = *?rhs*
proof
show *?rhs* **if** *?lhs* **using** *that* **unfolding** *False*
apply(*induction gpv′≡map-gpv f g gpv arbitrary*: *gpv*)
subgoal for *p gpv* **by**(*cases gpv*)(*rule lossless-gpvI*; *fastforce intro*: *rev-image-eqI*)
done
show *?lhs* **if** *?rhs* **using** *that* **unfolding** *False*
by *induction*(*auto 4 4 intro*: *lossless-gpvI*)
qed
qed

lemma *gen-lossless-gpv-map-id* [*simp*]:
gen-lossless-gpv b ℐ (*map-gpv f id gpv*) = *gen-lossless-gpv b ℐ gpv*
using *gen-lossless-gpv-parametric*[*of BNF-Def.Grp UNIV id BNF-Def.Grp UNIV f*] **unfolding** *gpv.rel-Grp*
by(*simp add*: *rel-fun-def eq-alt*[*symmetric*] *rel-ℐ-eq*)(*auto simp add*: *Grp-def*)

lemma *results-gpv-try-gpv* [*simp*]:
results-gpv ℐ (*TRY gpv ELSE gpv′*) =
results-gpv ℐ gpv ∪ (*if colossless-gpv ℐ gpv then* {} *else results-gpv ℐ gpv′*)
(**is** *?lhs* = *?rhs*)

```
proof(intro set-eqI iffI)
  show x ∈ ?rhs if x ∈ ?lhs for x using that
  proof(induction gpv''≡try-gpv gpv gpv' arbitrary: gpv)
    case Pure thus ?case
     by(auto split: if-split-asm intro: results-gpv.Pure dest: colossless-gpv-lossless-spmfD)
  next
    case (IO out c input)
    then show ?case
      apply(auto dest: colossless-gpv-lossless-spmfD split: if-split-asm)
        apply(force intro: results-gpv.IO dest: colossless-gpv-continuationD split: if-split-asm)+
      done
  qed
next
  fix x
  assume x ∈ ?rhs
  then consider (left) x ∈ results-gpv ℐ gpv | (right) ¬ colossless-gpv ℐ gpv x ∈ results-gpv ℐ gpv'
    by(auto split: if-split-asm)
  thus x ∈ ?lhs
  proof cases
    case left
    thus ?thesis
     by(induction)(auto 4 4 intro: results-gpv.intros rev-image-eqI split del: if-split)
  next
    case right
    from right(1) show ?thesis
    proof(rule contrapos-np)
      assume x ∉ ?lhs
      with right(2) show colossless-gpv ℐ gpv
      proof(coinduction arbitrary: gpv)
        case (colossless-gpv gpv)
        then have ?lossless-spmf
          apply(rewrite in asm try-gpv.code)
          apply(rule ccontr)
          apply(erule results-gpv.cases)
         apply(fastforce simp add: image-Un image-image generat.map-comp o-def)+
          done
        moreover have ?continuation using colossless-gpv
             by(auto 4 4 split del: if-split simp add: image-Un image-image generat.map-comp o-def intro: rev-image-eqI results-gpv.IO)
        ultimately show ?case ..
      qed
    qed
  qed
qed

lemma results'-gpv-try-gpv [simp]:
  results'-gpv (TRY gpv ELSE gpv') =
```

results′-gpv gpv ∪ (*if colossless-gpv ℐ-full gpv then* {} *else results′-gpv gpv′*)
by(*simp add*: *results-gpv-ℐ-full*[*symmetric*])

lemma *outs′-gpv-try-gpv* [*simp*]:
outs′-gpv (*TRY gpv ELSE gpv′*) =
outs′-gpv gpv ∪ (*if colossless-gpv ℐ-full gpv then* {} *else outs′-gpv gpv′*)
(**is** *?lhs* = *?rhs*)
proof(*intro set-eqI iffI*)
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
proof(*induction gpv″≡try-gpv gpv gpv′ arbitrary*: *gpv*)
case *Out* **thus** *?case*
by(*auto 4 3 simp add*: *generat.map-comp o-def elim*!: *generat.set-cases*(*2*)
intro: *outs′-gpv-Out split*: *if-split-asm dest*: *colossless-gpv-lossless-spmfD*)
next
case (*Cont generat c input*)
then show *?case*
apply(*auto dest*: *colossless-gpv-lossless-spmfD split*: *if-split-asm elim*!: *generat.set-cases*(*3*))
apply(*auto 4 3 dest*: *colossless-gpv-continuationD split*: *if-split-asm intro*:
outs′-gpv-Cont elim!: *meta-allE meta-impE*[*OF - refl*])+
done
qed
next
fix *x*
assume *x* ∈ *?rhs*
then consider (*left*) *x* ∈ *outs′-gpv gpv* | (*right*) ¬ *colossless-gpv ℐ-full gpv x* ∈
outs′-gpv gpv′
by(*auto split*: *if-split-asm*)
thus *x* ∈ *?lhs*
proof *cases*
case *left*
thus *?thesis*
by(*induction*)(*auto elim*!: *generat.set-cases*(*2*,*3*) *intro*: *outs′-gpvI intro*!: *rev-image-eqI*
split del: *if-split simp add*: *image-Un image-image generat.map-comp o-def*)
next
case *right*
from *right*(*1*) **show** *?thesis*
proof(*rule contrapos-np*)
assume *x* ∉ *?lhs*
with *right*(*2*) **show** *colossless-gpv ℐ-full gpv*
proof(*coinduction arbitrary*: *gpv*)
case (*colossless-gpv gpv*)
then have *?lossless-spmf*
apply(*rewrite* **in** *asm try-gpv.code*)
apply(*erule contrapos-np*)
apply(*erule gpv.set-cases*)
apply(*auto 4 3 simp add*: *image-Un image-image generat.map-comp o-def*
generat.set-map in-set-spmf[*symmetric*] *bind-UNION generat.map-id*[*unfolded id-def*]
elim!: *generat.set-cases*)

done
moreover have *?continuation* **using** *colossless-gpv*
by(*auto simp add*: *image-Un image-image generat.map-comp o-def split del*: *if-split intro*!: *rev-image-eqI intro*: *outs′-gpv-Cont*)
ultimately show *?case* **..**
qed
qed
qed
qed

lemma *pred-gpv-try* [*simp*]:
pred-gpv P Q (*try-gpv gpv gpv′*) = (*pred-gpv P Q gpv* ∧ (¬ *colossless-gpv ℐ-full gpv* ⟶ *pred-gpv P Q gpv′*))
by(*auto simp add*: *pred-gpv-def*)

lemma *lossless-WT-gpv-induct* [*consumes 2*, *case-names lossless-gpv*]:
assumes *lossless*: *lossless-gpv ℐ gpv*
and *WT*: *ℐ* ⊢*g gpv* √
and *step*: ⋀*p*. ⟦
lossless-spmf p;
⋀*out c*. *IO out c* ∈ *set-spmf p* ⟹ *out* ∈ *outs-ℐ ℐ*;
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *out* ∈ *outs-ℐ ℐ* ⟹ *input* ∈ *responses-ℐ ℐ out*⟧ ⟹ *lossless-gpv ℐ* (*c input*);
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *out* ∈ *outs-ℐ ℐ* ⟹ *input* ∈ *responses-ℐ ℐ out*⟧ ⟹ *ℐ* ⊢*g c input* √;
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *out* ∈ *outs-ℐ ℐ* ⟹ *input* ∈ *responses-ℐ ℐ out*⟧ ⟹ *P* (*c input*)⟧
⟹ *P* (*GPV p*)
shows *P gpv*
using *lossless WT*
apply(*induction*)
apply(*erule step*)
apply(*auto elim*: *WT-gpvD simp add*: *WT-gpv-simps*)
done

lemma *lossless-gpv-induct-strong* [*consumes 1*, *case-names lossless-gpv*]:
assumes *gpv*: *lossless-gpv ℐ gpv*
and *step*:
⋀*p*. ⟦ *lossless-spmf p*;
⋀*gpv*. *gpv* ∈ *sub-gpvs ℐ* (*GPV p*) ⟹ *lossless-gpv ℐ gpv*;
⋀*gpv*. *gpv* ∈ *sub-gpvs ℐ* (*GPV p*) ⟹ *P gpv* ⟧
⟹ *P* (*GPV p*)
shows *P gpv*
proof −
define *gpv′* **where** *gpv′* = *gpv*
then have *gpv′* ∈ *insert gpv* (*sub-gpvs ℐ gpv*) **by** *simp*
with *gpv* **have** *lossless-gpv ℐ gpv′* ∧ *P gpv′*
proof(*induction arbitrary*: *gpv′*)
case (*lossless-gpv p*)

from ‹*gpv′* ∈ *insert* (*GPV p*) -› **show** *?case*
proof(*rule insertE*)
assume *gpv′* = *GPV p*
moreover have *lossless-gpv* $\mathcal{I}$ (*GPV p*)
by(*auto 4 3 intro*: *lossless-gpvI lossless-gpv.hyps*)
moreover have *P* (*GPV p*) **using** *lossless-gpv.hyps*(*1*)
by(*rule step*)(*fastforce elim*: *sub-gpvs.cases lossless-gpv.IH*[*THEN conjunct1*] *lossless-gpv.IH*[*THEN conjunct2*])+
ultimately show *?case* **by** *simp*
qed(*fastforce elim*: *sub-gpvs.cases lossless-gpv.IH*[*THEN conjunct1*] *lossless-gpv.IH*[*THEN conjunct2*])
qed
thus *?thesis* **by**(*simp add*: *gpv′-def*)
qed

lemma *lossless-sub-gpvsI*:
assumes *spmf*: *lossless-spmf* (*the-gpv gpv*)
and *sub*: ⋀*gpv′*. *gpv′* ∈ *sub-gpvs* $\mathcal{I}$ *gpv* ⟹ *lossless-gpv* $\mathcal{I}$ *gpv′*
shows *lossless-gpv* $\mathcal{I}$ *gpv*
using *spmf* **by**(*rule lossless-gpvI*)(*rule sub*[*OF sub-gpvs.base*])

lemma *lossless-sub-gpvsD*:
assumes *lossless-gpv* $\mathcal{I}$ *gpv gpv′* ∈ *sub-gpvs* $\mathcal{I}$ *gpv*
shows *lossless-gpv* $\mathcal{I}$ *gpv′*
using *assms*(*2*,*1*) **by**(*induction*)(*auto dest*: *lossless-gpvD*)

lemma *lossless-WT-gpv-induct-strong* [*consumes 2*, *case-names lossless-gpv*]:
assumes *lossless*: *lossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I}$ ⊢*g gpv* √
and *step*: ⋀*p*. ⟦ *lossless-spmf p*;
⋀*out c*. *IO out c* ∈ *set-spmf p* ⟹ *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$;
⋀*gpv*. *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ *lossless-gpv* $\mathcal{I}$ *gpv*;
⋀*gpv*. *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ $\mathcal{I}$ ⊢*g gpv* √;
⋀*gpv*. *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ *P gpv* ⟧
⟹ *P* (*GPV p*)
shows *P gpv*
using *lossless WT*
apply(*induction rule*: *lossless-gpv-induct-strong*)
apply(*erule step*)
apply(*auto elim*: *WT-gpvD dest*: *WT-sub-gpvsD*)
done

lemma *try-gpv-gen-lossless*: — TODO: generalise to arbitrary typings ?
gen-lossless-gpv b $\mathcal{I}$*-full gpv* ⟹ (*TRY gpv ELSE gpv′*) = *gpv*
proof(*coinduction arbitrary*: *gpv*)
case (*Eq-gpv gpv*)
from *Eq-gpv*[*THEN gen-lossless-gpv-lossless-spmfD*]
have *eq*: *the-gpv gpv* = (*TRY the-gpv gpv ELSE the-gpv gpv′*) **by**(*simp*)
show *?case*

by(*subst eq*)(*auto simp add*: *spmf-rel-map generat.rel-map*[*abs-def*] *intro*!: *rel-spmf-try-spmf rel-spmf-reflI rel-generat-reflI elim*!: *generat.set-cases gen-lossless-gpv-continuationD*[*OF Eq-gpv*] *simp add*: *Eq-gpv*[*THEN gen-lossless-gpv-lossless-spmfD*])
qed

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

lemma *try-gpv-bind-gen-lossless*: — TODO: generalise to arbitrary typings?
gen-lossless-gpv b $\mathcal{I}$-full gpv $\Longrightarrow$ *TRY bind-gpv gpv f ELSE gpv′* = *bind-gpv gpv* (λ*x*. *TRY f x ELSE gpv′*)
proof(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
case (*Eq-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-gpv*]
by(*auto 4 3 simp del*: *bind-gpv-sel′ simp add*: *bind-gpv.sel try-spmf-bind-spmf-lossless split*: *generat.split intro*!: *rel-spmf-bind-reflI rel-spmf-try-spmf*)
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-cong*:
⟦ *gpv* = *gpv″*; ¬ *colossless-gpv $\mathcal{I}$-full gpv″* $\Longrightarrow$ *gpv′* = *gpv‴* ⟧
$\Longrightarrow$ *try-gpv gpv gpv′* = *try-gpv gpv″ gpv‴*
by(*cases colossless-gpv $\mathcal{I}$-full gpv″*) *simp-all*

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

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

end

lemma *WT-gpv-mk-lossless-gpv*:
assumes $\mathcal{I}$ ⊢*g gpv* √
and *outs*: *outs-$\mathcal{I}$ $\mathcal{I}$′* = *outs-$\mathcal{I}$ $\mathcal{I}$*
shows $\mathcal{I}$′ ⊢*g mk-lossless-gpv* (*responses-$\mathcal{I}$ $\mathcal{I}$*) *x gpv* √

using *assms*(*1*)
by(*coinduction arbitrary*: *gpv*)(*auto 4 3 split*: *generat.split-asm simp add*: *outs dest*: *WT-gpvD*)

4.15 Sequencing with failure handling included

definition *catch-gpv* :: ('*a*, '*out*, '*in*) *gpv* ⇒ ('*a option*, '*out*, '*in*) *gpv*
where *catch-gpv gpv* = *TRY map-gpv Some id gpv ELSE Done None*

lemma *catch-gpv-Done* [*simp*]: *catch-gpv* (*Done x*) = *Done* (*Some x*)
by(*simp add*: *catch-gpv-def*)

lemma *catch-gpv-Fail* [*simp*]: *catch-gpv Fail* = *Done None*
by(*simp add*: *catch-gpv-def*)

lemma *catch-gpv-Pause* [*simp*]: *catch-gpv* (*Pause out rpv*) = *Pause out* (λ*input*. *catch-gpv* (*rpv input*))
by(*simp add*: *catch-gpv-def*)

lemma *catch-gpv-lift-spmf* [*simp*]: *catch-gpv* (*lift-spmf p*) = *lift-spmf* (*spmf-of-pmf p*)
by(*rule gpv.expand*)(*auto simp add*: *catch-gpv-def spmf-of-pmf-def map-lift-spmf try-spmf-def o-def map-pmf-def bind-assoc-pmf bind-return-pmf intro*!: *bind-pmf-cong*[*OF refl*] *split*: *option.split*)

lemma *catch-gpv-assert* [*simp*]: *catch-gpv* (*assert-gpv b*) = *Done* (*assert-option b*)
by(*cases b*) *simp-all*

lemma *catch-gpv-sel* [*simp*]:
  the-gpv (*catch-gpv gpv*) =
    TRY map-spmf (*map-generat Some id* (λ*rpv input*. *catch-gpv* (*rpv input*))) (*the-gpv gpv*)
   ELSE return-spmf (*Pure None*)
by(*simp add*: *catch-gpv-def gpv.map-sel spmf.map-comp o-def generat.map-comp map-try-spmf id-def*)

lemma *catch-gpv-bind-gpv*: *catch-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*catch-gpv gpv*) (λ*x*. *case x of None* ⇒ *Done None* | *Some x'* ⇒ *catch-gpv* (*f x'*))
  using [[*show-variants*]]
  apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
  apply(*clarsimp simp add*: *map-bind-pmf bind-gpv.sel spmf.map-comp o-def*[*abs-def*] *map-bind-spmf generat.map-comp simp del*: *bind-gpv-sel'*)
  apply(*subst bind-spmf-def*)
  apply(*subst try-spmf-bind-pmf*)
  apply(*subst* (*2*) *try-spmf-def*)
  apply(*subst bind-spmf-pmf-assoc*)
  apply(*simp add*: *bind-map-pmf*)
  apply(*rule rel-pmf-bind-reflI*)
  apply(*auto split*!: *option.split generat.split simp add*: *spmf-rel-map spmf.map-comp*

o-def generat.map-comp id-def[*symmetric*] *generat.map-id rel-spmf-reflI generat.rel-refl refl rel-fun-def*)
  done

context includes *lifting-syntax* **begin**
lemma *catch-gpv-parametric* [*transfer-rule*]:
  (*rel-gpv A C* ===> *rel-gpv* (*rel-option A*) *C*) *catch-gpv catch-gpv*
unfolding *catch-gpv-def* **by** *transfer-prover*

lemma *catch-gpv-parametric'*:
  notes [*transfer-rule*] = *try-gpv-parametric' map-gpv-parametric' Done-parametric'*
  shows (*rel-gpv'' A C R* ===> *rel-gpv''* (*rel-option A*) *C R*) *catch-gpv catch-gpv*
unfolding *catch-gpv-def* **by** *transfer-prover*
end

lemma *catch-gpv-map'*: *catch-gpv* (*map-gpv' f g h gpv*) = *map-gpv'* (*map-option f*) *g h* (*catch-gpv gpv*)
by(*simp add*: *catch-gpv-def map'-try-gpv map-gpv-conv-map-gpv' map-gpv'-comp o-def*)

lemma *catch-gpv-map*: *catch-gpv* (*map-gpv f g gpv*) = *map-gpv* (*map-option f*) *g* (*catch-gpv gpv*)
  by(*simp add*: *map-gpv-conv-map-gpv' catch-gpv-map'*)

lemma *colossless-gpv-catch-gpv* [*simp*]: *colossless-gpv* $\mathcal{I}$*-full* (*catch-gpv gpv*)
by(*coinduction arbitrary*: *gpv*) *auto*

lemma *colosless-gpv-catch-gpv-conv-map*:
  colossless-gpv $\mathcal{I}$*-full gpv* $\Longrightarrow$ *catch-gpv gpv* = *map-gpv Some id gpv*
  apply(*coinduction arbitrary*: *gpv*)
  apply(*frule colossless-gpv-lossless-spmfD*)
  apply(*auto simp add*: *spmf-rel-map gpv.map-sel generat.rel-map intro*!: *rel-spmf-reflI generat.rel-refl-strong rel-funI elim*!: *colossless-gpv-continuationD generat.set-cases*)
  done

lemma *catch-gpv-catch-gpv* [*simp*]: *catch-gpv* (*catch-gpv gpv*) = *map-gpv Some id* (*catch-gpv gpv*)
  by(*simp add*: *colosless-gpv-catch-gpv-conv-map*)

lemma *case-map-resumption*:
  case-resumption done pause (*map-resumption f g r*) =
   case-resumption (*done* ∘ *map-option f*) (λ*out c. pause* (*g out*) (*map-resumption f g* ∘ *c*)) *r*
by(*cases r*) *simp-all*

lemma *catch-gpv-lift-resumption* [*simp*]: *catch-gpv* (*lift-resumption r*) = *lift-resumption* (*map-resumption Some id r*)
  apply(*coinduction arbitrary*: *r*)
   apply(*auto simp add*: *lift-resumption.sel case-map-resumption split*: *resump-*

tion.split option.split)
oops

lemma *results-gpv-catch-gpv*:
results-gpv $\mathcal{I}$ (*catch-gpv gpv*) = *Some* ‘ *results-gpv* $\mathcal{I}$ *gpv* ∪ (*if colossless-gpv* $\mathcal{I}$ *gpv then* {} *else* {*None*})
by(*simp add*: *catch-gpv-def*)

lemma *Some-in-results-gpv-catch-gpv* [*simp*]:
Some x ∈ *results-gpv* $\mathcal{I}$ (*catch-gpv gpv*) ⟷ *x* ∈ *results-gpv* $\mathcal{I}$ *gpv*
by(*auto simp add*: *results-gpv-catch-gpv*)

lemma *None-in-results-gpv-catch-gpv* [*simp*]:
None ∈ *results-gpv* $\mathcal{I}$ (*catch-gpv gpv*) ⟷ ¬ *colossless-gpv* $\mathcal{I}$ *gpv*
by(*auto simp add*: *results-gpv-catch-gpv*)

lemma *results′-gpv-catch-gpv*:
results′-gpv (*catch-gpv gpv*) = *Some* ‘ *results′-gpv gpv* ∪ (*if colossless-gpv* $\mathcal{I}$*-full gpv then* {} *else* {*None*})
by(*simp add*: *results-gpv-*$\mathcal{I}$*-full*[*symmetric*] *results-gpv-catch-gpv*)

lemma *Some-in-results′-gpv-catch-gpv* [*simp*]:
Some x ∈ *results′-gpv* (*catch-gpv gpv*) ⟷ *x* ∈ *results′-gpv gpv*
by(*simp add*: *results-gpv-*$\mathcal{I}$*-full*[*symmetric*])

lemma *None-in-results′-gpv-catch-gpv* [*simp*]:
None ∈ *results′-gpv* (*catch-gpv gpv*) ⟷ ¬ *colossless-gpv* $\mathcal{I}$*-full gpv*
by(*simp add*: *results-gpv-*$\mathcal{I}$*-full*[*symmetric*])

lemma *results′-gpv-catch-gpvE*:
assumes *x* ∈ *results′-gpv* (*catch-gpv gpv*)
obtains (*Some*) *x′*
where *x* = *Some x′ x′* ∈ *results′-gpv gpv*
| (*colossless*) *x* = *None* ¬ *colossless-gpv* $\mathcal{I}$*-full gpv*
using *assms* **by**(*auto simp add*: *results′-gpv-catch-gpv split*: *if-split-asm*)

lemma *outs′-gpv-catch-gpv* [*simp*]: *outs′-gpv* (*catch-gpv gpv*) = *outs′-gpv gpv*
by(*simp add*: *catch-gpv-def*)

lemma *pred-gpv-catch-gpv* [*simp*]: *pred-gpv* (*pred-option P*) *Q* (*catch-gpv gpv*) = *pred-gpv P Q gpv*
by(*simp add*: *pred-gpv-def results′-gpv-catch-gpv*)

abbreviation *bind-gpv′* :: (′*a*, ′*call*, ′*ret*) *gpv* ⇒ (′*a option* ⇒ (′*b*, ′*call*, ′*ret*) *gpv*) ⇒ (′*b*, ′*call*, ′*ret*) *gpv*
where *bind-gpv′ gpv* ≡ *bind-gpv* (*catch-gpv gpv*)

lemma *bind-gpv'-assoc* [*simp*]: *bind-gpv'* (*bind-gpv' gpv f*) *g* = *bind-gpv' gpv* (λ*x*. *bind-gpv'* (*f x*) *g*)
by(*simp add*: *catch-gpv-bind-gpv bind-map-gpv o-def bind-gpv-assoc*)

lemma *bind-gpv'-bind-gpv*: *bind-gpv'* (*bind-gpv gpv f*) *g* = *bind-gpv' gpv* (*case-option* (*g None*) (λ*y*. *bind-gpv'* (*f y*) *g*))
by(*clarsimp simp add*: *catch-gpv-bind-gpv bind-gpv-assoc intro*!: *bind-gpv-cong*[*OF refl*] *split*: *option.split*)

lemma *bind-gpv'-cong*:
⟦ *gpv* = *gpv'*; ⋀*x*. *x* ∈ *Some* ' *results'-gpv gpv'* ∨ (¬ *colossless-gpv* 𝓘*-full gpv* ∧ *x* = *None*) ⟹ *f x* = *f' x* ⟧
⟹ *bind-gpv' gpv f* = *bind-gpv' gpv' f'*
by(*auto elim*: *results'-gpv-catch-gpvE split*: *if-split-asm intro*!: *bind-gpv-cong*[*OF refl*])

lemma *bind-gpv'-cong2*:
⟦ *gpv* = *gpv'*; ⋀*x*. *x* ∈ *results'-gpv gpv'* ⟹ *f* (*Some x*) = *f'* (*Some x*); ¬ *colossless-gpv* 𝓘*-full gpv* ⟹ *f None* = *f' None* ⟧
⟹ *bind-gpv' gpv f* = *bind-gpv' gpv' f'*
by(*rule bind-gpv'-cong*) *auto*

4.16 Inlining

lemma *gpv-coinduct-bind* [*consumes 1*, *case-names Eq-gpv*]:
  fixes *gpv gpv'* :: (*'a*, *'call*, *'ret*) *gpv*
  assumes ∗: *R gpv gpv'*
  and *step*: ⋀*gpv gpv'*. *R gpv gpv'*
    ⟹ *rel-spmf* (*rel-generat* (=) (=) (*rel-fun* (=) (λ*gpv gpv'*. *R gpv gpv'* ∨ *gpv* = *gpv'* ∨
      (∃ *gpv2* :: (*'b*, *'call*, *'ret*) *gpv*. ∃ *gpv2'* :: (*'c*, *'call*, *'ret*) *gpv*. ∃ *f f'*. *gpv* = *bind-gpv gpv2 f* ∧ *gpv'* = *bind-gpv gpv2' f'* ∧
        rel-gpv (λ*x y*. *R* (*f x*) (*f' y*)) (=) *gpv2 gpv2'*))))
      (*the-gpv gpv*) (*the-gpv gpv'*)
  shows *gpv* = *gpv'*
proof −
  fix *x y*
  define *gpv1* :: (*'b*, *'call*, *'ret*) *gpv*
    and *f* :: *'b* ⇒ (*'a*, *'call*, *'ret*) *gpv*
    and *gpv1'* :: (*'c*, *'call*, *'ret*) *gpv*
    and *f'* :: *'c* ⇒ (*'a*, *'call*, *'ret*) *gpv*
    where *gpv1* = *Done x*
      and *f* = (λ-. *gpv*)
      and *gpv1'* = *Done y*
      and *f'* = (λ-. *gpv'*)
  from ∗ **have** *rel-gpv* (λ*x y*. *R* (*f x*) (*f' y*)) (=) *gpv1 gpv1'*
    by(*simp add*: *gpv1-def gpv1'-def f-def f'-def*)
  then have *gpv1* ≫= *f* = *gpv1'* ≫= *f'*
  proof(*coinduction arbitrary*: *gpv1 gpv1' f f' rule*: *gpv.coinduct-strong*)

case (*Eq-gpv gpv1 gpv1′ f f′*)
from *Eq-gpv*[*simplified gpv.rel-sel*] **show** *?case* **unfolding** *bind-gpv.sel spmf-rel-map*
apply(*rule rel-spmf-bindI*)
subgoal for *generat generat′*
apply(*cases generat generat′ rule*: *generat.exhaust*[*case-product generat.exhaust*]; *clarsimp simp add*: *o-def spmf-rel-map generat.rel-map*)
subgoal premises *Pure* **for** *x y*
using *step*[*OF* ‹*R* (*f x*) (*f′ y*)›] **apply** −
apply(*assumption* | *rule rel-spmf-mono rel-generat-mono rel-fun-mono refl*)+
apply(*fastforce intro*: *exI*[**where** *x*=*Done* -])+
done
subgoal by(*fastforce simp add*: *rel-fun-def*)
done
done
qed
thus *?thesis* **by**(*simp add*: *gpv1-def gpv1′-def f-def f′-def*)
qed

Inlining one gpv into another. This may throw out arbitrarily many interactions between the two gpvs if the inlined one does not call its callee. So we define it as the coiteration of a least-fixpoint search operator.

context
fixes *callee* :: *′s* ⇒ *′call* ⇒ (*′ret* × *′s*, *′call′*, *′ret′*) *gpv*
notes [[*function-internals*]]
begin

partial-function (*spmf*) *inline1*
:: (*′a*, *′call*, *′ret*) *gpv* ⇒ *′s*
⇒ (*′a* × *′s* + *′call′* × (*′ret* × *′s*, *′call′*, *′ret′*) *rpv* × (*′a*, *′call*, *′ret*) *rpv*) *spmf*
where
inline1 gpv s =
the-gpv gpv ≫=
case-generat (λ*x*. *return-spmf* (*Inl* (*x*, *s*)))
(λ*out rpv*. *the-gpv* (*callee s out*) ≫=
case-generat (λ(*x*, *y*). *inline1* (*rpv x*) *y*)
(λ*out rpv′*. *return-spmf* (*Inr* (*out*, *rpv′*, *rpv*))))

lemma *inline1-unfold*:
inline1 gpv s =
the-gpv gpv ≫=
case-generat (λ*x*. *return-spmf* (*Inl* (*x*, *s*)))
(λ*out rpv*. *the-gpv* (*callee s out*) ≫=
case-generat (λ(*x*, *y*). *inline1* (*rpv x*) *y*)
(λ*out rpv′*. *return-spmf* (*Inr* (*out*, *rpv′*, *rpv*))))
by(*fact inline1.simps*)

lemma *inline1-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*inline1′*.

P (*λgpv s. inline1′* (*gpv*, *s*)))
and *P* (*λ- -. return-pmf None*)
and ⋀*inline1′*. *P inline1′* ⟹ *P* (*λgpv s. the-gpv gpv* ⤜ *case-generat* (*λx. return-spmf* (*Inl* (*x*, *s*))) (*λout rpv. the-gpv* (*callee s out*) ⤜ *case-generat* (*λ*(*x*, *y*). *inline1′* (*rpv x*) *y*) (*λout rpv′*. *return-spmf* (*Inr* (*out*, *rpv′*, *rpv*)))))
shows *P inline1*
using *assms* **by**(*rule inline1.fixp-induct*[*unfolded curry-conv*[*abs-def*]])

lemma *inline1-fixp-induct-strong* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (*λinline1′*. *P* (*λgpv s. inline1′* (*gpv*, *s*)))
and *P* (*λ- -. return-pmf None*)
and ⋀*inline1′*. ⟦ ⋀*gpv s. ord-spmf* (=) (*inline1′ gpv s*) (*inline1 gpv s*); *P inline1′* ⟧
⟹ *P* (*λgpv s. the-gpv gpv* ⤜ *case-generat* (*λx. return-spmf* (*Inl* (*x*, *s*))) (*λout rpv. the-gpv* (*callee s out*) ⤜ *case-generat* (*λ*(*x*, *y*). *inline1′* (*rpv x*) *y*) (*λout rpv′*. *return-spmf* (*Inr* (*out*, *rpv′*, *rpv*)))))
shows *P inline1*
using *assms* **by**(*rule spmf.fixp-strong-induct-uc*[**where** *P*=*λf*. *P* (*curry f*) **and** *U*=*case-prod* **and** *C*=*curry*, *OF inline1.mono inline1-def*, *simplified curry-case-prod*, *simplified curry-conv*[*abs-def*] *fun-ord-def split-paired-All prod.case case-prod-eta*, *OF refl*]) *blast+*

lemma *inline1-fixp-induct-strong2* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (*λinline1′*. *P* (*λgpv s. inline1′* (*gpv*, *s*)))
and *P* (*λ- -. return-pmf None*)
and ⋀*inline1′*.
⟦ ⋀*gpv s. ord-spmf* (=) (*inline1′ gpv s*) (*inline1 gpv s*);
⋀*gpv s. ord-spmf* (=) (*inline1′ gpv s*) (*the-gpv gpv* ⤜ *case-generat* (*λx. return-spmf* (*Inl* (*x*, *s*))) (*λout rpv. the-gpv* (*callee s out*) ⤜ *case-generat* (*λ*(*x*, *y*). *inline1′* (*rpv x*) *y*) (*λout rpv′*. *return-spmf* (*Inr* (*out*, *rpv′*, *rpv*)))));
P inline1′ ⟧
⟹ *P* (*λgpv s. the-gpv gpv* ⤜ *case-generat* (*λx. return-spmf* (*Inl* (*x*, *s*))) (*λout rpv. the-gpv* (*callee s out*) ⤜ *case-generat* (*λ*(*x*, *y*). *inline1′* (*rpv x*) *y*) (*λout rpv′*. *return-spmf* (*Inr* (*out*, *rpv′*, *rpv*)))))
shows *P inline1*
using *assms*
by(*rule spmf.fixp-induct-strong2-uc*[**where** *P*=*λf*. *P* (*curry f*) **and** *U*=*case-prod* **and** *C*=*curry*, *OF inline1.mono inline1-def*, *simplified curry-case-prod*, *simplified curry-conv*[*abs-def*] *fun-ord-def split-paired-All prod.case case-prod-eta*, *OF refl*]) *blast+*

Iterate *local.inline1* over all interactions. We'd like to use (⤜) before the recursive call, but primcorec does not support this. So we emulate (⤜) by effectively defining two mutually recursive functions (sum type in the argument) where the second is exactly (⤜) specialised to call *inline* in the bind.

primcorec *inline-aux*

:: ('a, 'call, 'ret) gpv × 's + ('ret ⇒ ('a, 'call, 'ret) gpv) × ('ret × 's, 'call', 'ret') gpv
⇒ ('a × 's, 'call', 'ret') gpv
where
⋀state. the-gpv (inline-aux state) =
(case state of Inl (c, s) ⇒ map-spmf (λresult.
case result of Inl (x, s) ⇒ Pure (x, s)
| Inr (out, oracle, rpv) ⇒ IO out (λinput. inline-aux (Inr (rpv, oracle input)))) (inline1 c s)
| Inr (rpv, c) ⇒
map-spmf (λresult.
case result of Inl (Inl (x, s)) ⇒ Pure (x, s)
| Inl (Inr (out, oracle, rpv)) ⇒ IO out (λinput. inline-aux (Inr (rpv, oracle input)))
| Inr (out, c) ⇒ IO out (λinput. inline-aux (Inr (rpv, c input))))
(bind-spmf (the-gpv c) (λgenerat. case generat of Pure (x, s') ⇒ (map-spmf Inl (inline1 (rpv x) s'))
| IO out c ⇒ return-spmf (Inr (out, c)))
))

declare *inline-aux.simps*[*simp del*]

definition *inline* :: ('a, 'call, 'ret) gpv ⇒ 's ⇒ ('a × 's, 'call', 'ret') gpv
where *inline c s = inline-aux (Inl (c, s))*

lemma *inline-aux-Inr*:
inline-aux (Inr (rpv, oracl)) = bind-gpv oracl (λ(x, s). inline (rpv x) s)
unfolding *inline-def*
apply(*coinduction arbitrary*: *oracl rule*: *gpv.coinduct-strong*)
apply(*simp add*: *inline-aux.sel bind-gpv.sel spmf-rel-map del*: *bind-gpv-sel'*)
apply(*rule rel-spmf-bindI*[**where** *R*=(=)])
apply(*auto simp add*: *spmf-rel-map inline-aux.sel rel-spmf-reflI generat.rel-map generat.rel-refl rel-fun-def split*: *generat.split*)
done

lemma *inline-sel*:
*the-gpv (inline c s) =
map-spmf (λresult. case result of Inl xs ⇒ Pure xs
| Inr (out, oracle, rpv) ⇒ IO out (λinput. bind-gpv (oracle input) (λ(x, s'). inline (rpv x) s'))) (inline1 c s)*
by(*simp add*: *inline-def inline-aux.sel inline-aux-Inr cong del*: *sum.case-cong*)

lemma *inline1-Fail* [*simp*]: *inline1 Fail s = return-pmf None*
by(*rewrite inline1.simps*) *simp*

lemma *inline-Fail* [*simp*]: *inline Fail s = Fail*
by(*rule gpv.expand*)(*simp add*: *inline-sel*)

lemma *inline1-Done* [*simp*]: *inline1 (Done x) s = return-spmf (Inl (x, s))*

by(*rewrite inline1.simps*) *simp*

lemma *inline-Done* [*simp*]: *inline* (*Done x*) *s* = *Done* (*x*, *s*)
by(*rule gpv.expand*)(*simp add*: *inline-sel*)

lemma *inline1-lift-spmf* [*simp*]: *inline1* (*lift-spmf p*) *s* = *map-spmf* (*λx. Inl* (*x*, *s*)) *p*
by(*rewrite inline1.simps*)(*simp add*: *bind-map-spmf o-def map-spmf-conv-bind-spmf*)

lemma *inline-lift-spmf* [*simp*]: *inline* (*lift-spmf p*) *s* = *lift-spmf* (*map-spmf* (*λx.* (*x*, *s*)) *p*)
by(*rule gpv.expand*)(*simp add*: *inline-sel spmf.map-comp o-def*)

lemma *inline1-Pause*:
inline1 (*Pause out c*) *s* =
the-gpv (*callee s out*) ⤜ (*λreact. case react of Pure* (*x*, *s'*) ⇒ *inline1* (*c x*) *s'* | *IO out' c'* ⇒ *return-spmf* (*Inr* (*out'*, *c'*, *c*)))
by(*rewrite inline1.simps*) *simp*

lemma *inline-Pause* [*simp*]:
inline (*Pause out c*) *s* = *callee s out* ⤜ (*λ*(*x*, *s'*). *inline* (*c x*) *s'*)
by(*rule gpv.expand*)(*auto simp add*: *inline-sel inline1-Pause map-spmf-bind-spmf bind-gpv.sel o-def*[*abs-def*] *spmf.map-comp generat.map-comp id-def generat.map-id*[*unfolded id-def*] *simp del*: *bind-gpv-sel'* *intro*!: *bind-spmf-cong*[*OF refl*] *split*: *generat.split*)

lemma *inline1-bind-gpv*:
fixes *gpv f s*
defines [*simp*]: *inline11* ≡ *inline1* **and** [*simp*]: *inline12* ≡ *inline1* **and** [*simp*]: *inline13* ≡ *inline1*
shows *inline11* (*bind-gpv gpv f*) *s* = *bind-spmf* (*inline12 gpv s*)
(*λres. case res of Inl* (*x*, *s'*) ⇒ *inline13* (*f x*) *s'* | *Inr* (*out*, *rpv'*, *rpv*) ⇒ *return-spmf* (*Inr* (*out*, *rpv'*, *bind-rpv rpv f*)))
(**is** *?lhs* = *?rhs*)
proof(*rule spmf.leq-antisym*)
note [*intro*!] = *ord-spmf-bind-reflI* **and** [*split*] = *generat.split*
show *ord-spmf* (=) *?lhs ?rhs* **unfolding** *inline11-def*
proof(*induction arbitrary*: *gpv s f rule*: *inline1-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1'*)
show *?case* **unfolding** *inline12-def*
apply(*rewrite inline1.simps*; *clarsimp simp add*: *bind-rpv-def*)
apply(*rule conjI*; *clarsimp*)
subgoal premises *Pure* **for** *x*
apply(*rewrite inline1.simps*; *clarsimp*)
subgoal for *out c ret s'* **using** *step.IH*[*of Done x λ-. c ret s'*] **by** *simp*
done
subgoal for *out c ret s'* **using** *step.IH*[*of c ret f s'*] **by**(*simp cong del*: *sum.case-cong-weak*)

done
qed
show *ord-spmf* (=) *?rhs ?lhs* **unfolding** *inline12-def*
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* **unfolding** *inline11-def*
apply(*rewrite inline1.simps*; *clarsimp simp add*: *bind-rpv-def*)
apply(*rule conjI*; *clarsimp*)
subgoal by(*rewrite inline1.simps*; *simp*)
subgoal for *out c ret s'* **using** *step.IH*[*of c ret s'*] **by**(*simp cong del*: *sum.case-cong-weak*)
done
qed
qed

lemma *inline-bind-gpv* [*simp*]:
inline (*bind-gpv gpv f*) *s* = *bind-gpv* (*inline gpv s*) (λ(*x*, *s'*). *inline* (*f x*) *s'*)
apply(*coinduction arbitrary*: *gpv s rule*: *gpv-coinduct-bind*)
apply(*clarsimp simp add*: *map-spmf-bind-spmf o-def*[*abs-def*] *bind-gpv.sel inline-sel bind-map-spmf inline1-bind-gpv simp del*: *bind-gpv-sel'* *intro*!: *rel-spmf-bind-reflI split*: *generat.split*)
apply(*rule conjI*)
subgoal by(*auto split*: *sum.split-asm simp add*: *spmf-rel-map spmf.map-comp o-def generat.map-comp generat.map-id*[*unfolded id-def*] *spmf.map-id*[*unfolded id-def*] *inline-sel intro*!: *rel-spmf-reflI generat.rel-refl fun.rel-refl*)
by(*auto split*: *sum.split-asm simp add*: *bind-gpv-assoc split-def intro*!: *gpv.rel-refl exI disjI2 rel-funI*)

end

lemma *set-inline1-lift-spmf1*: *set-spmf* (*inline1* (λ*s x*. *lift-spmf* (*p s x*)) *gpv s*) ⊆ *range Inl*
apply(*induction arbitrary*: *gpv s rule*: *inline1-fixp-induct*)
subgoal by(*rule cont-intro ccpo-class.admissible-leI*)+
apply(*auto simp add*: *o-def bind-UNION split*: *generat.split-asm*)+
done

lemma *in-set-inline1-lift-spmf1*: *y* ∈ *set-spmf* (*inline1* (λ*s x*. *lift-spmf* (*p s x*)) *gpv s*) ⟹ ∃ *r s'*. *y* = *Inl* (*r*, *s'*)
by(*drule set-inline1-lift-spmf1*[*THEN subsetD*]) *auto*

lemma *inline-lift-spmf1*:
fixes *p* **defines** *callee* ≡ λ*s c*. *lift-spmf* (*p s c*)
shows *inline callee gpv s* = *lift-spmf* (*map-spmf projl* (*inline1 callee gpv s*))
by(*rule gpv.expand*)(*auto simp add*: *inline-sel spmf.map-comp callee-def intro*!: *map-spmf-cong*[*OF refl*] *dest*: *in-set-inline1-lift-spmf1*)

context includes *lifting-syntax* **begin**
lemma *inline1-parametric'*:
((*S* ===> *C* ===> *rel-gpv''* (*rel-prod R S*) *C' R'*) ===> *rel-gpv'' A C R* ===> *S*
===> *rel-spmf* (*rel-sum* (*rel-prod A S*) (*rel-prod C'* (*rel-prod* (*R'* ===> *rel-gpv''* (*rel-prod R S*) *C' R'*) (*R* ===> *rel-gpv'' A C R*)))))
inline1 inline1
(**is** (- ===> *?R*) - -)
proof(*rule rel-funI*)
note [*transfer-rule*] = *the-gpv-parametric'*
show *?R* (*inline1 callee*) (*inline1 callee'*)
if [*transfer-rule*]: (*S* ===> *C* ===> *rel-gpv''* (*rel-prod R S*) *C' R'*) *callee callee'*
for *callee callee'*
unfolding *inline1-def*
by(*unfold rel-fun-curry case-prod-curry*)(*rule fixp-spmf-parametric*[*OF inline1.mono inline1.mono*]; *transfer-prover*)
qed

lemma *inline1-parametric* [*transfer-rule*]:
((*S* ===> *C* ===> *rel-gpv* (*rel-prod* (=) *S*) *C'*) ===> *rel-gpv A C* ===> *S*
===> *rel-spmf* (*rel-sum* (*rel-prod A S*) (*rel-prod C'* (*rel-prod* (*rel-rpv* (*rel-prod* (=) *S*) *C'*) (*rel-rpv A C*)))))
inline1 inline1
unfolding *rel-gpv-conv-rel-gpv''* **by**(*rule inline1-parametric'*)

lemma *inline-parametric'*:
notes [*transfer-rule*] = *inline1-parametric' the-gpv-parametric' corec-gpv-parametric'*
shows ((*S* ===> *C* ===> *rel-gpv''* (*rel-prod R S*) *C' R'*) ===> *rel-gpv'' A C R* ===> *S* ===> *rel-gpv''* (*rel-prod A S*) *C' R'*)
inline inline
unfolding *inline-def*[*abs-def*] *inline-aux-def*

apply(*rule rel-funI*)+
subgoal premises [*transfer-rule*] **by** *transfer-prover*
done

lemma *inline-parametric* [*transfer-rule*]:
((*S* ===> *C* ===> *rel-gpv* (*rel-prod* (=) *S*) *C'*) ===> *rel-gpv A C* ===> *S*
===> *rel-gpv* (*rel-prod A S*) *C'*)
inline inline
unfolding *rel-gpv-conv-rel-gpv''* **by**(*rule inline-parametric'*)
end

Associativity rule for *inline*

context
fixes *callee1* :: *'s1* ⇒ *'c1* ⇒ (*'r1* × *'s1*, *'c*, *'r*) *gpv*
and *callee2* :: *'s2* ⇒ *'c2* ⇒ (*'r2* × *'s2*, *'c1*, *'r1*) *gpv*
begin

partial-function (*spmf*) *inline2* :: (′*a*, ′*c2*, ′*r2*) *gpv* ⇒ ′*s2* ⇒ ′*s1*
⇒ (′*a* × (′*s2* × ′*s1*) + ′*c* × (′*r1* × ′*s1*, ′*c*, ′*r*) *rpv* × (′*r2* × ′*s2*, ′*c1*, ′*r1*) *rpv* × (′*a*, ′*c2*, ′*r2*) *rpv*) *spmf*
where
inline2 gpv s2 s1 =
bind-spmf (*the-gpv gpv*)
(*case-generat* (λ*x*. *return-spmf* (*Inl* (*x*, *s2*, *s1*)))
(λ*out rpv*. *bind-spmf* (*inline1 callee1* (*callee2 s2 out*) *s1*)
(*case-sum* (λ((*r2*, *s2*), *s1*). *inline2* (*rpv r2*) *s2 s1*)
(λ(*x*, *rpv*′′, *rpv*′). *return-spmf* (*Inr* (*x*, *rpv*′′, *rpv*′, *rpv*))))))

lemma *inline2-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*inline2*. *P* (λ*gpv s2 s1*. *inline2* ((*gpv*, *s2*), *s1*)))
and *P* (λ- - -. *return-pmf None*)
and ⋀*inline2*′. *P inline2*′ ⟹
P (λ*gpv s2 s1*. *bind-spmf* (*the-gpv gpv*) (λ*generat*. *case generat of*
Pure x ⇒ *return-spmf* (*Inl* (*x*, *s2*, *s1*))
| *IO out rpv* ⇒ *bind-spmf* (*inline1 callee1* (*callee2 s2 out*) *s1*) (λ*lr*. *case lr of*
Inl ((*r2*, *s2*), *c*) ⇒ *inline2*′ (*rpv r2*) *s2 c*
| *Inr* (*x*, *rpv*′′, *rpv*′) ⇒ *return-spmf* (*Inr* (*x*, *rpv*′′, *rpv*′, *rpv*)))))
shows *P inline2*
using *assms* **unfolding** *split-def* **by**(*rule inline2.fixp-induct*[*unfolded curry-conv*[*abs-def*] *split-def*])

lemma *inline1-inline-conv-inline2*:
fixes *gpv*′ :: (′*r2* × ′*s2*, ′*c1*, ′*r1*) *gpv*
shows *inline1 callee1* (*inline callee2 gpv s2*) *s1* =
map-spmf (*map-sum* (λ(*x*, (*s2*, *s1*)). ((*x*, *s2*), *s1*))
(λ(*x*, *rpv*′′, *rpv*′, *rpv*). (*x*, *rpv*′′, λ*r1*. *rpv*′ *r1* ⤜ (λ(*r2*, *s2*). *inline callee2* (*rpv r2*) *s2*))))
(*inline2 gpv s2 s1*)
(**is** *?lhs* = *?rhs*)
proof(*rule spmf.leq-antisym*)
define *inline1-1* :: (′*s1* ⇒ ′*c1* ⇒ (′*r1* × ′*s1*, ′*c*, ′*r*) *gpv*) ⇒ (′*r2* × ′*s2*, ′*c1*, ′*r1*) *gpv* ⇒ ′*s1* ⇒ -
where *inline1-1* = *inline1*
have *ord-spmf* (=) *?lhs ?rhs*
— We need in the inductive step that the approximation behaves well with (⤜) because of *inline-aux-Inr*. So we have to thread it through the induction and do one half of the proof from *inline1-bind-gpv* again. We cannot inline *inline1-bind-gpv* in this proof here because the types are too specific.
and *ord-spmf* (=) (*inline1 callee1* (*gpv*′ ⤜ *f*) *s1*′)
(*do* {
res ← *inline1-1 callee1 gpv*′ *s1*′;
case res of Inl (*x*, *s*′) ⇒ *inline1 callee1* (*f x*) *s*′
| *Inr* (*out*, *rpv*′, *rpv*) ⇒ *return-spmf* (*Inr* (*out*, *rpv*′, *rpv* ⤜ *f*))

}) **for** *gpv'* **and** *f* :: - ⇒ ('*a* × '*s2*, '*c1*, '*r1*) *gpv* **and** *s1'*
proof(*induction arbitrary*: *gpv s2 s1 gpv' f s1' rule*: *inline1-fixp-induct-strong2*)
case *adm* **thus** *?case*
apply(*rule cont-intro*)+
subgoal for *a b c d* **by**(*cases d*; *clarsimp*)
done

case (*step inline1'*)
note *step-IH* = *step.IH*[*unfolded inline1-1-def*] **and** *step-hyps* = *step.hyps*[*unfolded inline1-1-def*]
{ case *1*
have *inline1*: *ord-spmf* (=)
(*inline1 callee2 gpv s2* ⤜ (λ*lr*. *case lr of Inl as2* ⇒ *return-spmf* (*Inl* (*as2*, *s1*))
| *Inr* (*out1*, *rpv'*, *rpv*) ⇒ *the-gpv* (*callee1 s1 out1*) ⤜ (λ*generat*. *case generat of*
Pure (*r1*, *s1*) ⇒ *inline1'* (*bind-gpv* (*rpv' r1*) (λ(*r2*, *s2*). *inline callee2* (*rpv r2*) *s2*)) *s1*
| *IO out rpv''* ⇒ *return-spmf* (*Inr* (*out*, *rpv''*, λ*r1*. *bind-gpv* (*rpv' r1*) (λ(*r2*, *s2*). *inline callee2* (*rpv r2*) *s2*))))))
(*the-gpv gpv* ⤜ (λ*generat*. *case generat of Pure x* ⇒ *return-spmf* (*Inl* ((*x*, *s2*), *s1*))
| *IO out2 rpv* ⇒ *inline1-1 callee1* (*callee2 s2 out2*) *s1* ⤜ (λ*lr*. *case lr of*
Inl ((*r2*, *s2*), *s1*) ⇒
map-spmf (*map-sum* (λ(*x*, *s2*, *s1*). ((*x*, *s2*), *s1*)) (λ(*x*, *rpv''*, *rpv'*, *rpv*). (*x*, *rpv''*, λ*r1*. *bind-gpv* (*rpv' r1*) (λ(*r2*, *s2*). *inline callee2* (*rpv r2*) *s2*))))
(*inline2* (*rpv r2*) *s2 s1*)
| *Inr* (*out*, *rpv''*, *rpv'*) ⇒
return-spmf (*Inr* (*out*, *rpv''*, λ*r1*. *bind-gpv* (*rpv' r1*) (λ(*r2*, *s2*). *inline callee2* (*rpv r2*) *s2*))))))
proof(*induction arbitrary*: *gpv s2 s1 rule*: *inline1-fixp-induct*)
case *step2*: (*step inline1''*)
note *step2-IH* = *step2.IH*[*unfolded inline1-1-def*]

show *?case* **unfolding** *inline1-1-def*
apply(*rewrite* **in** *ord-spmf* - - ⌑ *inline1.simps*)
apply(*clarsimp intro*!: *ord-spmf-bind-reflI split*: *generat.split*)
apply(*rule conjI*)
subgoal by(*rewrite* **in** *ord-spmf* - - ⌑ *inline2.simps*)(*clarsimp simp add*: *map-spmf-bind-spmf o-def split*: *generat.split sum.split intro*!: *ord-spmf-bind-reflI spmf.leq-trans*[*OF step2-IH*])
subgoal by(*clarsimp intro*!: *ord-spmf-bind-reflI step-IH*[*THEN spmf.leq-trans*] *split*: *generat.split sum.split simp add*: *bind-rpv-def*)
done
qed *simp-all*
show *?case*
apply(*rewrite* **in** *ord-spmf* - ⌑ - *inline-sel*)
apply(*rewrite* **in** *ord-spmf* - - ⌑ *inline2.simps*)
apply(*clarsimp simp add*: *map-spmf-bind-spmf bind-map-spmf o-def intro*!:

ord-spmf-bind-reflI split: *generat.split*)
apply(*rule spmf.leq-trans*[*OF spmf.leq-trans*, *OF* - *inline1*])
apply(*auto intro*!: *ord-spmf-bind-reflI split*: *sum.split generat.split simp add*: *inline1-1-def map-spmf-bind-spmf*)
done **}**
{ **case** *2*
show *?case* **unfolding** *inline1-1-def*
by(*rewrite inline1.simps*)(*auto simp del*: *bind-gpv-sel′ simp add*: *bind-gpv.sel map-spmf-bind-spmf bind-map-spmf o-def bind-rpv-def intro*!: *ord-spmf-bind-reflI step-IH*(*2*)[*THEN spmf.leq-trans*] *step-hyps*(*2*) *split*: *generat.split sum.split*) **}**
qed *simp-all*
thus *ord-spmf* (=) *?lhs ?rhs* **by** −

show *ord-spmf* (=) *?rhs ?lhs*
proof(*induction arbitrary*: *gpv s2 s1 rule*: *inline2-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline2′*)
show *?case*
apply(*rewrite* **in** *ord-spmf* - - ⌑ *inline1.simps*)
apply(*rewrite inline-sel*)
apply(*rewrite* **in** *ord-spmf* - ⌑ - *inline1.simps*)
apply(*rewrite* **in** *ord-spmf* - - ⌑ *inline1.simps*)
apply(*clarsimp simp add*: *map-spmf-bind-spmf bind-map-spmf intro*!: *ord-spmf-bind-reflI split*: *generat.split*)
apply(*rule conjI*)
subgoal
apply *clarsimp*
apply(*rule step.IH*[*THEN spmf.leq-trans*])
apply(*rewrite* **in** *ord-spmf* - ⌑ - *inline1.simps*)
apply(*rewrite inline-sel*)
apply(*simp add*: *bind-map-spmf*)
done
subgoal **by**(*clarsimp intro*!: *ord-spmf-bind-reflI split*: *generat.split sum.split simp add*: *o-def inline1-bind-gpv bind-rpv-def step.IH*)
done
qed
qed

lemma *inline1-inline-conv-inline2′*:
inline1 (λ(*s2*, *s1*) *c2*. *map-gpv* (λ((*r*, *s2*), *s1*). (*r*, *s2*, *s1*)) *id* (*inline callee1* (*callee2 s2 c2*) *s1*)) *gpv* (*s2*, *s1*) =
map-spmf (*map-sum id* (λ(*x*, *rpv″*, *rpv′*, *rpv*). (*x*, λ*r*. *bind-gpv* (*rpv″ r*)
(λ(*r1*, *s1*). *map-gpv* (λ((*r2*, *s2*), *s1*). (*r2*, *s2*, *s1*)) *id* (*inline callee1* (*rpv′ r1*) *s1*)), *rpv*)))
(*inline2 gpv s2 s1*)
(**is** *?lhs* = *?rhs*)
proof(*rule spmf.leq-antisym*)
show *ord-spmf* (=) *?lhs ?rhs*

proof(*induction arbitrary*: *gpv s2 s1 rule*: *inline1-fixp-induct*)
case (*step inline1'*) **show** *?case*
by(*rewrite inline2.simps*)(*auto simp add*: *map-spmf-bind-spmf o-def inline-sel gpv.map-sel bind-map-spmf id-def*[*symmetric*] *gpv.map-id map-gpv-bind-gpv split-def intro*!: *ord-spmf-bind-reflI step.IH*[*THEN spmf.leq-trans*] *split*: *generat.split sum.split*)
qed *simp-all*
show *ord-spmf* (=) *?rhs ?lhs*
proof(*induction arbitrary*: *gpv s2 s1 rule*: *inline2-fixp-induct*)
case (*step inline2'*)
show *?case*
apply(*rewrite* **in** *ord-spmf - - ⌑ inline1.simps*)
apply(*clarsimp simp add*: *map-spmf-bind-spmf bind-rpv-def o-def gpv.map-sel bind-map-spmf inline-sel map-gpv-bind-gpv id-def*[*symmetric*] *gpv.map-id split-def split*: *generat.split sum.split intro*!: *ord-spmf-bind-reflI*)
apply(*rule spmf.leq-trans*[*OF spmf.leq-trans*, *OF - step.IH*])
apply(*auto simp add*: *split-def id-def*[*symmetric*] *intro*!: *ord-spmf-reflI*)
done
qed *simp-all*
qed

lemma *inline-assoc*:
inline callee1 (*inline callee2 gpv s2*) *s1* =
map-gpv (λ(*r*, *s2*, *s1*). ((*r*, *s2*), *s1*)) *id* (*inline* (λ(*s2*, *s1*) *c2*. *map-gpv* (λ((*r*, *s2*), *s1*). (*r*, *s2*, *s1*)) *id* (*inline callee1* (*callee2 s2 c2*) *s1*)) *gpv* (*s2*, *s1*))
proof(*coinduction arbitrary*: *s2 s1 gpv rule*: *gpv-coinduct-bind*[**where** *?'b* = (*'r2* × *'s2*) × *'s1* **and** *?'c* = (*'r2* × *'s2*) × *'s1*])
case (*Eq-gpv s2 s1 gpv*)
have ∃ *gpv2 gpv2'* (*f* :: (*'r2* × *'s2*) × *'s1* ⇒ -) (*f'* :: (*'r2* × *'s2*) × *'s1* ⇒ -).
bind-gpv (*bind-gpv* (*rpv'' r*) (λ(*r1*, *s1*). *inline callee1* (*rpv' r1*) *s1*)) (λ((*r2*, *s2*), *s1*). *inline callee1* (*inline callee2* (*rpv r2*) *s2*) *s1*) = *gpv2* ⤜ *f* ∧
bind-gpv (*bind-gpv* (*rpv'' r*) (λ(*r1*, *s1*). *inline callee1* (*rpv' r1*) *s1*)) (λ((*r2*, *s2*), *s1*). *map-gpv* (λ(*r*, *s2*, *y*). ((*r*, *s2*), *y*)) *id* (*inline* (λ(*s2*, *s1*) *c2*. *map-gpv* (λ((*r*, *s2*), *s1*). (*r*, *s2*, *s1*)) *id* (*inline callee1* (*callee2 s2 c2*) *s1*)) (*rpv r2*) (*s2*, *s1*))) = *gpv2'* ⤜ *f'* ∧
rel-gpv (λ*x y*. ∃ *s2 s1 gpv*. *f x* = *inline callee1* (*inline callee2 gpv s2*) *s1* ∧
f' y = *map-gpv* (λ(*r*, *s2*, *y*). ((*r*, *s2*), *y*)) *id* (*inline* (λ(*s2*, *s1*) *c2*. *map-gpv* (λ((*r*, *s2*), *s1*). (*r*, *s2*, *s1*)) *id* (*inline callee1* (*callee2 s2 c2*) *s1*)) *gpv* (*s2*, *s1*)))
(=) *gpv2 gpv2'*
for *rpv''* :: (*'r1* × *'s1*, *'c*, *'r*) *rpv* **and** *rpv'* :: (*'r2* × *'s2*, *'c1*, *'r1*) *rpv* **and** *rpv* :: (*'a*, *'c2*, *'r2*) *rpv* **and** *r* :: *'r*
by(*auto intro*!: *exI gpv.rel-refl*)
then show *?case*
apply(*subst inline-sel*)
apply(*subst gpv.map-sel*)
apply(*subst inline-sel*)
apply(*subst inline1-inline-conv-inline2*)
apply(*subst inline1-inline-conv-inline2'*)
apply(*unfold spmf.map-comp o-def case-sum-map-sum spmf-rel-map generat.rel-map*)

apply(*rule rel-spmf-reflI*)
subgoal for *lr* **by**(*cases lr*)(*auto del*: *disjCI intro*!: *rel-funI disjI2 simp add*: *split-def map-gpv-conv-bind*[*folded id-def*] *bind-gpv-assoc*)
done
qed

end

lemma *set-inline2-lift-spmf1*: *set-spmf* (*inline2* (λ*s x*. *lift-spmf* (*p s x*)) *callee gpv s s'*) $\subseteq$ *range Inl*
apply(*induction arbitrary*: *gpv s s' rule*: *inline2-fixp-induct*)
subgoal by(*rule cont-intro ccpo-class.admissible-leI*)+
apply(*auto simp add*: *o-def bind-UNION split*: *generat.split-asm sum.split-asm dest*!: *in-set-inline1-lift-spmf1*)
apply *blast*
done

lemma *in-set-inline2-lift-spmf1*: *y* $\in$ *set-spmf* (*inline2* (λ*s x*. *lift-spmf* (*p s x*)) *callee gpv s s'*) $\Longrightarrow$ $\exists$ *r s s'*. *y* = *Inl* (*r*, *s*, *s'*)
by(*drule set-inline2-lift-spmf1*[*THEN subsetD*]) *auto*

context
fixes *consider'* :: *'call* $\Rightarrow$ *bool*
and *consider* :: *'call'* $\Rightarrow$ *bool*
and *callee* :: *'s* $\Rightarrow$ *'call* $\Rightarrow$ (*'ret* $\times$ *'s*, *'call'*, *'ret'*) *gpv*
notes [[*function-internals*]]
begin

private partial-function (*spmf*) *inline1'*
:: (*'a*, *'call*, *'ret*) *gpv* $\Rightarrow$ *'s*
$\Rightarrow$ (*'a* $\times$ *'s* + *'call* $\times$ *'call'* $\times$ (*'ret* $\times$ *'s*, *'call'*, *'ret'*) *rpv* $\times$ (*'a*, *'call*, *'ret*) *rpv*) *spmf*
where
inline1' gpv s =
the-gpv gpv $\gg$=
case-generat (λ*x*. *return-spmf* (*Inl* (*x*, *s*)))
(λ*out rpv*. *the-gpv* (*callee s out*) $\gg$=
case-generat (λ(*x*, *y*). *inline1'* (*rpv x*) *y*)
(λ*out' rpv'*. *return-spmf* (*Inr* (*out*, *out'*, *rpv'*, *rpv*))))

private lemma *inline1'-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*inline1'*. *P* (λ*gpv s*. *inline1'* (*gpv*, *s*)))
and *P* (λ- -. *return-pmf None*)
and $\bigwedge$*inline1'*. *P inline1'* $\Longrightarrow$ *P* (λ*gpv s*. *the-gpv gpv* $\gg$= *case-generat* (λ*x*. *return-spmf* (*Inl* (*x*, *s*))) (λ*out rpv*. *the-gpv* (*callee s out*) $\gg$= *case-generat* (λ(*x*, *y*). *inline1'* (*rpv x*) *y*) (λ*out' rpv'*. *return-spmf* (*Inr* (*out*, *out'*, *rpv'*, *rpv*)))))
shows *P inline1'*
using *assms* **by**(*rule inline1'.fixp-induct*[*unfolded curry-conv*[*abs-def*]])

private lemma *inline1-conv-inline1′*: *inline1 callee gpv s* = *map-spmf* (*map-sum id snd*) (*inline1′ gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf inline1.mono inline1′.mono inline1-def inline1′-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
  case *adm* **show** *?case* **by** *simp*
  case *bottom* **show** *?case* **by** *simp*
  case (*step inline1 inline1′*)
 thus *?case* **by**(*clarsimp simp add*: *map-spmf-bind-spmf o-def intro*!: *bind-spmf-cong*[*OF refl*] *split*: *generat.split*)
qed

context
  fixes *q* :: *enat*
  assumes *q*: $\bigwedge$*s x*. *consider′ x* $\Longrightarrow$ *interaction-bound consider* (*callee s x*) ≤ *q*
  and *ignore*: $\bigwedge$*s x*. ¬ *consider′ x* $\Longrightarrow$ *interaction-bound consider* (*callee s x*) = *0*
begin

private lemma *interaction-bound-inline1′-aux*:
  interaction-bound consider′ gpv ≤ *p*
  $\Longrightarrow$ *set-spmf* (*inline1′ gpv s*) ⊆ {*Inr* (*out′*, *out*, *c′*, *rpv*) | *out′ out c′ rpv*.
      if consider′ out′
        then (∀ *input*. (*if consider out then eSuc* (*interaction-bound consider* (*c′ input*)) *else interaction-bound consider* (*c′ input*)) ≤ *q*) ∧
            (∀ *x*. *eSuc* (*interaction-bound consider′* (*rpv x*)) ≤ *p*)
      else ¬ *consider out* ∧ (∀ *input*. *interaction-bound consider* (*c′ input*) = *0*) ∧ (∀ *x*. *interaction-bound consider′* (*rpv x*) ≤ *p*)}
    ∪ *range Inl*
proof(*induction arbitrary*: *gpv s rule*: *inline1′-fixp-induct*)
  { case *adm* **show** *?case* **by**(*rule cont-intro ccpo-class.admissible-leI*)+ **}**
  { case *bottom* **show** *?case* **by** *simp* **}**
  case (*step inline1′*)
 have ∗: *interaction-bound consider′* (*c input*) ≤ *p* **if** *IO out c* ∈ *set-spmf* (*the-gpv gpv*) **for** *out c input*
    by(*cases consider′ out*)(*auto intro*: *interaction-bound-IO-consider*[*OF that*, *THEN order-trans*, *THEN order-trans*[*OF ile-eSuc*]] *interaction-bound-IO-ignore*[*OF that*, *THEN order-trans*] *step.prems*)
  have ∗∗: *if consider′ out′*
    then (∀ *input*. (*if consider out then eSuc* (*interaction-bound consider* (*c input*)) *else interaction-bound consider* (*c input*)) ≤ *q*) ∧
        (∀ *x*. *eSuc* (*interaction-bound consider′* (*rpv x*)) ≤ *p*)
    else ¬ *consider out* ∧ (∀ *input*. *interaction-bound consider* (*c input*) = *0*) ∧ (∀ *x*. *interaction-bound consider′* (*rpv x*) ≤ *p*)
    if *IO out′ rpv* ∈ *set-spmf* (*the-gpv gpv*) *IO out c* ∈ *set-spmf* (*the-gpv* (*callee s out′*))
    for *out′ rpv out c*
  proof(*cases consider′ out′*)
    case *True*

then show *?thesis* **using** *that q*
by(*auto split del*: *if-split intro*!: *interaction-bound-IO*[*THEN order-trans*] *interaction-bound-IO-consider*[*THEN order-trans*] *step.prems*)
next
case *False*
have ¬ *consider out interaction-bound consider* (*c input*) = *0* **for** *input*
using *interaction-bound-IO*[*OF that*(*2*), *of consider input*] *ignore*[*OF False*, *of s*]
by(*auto split*: *if-split-asm*)
then show *?thesis* **using** *False that*
by(*auto split del*: *if-split intro*: *interaction-bound-IO-ignore*[*THEN order-trans*] *step.prems*)
qed
show *?case*
by(*auto 6 4 simp add*: *bind-UNION del*: *subsetI intro*!: *UN-least intro*: *step.IH* ∗ ∗∗ *split*: *generat.split split del*: *if-split*)
qed

lemma *interaction-bound-inline1′*:
⟦ *Inr* (*out′*, *out*, *c′*, *rpv*) ∈ *set-spmf* (*inline1′ gpv s*); *interaction-bound consider′ gpv* ≤ *p* ⟧
⟹ *if consider′ out′ then*
(*if consider out then eSuc* (*interaction-bound consider* (*c′ input*)) *else interaction-bound consider* (*c′ input*)) ≤ *q* ∧
eSuc (*interaction-bound consider′* (*rpv x*)) ≤ *p*
else ¬ *consider out* ∧ *interaction-bound consider* (*c′ input*) = *0* ∧ *interaction-bound consider′* (*rpv x*) ≤ *p*
using *interaction-bound-inline1′-aux*[**where** *gpv*=*gpv* **and** *p*=*p* **and** *s*=*s*] **by**(*auto split*: *if-split-asm*)

end

lemma *interaction-bounded-by-inline1*:
⟦ *Inr* (*out′*, *out*, *c′*, *rpv*) ∈ *set-spmf* (*inline1′ gpv s*);
interaction-bounded-by consider′ gpv p;
⋀*s x*. *consider′ x* ⟹ *interaction-bounded-by consider* (*callee s x*) *q*;
⋀*s x*. ¬ *consider′ x* ⟹ *interaction-bounded-by consider* (*callee s x*) *0* ⟧
⟹ *if consider′ out′ then*
(*if consider out then q* ≠ *0* ∧ *interaction-bounded-by consider* (*c′ input*) (*q* − *1*) *else interaction-bounded-by consider* (*c′ input*) *q*) ∧
p ≠ *0* ∧ *interaction-bounded-by consider′* (*rpv x*) (*p* − *1*)
else ¬ *consider out* ∧ *interaction-bounded-by consider* (*c′ input*) *0* ∧ *interaction-bounded-by consider′* (*rpv x*) *p*
unfolding *interaction-bounded-by-0* **unfolding** *interaction-bounded-by.simps*
apply(*drule* (*1*) *interaction-bound-inline1′*[**where** *input*=*input* **and** *x*=*x*, *rotated 2*], *assumption*, *assumption*)
apply(*cases p q rule*: *co.enat.exhaust*[*case-product co.enat.exhaust*])
apply(*simp-all add*: *zero-enat-def*[*symmetric*] *eSuc-enat*[*symmetric*] *split*: *if-split-asm*)
done

declare *enat-0-iff* [*simp*]

lemma *interaction-bounded-by-inline* [*interaction-bound*]:
assumes *p*: *interaction-bounded-by consider′ gpv p*
and *q*: ⋀*s x*. *consider′ x* ⟹ *interaction-bounded-by consider* (*callee s x*) *q*
and *ignore*: ⋀*s x*. ¬ *consider′ x* ⟹ *interaction-bounded-by consider* (*callee s x*) *0*
shows *interaction-bounded-by consider* (*inline callee gpv s*) (*p* ∗ *q*)
proof
have *interaction-bounded-by consider′ gpv p* ⟹ *interaction-bound consider* (*inline callee gpv s*) ≤ *p* ∗ *q*
and *interaction-bound consider* (*bind-gpv gpv′ f*) ≤ *interaction-bound consider gpv′* + (*SUP x*∈*results′-gpv gpv′*. *interaction-bound consider* (*f x*))
for *gpv′* **and** *f* :: *′ret* × *′s* ⇒ (*′a* × *′s*, *′call′*, *′ret′*) *gpv*
proof(*induction arbitrary*: *gpv s p gpv′ f rule*: *interaction-bound-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **case** *1* **show** *?case* **by** *simp*
case (*step interaction-bound′*) **case** *step*: *1*
show *?case* (**is** (*SUP generat*∈*?inline*. *?lhs generat*) ≤ *?rhs*)
proof(*rule SUP-least*)
fix *generat*
assume *generat* ∈ *?inline*
then consider (*Pure*) *ret s′* **where** *generat* = *Pure* (*ret*, *s′*)
and *Inl* (*ret*, *s′*) ∈ *set-spmf* (*inline1 callee gpv s*)
| (*IO*) *out c rpv* **where** *generat* = *IO out* (λ*input*. *bind-gpv* (*c input*) (λ(*ret*, *s′*). *inline callee* (*rpv ret*) *s′*))
and *Inr* (*out*, *c*, *rpv*) ∈ *set-spmf* (*inline1 callee gpv s*)
by(*clarsimp simp add*: *inline-sel split*: *sum.split-asm*)
then show *?lhs generat* ≤ *?rhs*
proof(*cases*)
case *Pure* **thus** *?thesis* **by** *simp*
next
case *IO*
from *IO*(*2*) **obtain** *out′* **where** *out′*: *Inr* (*out′*, *out*, *c*, *rpv*) ∈ *set-spmf* (*inline1′ gpv s*)
by(*auto simp add*: *inline1-conv-inline1′ Inr-eq-map-sum-iff*)
show *?thesis*
proof(*cases consider′ out′*)
case *True*
with *interaction-bounded-by-inline1*[*OF out′ step.prems q ignore*]
have *p*: *p* ≠ *0* **and** *rpv*: ⋀*x*. *interaction-bounded-by consider′* (*rpv x*) (*p* − *1*)
and *c*: ⋀*input*. *if consider out then q* ≠ *0* ∧ *interaction-bounded-by consider* (*c input*) (*q* − *1*) *else interaction-bounded-by consider* (*c input*) *q*
by *auto*

have *?lhs generat* ≤ (*if consider out then 1 else 0*) + (*SUP input*. *interaction-bound′* (*bind-gpv* (*c input*) (λ(*ret*, *s′*). *inline callee* (*rpv ret*) *s′*)))

(**is** - ≤ - + *?sup*)
using *IO*(*1*) **by**(*auto simp add*: *plus-1-eSuc*)
also have *?sup* ≤ (*SUP input*. *interaction-bound consider* (*c input*) + (*SUP* (*ret*, *s′*) ∈ *results′-gpv* (*c input*). *interaction-bound′* (*inline callee* (*rpv ret*) *s′*)))
unfolding *split-def* **by**(*rule SUP-mono*)(*blast intro*: *step.IH*)
also have ... ≤ (*SUP input*. *interaction-bound consider* (*c input*) + (*SUP* (*ret*, *s′*) ∈ *results′-gpv* (*c input*). (*p* − *1*) ∗ *q*))
using *rpv* **by**(*auto intro*!: *SUP-mono rev-bexI add-mono step.IH*)
also have ... ≤ (*SUP input*. *interaction-bound consider* (*c input*) + (*p* − *1*) ∗ *q*)
apply(*auto simp add*: *SUP-constant bot-enat-def intro*!: *SUP-mono*)
apply(*metis add.right-neutral add-mono i0-lb order-refl*)+
done
also have ... ≤ (*SUP input* :: *′ret′*. (*if consider out then q* − *1 else q*) + (*p* − *1*) ∗ *q*)
apply(*rule SUP-mono rev-bexI UNIV-I add-mono*)+
using *c*
apply(*auto simp add*: *interaction-bounded-by.simps*)
done
also have ... = (*if consider out then q* − *1 else q*) + (*p* − *1*) ∗ *q*
by(*simp add*: *SUP-constant*)
finally show *?thesis*
apply(*rule order-trans*)
prefer *5*
using *p c*
apply(*cases p*; *cases q*)
apply(*auto simp add*: *one-enat-def algebra-simps Suc-leI*)
done
next
case *False*
with *interaction-bounded-by-inline1*[*OF out′ step.prems q ignore*]
have *out*: ¬ *consider out* **and** *zero*: ⋀*input*. *interaction-bounded-by consider* (*c input*) *0*
and *rpv*: ⋀*x*. *interaction-bounded-by consider′* (*rpv x*) *p* **by** *auto*
have *?lhs generat* ≤ (*SUP input*. *interaction-bound′* (*bind-gpv* (*c input*) (λ(*ret*, *s′*). *inline callee* (*rpv ret*) *s′*)))
using *IO*(*1*) *out* **by** *auto*
also have ... ≤ (*SUP input*. *interaction-bound consider* (*c input*) + (*SUP* (*ret*, *s′*) ∈ *results′-gpv* (*c input*). *interaction-bound′* (*inline callee* (*rpv ret*) *s′*)))
unfolding *split-def* **by**(*rule SUP-mono*)(*blast intro*: *step.IH*)
also have ... ≤ (*SUP input*. (*SUP* (*ret*, *s′*) ∈ *results′-gpv* (*c input*). *p* ∗ *q*))
using *rpv zero* **by**(*auto intro*!: *SUP-mono rev-bexI add-mono step.IH simp add*: *interaction-bounded-by-0*)
also have ... ≤ (*SUP input* :: *′ret′*. *p* ∗ *q*)
by(*rule SUP-mono rev-bexI*)+(*auto simp add*: *SUP-constant*)
also have ... = *p* ∗ *q* **by**(*simp add*: *SUP-constant*)
finally show *?thesis* **.**

qed
qed
qed
next
case *bottom* **case** *2* **show** *?case* **by** *simp*
case *step* **case** *2* **show** *?case* **using** *step* **by** $-$(*rule interaction-bound-bind-step*)
qed
then show *interaction-bound consider* (*inline callee gpv s*) $\leq p * q$ **using** *p* **by** $-$
qed

end

lemma *interaction-bounded-by-inline-invariant*:
includes *lifting-syntax*
fixes *consider'* :: *'call* ⇒ *bool*
and *consider* :: *'call'* ⇒ *bool*
and *callee* :: *'s* ⇒ *'call* ⇒ (*'ret* × *'s*, *'call'*, *'ret'*) *gpv*
and *gpv* :: (*'a*, *'call*, *'ret*) *gpv*
assumes *p*: *interaction-bounded-by consider' gpv p*
and *q*: ⋀*s x*. ⟦ *I s*; *consider' x* ⟧ ⟹ *interaction-bounded-by consider* (*callee s x*) *q*
and *ignore*: ⋀*s x*. ⟦ *I s*; ¬ *consider' x* ⟧ ⟹ *interaction-bounded-by consider* (*callee s x*) *0*
and *I*: *I s*
and *invariant*: ⋀*s x y s'*. ⟦ (*y*, *s'*) ∈ *results'-gpv* (*callee s x*); *I s* ⟧ ⟹ *I s'*
shows *interaction-bounded-by consider* (*inline callee gpv s*) (*p* ∗ *q*)
proof −
{ assume ∃(*Rep* :: *'s'* ⇒ *'s*) *Abs*. *type-definition Rep Abs* {*s*. *I s*}
then obtain *Rep* :: *'s'* ⇒ *'s* **and** *Abs* **where** *td*: *type-definition Rep Abs* {*s*. *I s*} **by** *blast*
then interpret *td*: *type-definition Rep Abs* {*s*. *I s*} **.**
define *cr* **where** *cr x y* ⟷ *x* = *Rep y* **for** *x y*
have [*transfer-rule*]: *bi-unique cr right-total cr*
using *td cr-def*[*abs-def*] **by**(*rule typedef-bi-unique typedef-right-total*)+
have [*transfer-domain-rule*]: *Domainp cr* = *I*
using *type-definition-Domainp*[*OF td cr-def*[*abs-def*]] **by** *simp*

define *callee'* **where** *callee'* = (*Rep* −−−> *id* −−−> *map-gpv* (*map-prod id Abs*) *id*) *callee*
have [*transfer-rule*]: (*cr* ===> (=) ===> *rel-gpv* (*rel-prod* (=) *cr*) (=)) *callee callee'*
by(*auto simp add*: *callee'-def rel-fun-def cr-def gpv.rel-map prod.rel-map td.Abs-inverse intro*!: *gpv.rel-refl-strong intro*: *td.Rep*[*simplified*] *dest*: *invariant*)

define *s'* **where** *s'* = *Abs s*
have [*transfer-rule*]: *cr s s'* **using** *I* **by**(*simp add*: *cr-def s'-def td.Abs-inverse*)

note *p* **moreover**

have *consider′ x* ⟹ *interaction-bounded-by consider* (*callee′ s x*) *q* **for** *s x*
by(*transfer fixing*: *consider consider′ q*)(*clarsimp simp add*: *q*)
moreover have ¬ *consider′ x* ⟹ *interaction-bounded-by consider* (*callee′ s x*) *0* **for** *s x*
by(*transfer fixing*: *consider consider′*)(*clarsimp simp add*: *ignore*)
ultimately have *interaction-bounded-by consider* (*inline callee′ gpv s′*) (*p* ∗ *q*)

by(*rule interaction-bounded-by-inline*)
then have *interaction-bounded-by consider* (*inline callee gpv s*) (*p* ∗ *q*) **by** *transfer* **}**
from *this*[*cancel-type-definition*] *I* **show** *?thesis* **by** *blast*
qed

context
fixes 𝓘 :: (*′call*, *′ret*) 𝓘
and 𝓘′ :: (*′call′*, *′ret′*) 𝓘
and *callee* :: *′s* ⇒ *′call* ⇒ (*′ret* × *′s*, *′call′*, *′ret′*) *gpv*
assumes *results*: ⋀*s x*. *x* ∈ *outs-𝓘* 𝓘 ⟹ *results-gpv* 𝓘′ (*callee s x*) ⊆ *responses-𝓘* 𝓘 *x* × *UNIV*
begin

lemma *inline1-in-sub-gpvs-callee*:
assumes *Inr* (*out*, *callee′*, *rpv′*) ∈ *set-spmf* (*inline1 callee gpv s*)
and *WT*: 𝓘 ⊢*g gpv* √
shows ∃ *call*∈*outs-𝓘* 𝓘. ∃ *s*. ∀ *x* ∈ *responses-𝓘* 𝓘′ *out*. *callee′ x* ∈ *sub-gpvs* 𝓘′ (*callee s call*)
proof −
from *WT*
have *set-spmf* (*inline1 callee gpv s*) ⊆ {*Inr* (*out*, *callee′*, *rpv′*) | *out callee′ rpv′*.
∃ *call*∈*outs-𝓘* 𝓘. ∃ *s*. ∀ *x* ∈ *responses-𝓘* 𝓘′ *out*. *callee′ x* ∈ *sub-gpvs* 𝓘′ (*callee s call*)} ∪ *range Inl*
(**is** *?concl* (*inline1 callee*) *gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *inline1-fixp-induct*)
case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*)
case *bottom* **show** *?case* **by** *simp*
case (*step inline1′*)
{ fix *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
from *step.prems IO* **have** *out*: *out* ∈ *outs-𝓘* 𝓘 **by**(*rule WT-gpvD*)
{ fix *x s′*
assume *Pure*: *Pure* (*x*, *s′*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
then have (*x*, *s′*) ∈ *results-gpv* 𝓘′ (*callee s out*) **by**(*rule results-gpv.Pure*)
with *out* **have** *x* ∈ *responses-𝓘* 𝓘 *out* **by**(*auto dest*: *results*)
with *step.prems IO* **have** 𝓘 ⊢*g c x* √ **by**(*rule WT-gpvD*)
hence *?concl inline1′* (*c x*) *s′* **by**(*rule step.IH*)
} moreover {
fix *out′ c′*
assume *IO out′ c′* ∈ *set-spmf* (*the-gpv* (*callee s out*))
hence ∀ *x*∈*responses-𝓘* 𝓘′ *out′*. *c′ x* ∈ *sub-gpvs* 𝓘′ (*callee s out*)

by(*auto intro*: *sub-gpvs.base*)
then have ∃ *call*∈*outs-ℐ ℐ*. ∃ *s*. ∀ *x*∈*responses-ℐ ℐ′ out′*. *c′ x* ∈ *sub-gpvs ℐ′* (*callee s call*)
using *out* **by** *blast*
} **moreover note** *calculation* **}**
then show *?case* **using** *step.prems*
by(*auto del*: *subsetI simp add*: *bind-UNION intro*!: *UN-least split*: *generat.split*)
qed
thus *?thesis* **using** *assms* **by** *fastforce*
qed

lemma *inline1-in-sub-gpvs*:
assumes *Inr* (*out*, *callee′*, *rpv′*) ∈ *set-spmf* (*inline1 callee gpv s*)
and (*x*, *s′*) ∈ *results-gpv ℐ′* (*callee′ input*)
and *input* ∈ *responses-ℐ ℐ′ out*
and *ℐ* ⊢*g gpv* √
shows *rpv′ x* ∈ *sub-gpvs ℐ gpv*
proof −
from ‹*ℐ* ⊢*g gpv* √›
have *set-spmf* (*inline1 callee gpv s*) ⊆ {*Inr* (*out*, *callee′*, *rpv′*) | *out callee′ rpv′*. ∀ *input* ∈ *responses-ℐ ℐ′ out*. ∀ (*x*, *s′*)∈*results-gpv ℐ′* (*callee′ input*). *rpv′ x* ∈ *sub-gpvs ℐ gpv*}
∪ *range Inl* (**is** *?concl* (*inline1 callee*) *gpv s* **is** - ⊆ *?rhs gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *inline1-fixp-induct*)
case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*)
case *bottom* **show** *?case* **by** *simp*
next
case (*step inline1′*)
{ **fix** *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
from *step.prems IO* **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpvD*)
{ **fix** *x s′*
assume *Pure*: *Pure* (*x*, *s′*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
then have (*x*, *s′*) ∈ *results-gpv ℐ′* (*callee s out*) **by**(*rule results-gpv.Pure*)
with *out* **have** *x* ∈ *responses-ℐ ℐ out* **by**(*auto dest*: *results*)
with *step.prems IO* **have** *ℐ* ⊢*g c x* √ **by**(*rule WT-gpvD*)
hence *?concl inline1′* (*c x*) *s′* **by**(*rule step.IH*)
also have … ⊆ *?rhs gpv s′* **using** *IO Pure*
by(*fastforce intro*: *sub-gpvs.cont dest*: *WT-gpv-OutD*[*OF step.prems*] *results*[*THEN subsetD*, *OF* - *results-gpv.Pure*])
finally have *set-spmf* (*inline1′* (*c x*) *s′*) ⊆ … **.**
} **moreover** **{**
fix *out′ c′ input x s′*
assume *IO out′ c′* ∈ *set-spmf* (*the-gpv* (*callee s out*))
and *input* ∈ *responses-ℐ ℐ′ out′* **and** (*x*, *s′*) ∈ *results-gpv ℐ′* (*c′ input*)
then have *c x* ∈ *sub-gpvs ℐ gpv* **using** *IO*
by(*auto intro*!: *sub-gpvs.base dest*: *WT-gpv-OutD*[*OF step.prems*] *results*[*THEN subsetD*, *OF* - *results-gpv.IO*])
} **moreover note** *calculation* **}**

then show *?case*
by(*auto simp add*: *bind-UNION intro*!: *UN-least split*: *generat.split del*: *subsetI*)
qed
with *assms* **show** *?thesis* **by** *fastforce*
qed

context
assumes *WT*: $\bigwedge x\ s.\ x \in$ *outs-*$\mathcal{I}\ \mathcal{I} \Longrightarrow \mathcal{I}' \vdash g$ *callee s x* $\surd$
begin

lemma *WT-gpv-inline1*:
assumes *Inr* (*out*, *rpv*, *rpv*′) ∈ *set-spmf* (*inline1 callee gpv s*)
and $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *out* ∈ *outs-*$\mathcal{I}\ \mathcal{I}'$ (**is** *?thesis1*)
and *input* ∈ *responses-*$\mathcal{I}\ \mathcal{I}'$ *out* $\Longrightarrow \mathcal{I}' \vdash g$ *rpv input* $\surd$ (**is** *PROP ?thesis2*)
and ⟦ *input* ∈ *responses-*$\mathcal{I}\ \mathcal{I}'$ *out*; (*x*, *s*′) ∈ *results-gpv* $\mathcal{I}'$ (*rpv input*) ⟧ $\Longrightarrow \mathcal{I} \vdash g$ *rpv*′ *x* $\surd$ (**is** *PROP ?thesis3*)
proof −
from ‹$\mathcal{I} \vdash g$ *gpv* $\surd$›
have *set-spmf* (*inline1 callee gpv s*) ⊆ {*Inr* (*out*, *rpv*, *rpv*′) | *out rpv rpv*′. *out* ∈ *outs-*$\mathcal{I}\ \mathcal{I}'$} ∪ *range Inl*
proof(*induction arbitrary*: *gpv s rule*: *inline1-fixp-induct*)
{ case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*) **}**
{ case *bottom* **show** *?case* **by** *simp* **}**
case (*step inline1*′)
{ fix *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
from *step.prems IO* **have** *out*: *out* ∈ *outs-*$\mathcal{I}\ \mathcal{I}$ **by**(*rule WT-gpvD*)
{ fix *x s*′
assume *Pure*: *Pure* (*x*, *s*′) ∈ *set-spmf* (*the-gpv* (*callee s out*))
then have (*x*, *s*′) ∈ *results-gpv* $\mathcal{I}'$ (*callee s out*) **by**(*rule results-gpv.Pure*)
with *out* **have** *x* ∈ *responses-*$\mathcal{I}\ \mathcal{I}$ *out* **by**(*auto dest*: *results*)
with *step.prems IO* **have** $\mathcal{I} \vdash g$ *c x* $\surd$ **by**(*rule WT-gpvD*)
} moreover {
fix *out*′ *c*′
from *out* **have** $\mathcal{I}' \vdash g$ *callee s out* $\surd$ **by**(*rule WT*)
moreover assume *IO out*′ *c*′ ∈ *set-spmf* (*the-gpv* (*callee s out*))
ultimately have *out*′ ∈ *outs-*$\mathcal{I}\ \mathcal{I}'$ **by**(*rule WT-gpvD*)
} moreover note *calculation* **}**
then show *?case*
by(*auto del*: *subsetI simp add*: *bind-UNION intro*!: *UN-least split*: *generat.split intro*!: *step.IH*[*THEN order-trans*])
qed
then show *?thesis1* **using** *assms* **by** *auto*

assume *input* ∈ *responses-*$\mathcal{I}\ \mathcal{I}'$ *out*
with *inline1-in-sub-gpvs-callee*[*OF* ‹*Inr* - ∈ -›] ‹$\mathcal{I} \vdash g$ *gpv* $\surd$›
obtain *out*′ *s* **where** *out*′ ∈ *outs-*$\mathcal{I}\ \mathcal{I}$
and ∗: *rpv input* ∈ *sub-gpvs* $\mathcal{I}'$ (*callee s out*′) **by** *auto*

from ‹*out*′ ∈ -› **have** $\mathcal{I}'$ ⊢*g callee s out*′ √ **by**(*rule WT*)
then show $\mathcal{I}'$ ⊢*g rpv input* √ **using** ∗ **by**(*rule WT-sub-gpvsD*)

assume (*x*, *s*′) ∈ *results-gpv* $\mathcal{I}'$ (*rpv input*)
with ‹*Inr* - ∈ -› **have** *rpv*′ *x* ∈ *sub-gpvs* $\mathcal{I}$ *gpv*
using ‹*input* ∈ -› ‹$\mathcal{I}$ ⊢*g gpv* √› **by**(*rule inline1-in-sub-gpvs*)
with ‹$\mathcal{I}$ ⊢*g gpv* √› **show** $\mathcal{I}$ ⊢*g rpv*′ *x* √ **by**(*rule WT-sub-gpvsD*)
qed

lemma *WT-gpv-inline*:
assumes $\mathcal{I}$ ⊢*g gpv* √
shows $\mathcal{I}'$ ⊢*g inline callee gpv s* √
using *assms*
proof(*coinduction arbitrary*: *gpv s rule*: *WT-gpv-coinduct-bind*)
case (*WT-gpv out c gpv*)
from ‹*IO out c* ∈ -› **obtain** *callee*′ *rpv*′
where *Inr*: *Inr* (*out*, *callee*′, *rpv*′) ∈ *set-spmf* (*inline1 callee gpv s*)
and *c*: *c* = (λ*input*. *callee*′ *input* ≫= (λ(*x*, *s*). *inline callee* (*rpv*′ *x*) *s*))
by(*clarsimp simp add*: *inline-sel split*: *sum.split-asm*)
from *Inr* ‹$\mathcal{I}$ ⊢*g gpv* √› **have** *?out* **by**(*rule WT-gpv-inline1*)
moreover have *?cont TYPE*(′*ret* × ′*s*) (**is** ∀ *input*∈-. - ∨ - ∨ *?case*′ *input*)
proof(*rule ballI disjI2*)+
fix *input*
assume *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out*
with *Inr* ‹$\mathcal{I}$ ⊢*g gpv* √ ›**have** $\mathcal{I}'$ ⊢*g callee*′ *input* √
and ⋀*x s*′. (*x*, *s*′) ∈ *results-gpv* $\mathcal{I}'$ (*callee*′ *input*) ⟹ $\mathcal{I}$ ⊢*g rpv*′ *x* √
by(*blast intro*: *WT-gpv-inline1*)+
then show *?case*′ *input* **by**(*subst c*)(*auto 4 4*)
qed
ultimately show *?case TYPE*(′*ret* × ′*s*) **..**
qed

end

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

lemma *lossless-spmf-inline1*:
assumes *lossless*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-spmf* (*the-gpv* (*callee s x*))
shows *lossless-spmf* (*inline1 callee gpv s*)
using *gpv*
proof(*induction arbitrary*: *s rule*: *lossless-WT-gpv-induct*)
case (*lossless-gpv p*)
show *?case* **using** ‹*lossless-spmf p*›
apply(*subst inline1-unfold*)
apply(*auto split*: *generat.split intro*: *lossless lossless-gpv.hyps dest*: *results*[*THEN subsetD*, *rotated*, *OF results-gpv.Pure*] *intro*: *lossless-gpv.IH*)

done
qed

lemma *lossless-gpv-inline1*:
assumes ∗: *Inr* (*out*, *rpv*, *rpv*′) ∈ *set-spmf* (*inline1 callee gpv s*)
and ∗∗: *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out*
and *lossless*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-gpv* $\mathcal{I}'$ (*callee s x*)
shows *lossless-gpv* $\mathcal{I}'$ (*rpv input*)
proof −
from *inline1-in-sub-gpvs-callee*[*OF* ∗ *gpv*(*2*)] ∗∗
obtain *out*′ *s* **where** *out*′ ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **and** ∗∗∗: *rpv input* ∈ *sub-gpvs* $\mathcal{I}'$ (*callee s out*′) **by** *blast*
from ‹*out*′ ∈ -› **have** *lossless-gpv* $\mathcal{I}'$ (*callee s out*′) **by**(*rule lossless*)
thus *?thesis* **using** ∗∗∗ **by**(*rule lossless-sub-gpvsD*)
qed

lemma *lossless-results-inline1*:
assumes *Inr* (*out*, *rpv*, *rpv*′) ∈ *set-spmf* (*inline1 callee gpv s*)
and (*x*, *s*′) ∈ *results-gpv* $\mathcal{I}'$ (*rpv input*)
and *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out*
shows *lossless-gpv* $\mathcal{I}$ (*rpv*′ *x*)
proof −
from *assms gpv*(*2*) **have** *rpv*′ *x* ∈ *sub-gpvs* $\mathcal{I}$ *gpv* **by**(*rule inline1-in-sub-gpvs*)
with *gpv*(*1*) **show** *lossless-gpv* $\mathcal{I}$ (*rpv*′ *x*) **by**(*rule lossless-sub-gpvsD*)
qed

end

lemmas *lossless-inline1*[*rotated 2*] = *lossless-spmf-inline1 lossless-gpv-inline1 lossless-results-inline1*

lemma *lossless-inline*[*rotated*]:
fixes *gpv* :: (′*a*, ′*call*, ′*ret*) *gpv*
assumes *gpv*: *lossless-gpv* $\mathcal{I}$ *gpv* $\mathcal{I}$ ⊢*g gpv* √
and *lossless*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-gpv* $\mathcal{I}'$ (*callee s x*)
shows *lossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
using *gpv*
proof(*induction arbitrary*: *s rule*: *lossless-WT-gpv-induct-strong*)
case (*lossless-gpv p*)
have *lp*: *lossless-gpv* $\mathcal{I}$ (*GPV p*) **by**(*rule lossless-sub-gpvsI*)(*auto intro*: *lossless-gpv.hyps*)
moreover have *wp*: $\mathcal{I}$ ⊢*g GPV p* √ **by**(*rule WT-sub-gpvsI*)(*auto intro*: *lossless-gpv.hyps*)
ultimately have *lossless-spmf* (*the-gpv* (*inline callee* (*GPV p*) *s*))
by(*auto simp add*: *inline-sel intro*: *lossless-spmf-inline1 lossless-gpv-lossless-spmfD*[*OF lossless*])
moreover {
fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv* (*inline callee* (*GPV p*) *s*))

and *input* ∈ *responses-ℐ ℐ′ out*
from *IO* **obtain** *callee′ rpv*
where *Inr*: *Inr* (*out*, *callee′*, *rpv*) ∈ *set-spmf* (*inline1 callee* (*GPV p*) *s*)
and *c*: *c* = (λ*input*. *callee′ input* ≫= (λ(*x*, *y*). *inline callee* (*rpv x*) *y*))
by(*clarsimp simp add*: *inline-sel split*: *sum.split-asm*)
from *Inr* ‹*input* ∈ -› *lossless lp wp* **have** *lossless-gpv ℐ′* (*callee′ input*) **by**(*rule lossless-inline1*)
moreover **{**
fix *x s′*
assume (*x*, *s′*) ∈ *results-gpv ℐ′* (*callee′ input*)
with *Inr* **have** *rpv x* ∈ *sub-gpvs ℐ* (*GPV p*) **using** ‹*input* ∈ -› *wp* **by**(*rule inline1-in-sub-gpvs*)
hence *lossless-gpv ℐ′* (*inline callee* (*rpv x*) *s′*) **by**(*rule lossless-gpv.IH*)
} **ultimately have** *lossless-gpv ℐ′* (*c input*) **unfolding** *c* **by** *clarsimp*
} **ultimately show** *?case* **by**(*rule lossless-gpvI*)
qed

end

definition *id-oracle* :: ′*s* ⇒ ′*call* ⇒ (′*ret* × ′*s*, ′*call*, ′*ret*) *gpv*
where *id-oracle s x* = *Pause x* (λ*x*. *Done* (*x*, *s*))

lemma *inline1-id-oracle*:
inline1 id-oracle gpv s =
map-spmf (λ*generat*. *case generat of Pure x* ⇒ *Inl* (*x*, *s*) | *IO out c* ⇒ *Inr* (*out*, λ*x*. *Done* (*x*, *s*), *c*)) (*the-gpv gpv*)
by(*subst inline1.simps*)(*auto simp add*: *id-oracle-def map-spmf-conv-bind-spmf intro*!: *bind-spmf-cong split*: *generat.split*)

lemma *inline-id-oracle* [*simp*]: *inline id-oracle gpv s* = *map-gpv* (λ*x*. (*x*, *s*)) *id gpv*
by(*coinduction arbitrary*: *gpv s*)(*auto 4 3 simp add*: *inline-sel inline1-id-oracle spmf-rel-map gpv.map-sel o-def generat.rel-map intro*!: *rel-spmf-reflI rel-funI split*: *generat.split*)

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

context begin
private lemma *aux*:
set-spmf (*inline1 callee gpv s*) ⊆ {*Inr* (*out*, *callee′*, *rpv′*) | *out callee′ rpv′*.
∃ *call*∈*outs-ℐ ℐ*. ∃ *s*. *I s* ∧ (∀ *x* ∈ *responses-ℐ ℐ′ out*. *callee′ x* ∈ *sub-gpvs ℐ′* (*callee s call*))} ∪

{*Inl* (*x*, *s*′) | *x s*′. *x* ∈ *results-gpv* ℐ *gpv* ∧ *I s*′}
(**is** *?concl* (*inline1 callee*) *gpv s* **is** - ⊆ *?rhs1* ∪ *?rhs2 gpv*)
if ℐ ⊢*g gpv* √ *I s*
using *that*
proof(*induction arbitrary*: *gpv s rule*: *inline1-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1*′)
{ fix *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
from *step.prems*(*1*) *IO* **have** *out*: *out* ∈ *outs-*ℐ ℐ **by**(*rule WT-gpvD*)
{ fix *x s*′
assume *Pure*: *Pure* (*x*, *s*′) ∈ *set-spmf* (*the-gpv* (*callee s out*))
then have (*x*, *s*′) ∈ *results-gpv* ℐ′ (*callee s out*) **by**(*rule results-gpv.Pure*)
with *out step.prems*(*2*) **have** *x* ∈ *responses-*ℐ ℐ *out I s*′ **by**(*auto dest*: *results-callee*)
from *step.prems*(*1*) *IO this*(*1*) **have** ℐ ⊢*g c x* √ **by**(*rule WT-gpvD*)
hence *?concl inline1*′ (*c x*) *s*′ **using** ‹*I s*′› **by**(*rule step.IH*)
also have ... ⊆ *?rhs1* ∪ *?rhs2 gpv* **using** ‹*x* ∈ -› *IO* **by**(*auto intro*: *results-gpv.intros*)
also note *calculation*
} moreover {
fix *out*′ *c*′
assume *IO out*′ *c*′ ∈ *set-spmf* (*the-gpv* (*callee s out*))
hence ∀ *x*∈*responses-*ℐ ℐ′ *out*′. *c*′ *x* ∈ *sub-gpvs* ℐ′ (*callee s out*)
by(*auto intro*: *sub-gpvs.base*)
then have ∃ *call*∈*outs-*ℐ ℐ. ∃ *s*. *I s* ∧ (∀ *x*∈*responses-*ℐ ℐ′ *out*′. *c*′ *x* ∈ *sub-gpvs* ℐ′ (*callee s call*))
using *out step.prems*(*2*) **by** *blast*
} moreover note *calculation* **}**
then show *?case* **using** *step.prems*
by(*auto 4 3 del*: *subsetI simp add*: *bind-UNION intro*!: *UN-least split*: *generat.split intro*: *results-gpv.intros*)
qed

lemma *inline1-in-sub-gpvs-callee*:
assumes *Inr* (*out*, *callee*′, *rpv*′) ∈ *set-spmf* (*inline1 callee gpv s*)
and *WT*: ℐ ⊢*g gpv* √
and *s*: *I s*
shows ∃ *call*∈*outs-*ℐ ℐ. ∃ *s*. *I s* ∧ (∀ *x* ∈ *responses-*ℐ ℐ′ *out*. *callee*′ *x* ∈ *sub-gpvs* ℐ′ (*callee s call*))
using *aux*[*OF WT s*] *assms*(*1*) **by** *fastforce*

lemma *inline1-Inl-results-gpv*:
assumes *Inl* (*x*, *s*′) ∈ *set-spmf* (*inline1 callee gpv s*)
and *WT*: ℐ ⊢*g gpv* √
and *s*: *I s*
shows *x* ∈ *results-gpv* ℐ *gpv* ∧ *I s*′
using *aux*[*OF WT s*] *assms*(*1*) **by** *fastforce*

end

lemma *inline1-in-sub-gpvs*:
  assumes *Inr* (*out*, *callee'*, *rpv'*) ∈ *set-spmf* (*inline1 callee gpv s*)
    and (*x*, *s'*) ∈ *results-gpv* ℐ' (*callee' input*)
    and *input* ∈ *responses-ℐ* ℐ' *out*
    and ℐ ⊢*g gpv* √
    and *I s*
  shows *rpv' x* ∈ *sub-gpvs* ℐ *gpv* ∧ *I s'*
proof −
  from ‹ℐ ⊢*g gpv* √› ‹*I s*›
  have *set-spmf* (*inline1 callee gpv s*) ⊆ {*Inr* (*out*, *callee'*, *rpv'*) | *out callee' rpv'*. ∀ *input* ∈ *responses-ℐ* ℐ' *out*. ∀ (*x*, *s'*)∈*results-gpv* ℐ' (*callee' input*). *I s'* ∧ *rpv' x* ∈ *sub-gpvs* ℐ *gpv*}
    ∪ {*Inl* (*x*, *s'*) | *x s'*. *I s'*} (**is** *?concl* (*inline1 callee*) *gpv s* **is** - ⊆ *?rhs gpv s*)
  proof(*induction arbitrary*: *gpv s rule*: *inline1-fixp-induct*)
    case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*)
    case *bottom* **show** *?case* **by** *simp*
    case (*step inline1'*)
    { fix *out c*
      assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
      from *step.prems*(*1*) *IO* **have** *out*: *out* ∈ *outs-ℐ* ℐ **by**(*rule WT-gpvD*)
      { fix *x s'*
        assume *Pure*: *Pure* (*x*, *s'*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
        then have (*x*, *s'*) ∈ *results-gpv* ℐ' (*callee s out*) **by**(*rule results-gpv.Pure*)
          with *out step.prems*(*2*) **have** *x* ∈ *responses-ℐ* ℐ *out I s'* **by**(*auto dest*: *results-callee*)
        from *step.prems*(*1*) *IO this*(*1*) **have** ℐ ⊢*g c x* √ **by**(*rule WT-gpvD*)
        hence *?concl inline1'* (*c x*) *s'* **using** ‹*I s'*› **by**(*rule step.IH*)
        also have ... ⊆ *?rhs gpv s'* **using** *IO Pure* ‹*I s*›
          by(*fastforce intro*: *sub-gpvs.cont dest*: *WT-gpv-OutD*[*OF step.prems*(*1*)] *results-callee*[*THEN subsetD*, *OF* - - *results-gpv.Pure*])
        finally have *set-spmf* (*inline1'* (*c x*) *s'*) ⊆ ... **.**
      } moreover {
        fix *out' c' input x s'*
        assume *IO out' c'* ∈ *set-spmf* (*the-gpv* (*callee s out*))
          and *input* ∈ *responses-ℐ* ℐ' *out'* **and** (*x*, *s'*) ∈ *results-gpv* ℐ' (*c' input*)
        then have *c x* ∈ *sub-gpvs* ℐ *gpv I s'* **using** *IO* ‹*I s*›
          by(*auto intro*!: *sub-gpvs.base dest*: *WT-gpv-OutD*[*OF step.prems*(*1*)] *results-callee*[*THEN subsetD*, *OF* - - *results-gpv.IO*])
      } moreover note *calculation* **}**
      then show *?case* **using** *step.prems*(*2*)
        by(*auto simp add*: *bind-UNION intro*!: *UN-least split*: *generat.split del*: *subsetI*)
    qed
    with *assms* **show** *?thesis* **by** *fastforce*
  qed

lemma *WT-gpv-inline1*:

```
  assumes Inr (out, rpv, rpv') ∈ set-spmf (inline1 callee gpv s)
    and ℐ ⊢g gpv √
    and I s
  shows out ∈ outs-ℐ ℐ' (is ?thesis1)
    and input ∈ responses-ℐ ℐ' out ⟹ ℐ' ⊢g rpv input √ (is PROP ?thesis2)
    and ⟦ input ∈ responses-ℐ ℐ' out; (x, s') ∈ results-gpv ℐ' (rpv input) ⟧ ⟹ ℐ
⊢g rpv' x √ ∧ I s' (is PROP ?thesis3)
proof −
  from ‹ℐ ⊢g gpv √› ‹I s›
  have set-spmf (inline1 callee gpv s) ⊆ {Inr (out, rpv, rpv') | out rpv rpv'. out ∈
outs-ℐ ℐ'} ∪ {Inl (x, s')| x s'. I s'}
  proof(induction arbitrary: gpv s rule: inline1-fixp-induct)
    { case adm show ?case by(intro cont-intro ccpo-class.admissible-leI) }
    { case bottom show ?case by simp }
    case (step inline1')
    { fix out c
      assume IO: IO out c ∈ set-spmf (the-gpv gpv)
      from step.prems(1) IO have out: out ∈ outs-ℐ ℐ by(rule WT-gpvD)
      { fix x s'
        assume Pure: Pure (x, s') ∈ set-spmf (the-gpv (callee s out))
       then have ∗: (x, s') ∈ results-gpv ℐ' (callee s out) by(rule results-gpv.Pure)
          with out step.prems(2) have x ∈ responses-ℐ ℐ out I s' by(auto dest:
results-callee)
        from step.prems(1) IO this(1) have ℐ ⊢g c x √ by(rule WT-gpvD)
        note this ‹I s'›
      } moreover {
        fix out' c'
        from out step.prems(2) have ℐ' ⊢g callee s out √ by(rule WT-callee)
        moreover assume IO out' c' ∈ set-spmf (the-gpv (callee s out))
        ultimately have out' ∈ outs-ℐ ℐ' by(rule WT-gpvD)
      } moreover note calculation }
      then show ?case using step.prems(2)
      by(auto del: subsetI simp add: bind-UNION intro!: UN-least split: generat.split
intro!: step.IH[THEN order-trans])
    qed
    then show ?thesis1 using assms by auto

    assume input ∈ responses-ℐ ℐ' out
    with inline1-in-sub-gpvs-callee[OF ‹Inr - ∈ -› ‹ℐ ⊢g gpv √› ‹I s›]
    obtain out' s where out' ∈ outs-ℐ ℐ
      and ∗: rpv input ∈ sub-gpvs ℐ' (callee s out') and I s by blast
    from ‹out' ∈ -› ‹I s› have ℐ' ⊢g callee s out' √ by(rule WT-callee)
    then show ℐ' ⊢g rpv input √ using ∗ by(rule WT-sub-gpvsD)

    assume (x, s') ∈ results-gpv ℐ' (rpv input)
    with ‹Inr - ∈ -› have rpv' x ∈ sub-gpvs ℐ gpv ∧ I s'
      using ‹input ∈ -› ‹ℐ ⊢g gpv √› assms(3) ‹I s› by−(rule inline1-in-sub-gpvs)
    with ‹ℐ ⊢g gpv √› show ℐ ⊢g rpv' x √ ∧ I s' by(blast intro: WT-sub-gpvsD)
  qed
```

lemma *WT-gpv-inline-invar*:
  assumes $\mathcal{I} \vdash g\ gpv\ \surd$
    and *I s*
  shows $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$
  using *assms*
proof(*coinduction arbitrary*: *gpv s rule*: *WT-gpv-coinduct-bind*)
  case (*WT-gpv out c gpv*)
  from ‹*IO out c* ∈ -› **obtain** *callee′ rpv′*
    where *Inr*: *Inr* (*out*, *callee′*, *rpv′*) ∈ *set-spmf* (*inline1 callee gpv s*)
      and *c*: *c* = (λ*input*. *callee′ input* ≫= (λ(*x*, *s*). *inline callee* (*rpv′ x*) *s*))
    by(*clarsimp simp add*: *inline-sel split*: *sum.split-asm*)
  from *Inr* ‹$\mathcal{I} \vdash g\ gpv\ \surd$› ‹*I s*› **have** *?out* **by**(*rule WT-gpv-inline1*)
  moreover have *?cont TYPE*(*′ret* × *′s*) (**is** ∀ *input*∈-. - ∨ - ∨ *?case′ input*)
  proof(*rule ballI disjI2*)+
    fix *input*
    assume *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out*
    with *Inr* ‹$\mathcal{I} \vdash g\ gpv\ \surd$ › ‹*I s*› **have** $\mathcal{I}' \vdash g$ *callee′ input* $\surd$
      and ⋀*x s′*. (*x*, *s′*) ∈ *results-gpv* $\mathcal{I}'$ (*callee′ input*) ⟹ $\mathcal{I} \vdash g$ *rpv′ x* $\surd$ ∧ *I s′*
      by(*blast dest*: *WT-gpv-inline1*)+
    then show *?case′ input* **by**(*subst c*)(*auto 4 5*)
  qed
  ultimately show *?case TYPE*(*′ret* × *′s*) **..**
qed

end

lemma *WT-gpv-inline′*:
  assumes ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *results-gpv* $\mathcal{I}'$ (*callee s x*) ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* × *UNIV*
    and ⋀*x s*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}' \vdash g$ *callee s x* $\surd$
    and $\mathcal{I} \vdash g\ gpv\ \surd$
  shows $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$
proof −
  interpret *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ *callee* λ-. *True*
    using *assms* **by**(*unfold-locales*)*auto*
  show *?thesis* **by**(*rule WT-gpv-inline-invar*)(*use assms* **in** *auto*)
qed

lemma *results-gpv-sub-gvps*: *gpv′* ∈ *sub-gpvs* $\mathcal{I}$ *gpv* ⟹ *results-gpv* $\mathcal{I}$ *gpv′* ⊆ *results-gpv* $\mathcal{I}$ *gpv*
  by(*induction rule*: *sub-gpvs.induct*)(*auto intro*: *results-gpv.IO*)

lemma *in-results-gpv-sub-gvps*: ⟦ *x* ∈ *results-gpv* $\mathcal{I}$ *gpv′*; *gpv′* ∈ *sub-gpvs* $\mathcal{I}$ *gpv* ⟧ ⟹ *x* ∈ *results-gpv* $\mathcal{I}$ *gpv*
  using *results-gpv-sub-gvps*[*of gpv′* $\mathcal{I}$ *gpv*] **by** *blast*

context *raw-converter-invariant* **begin**
lemma *results-gpv-inline-aux*:

assumes (*x*, *s'*) ∈ *results-gpv* ℐ' (*inline-aux callee y*)
shows ⟦ *y* = *Inl* (*gpv*, *s*); ℐ ⊢*g gpv* √; *I s* ⟧ ⟹ *x* ∈ *results-gpv* ℐ *gpv* ∧ *I s'*
and ⟦ *y* = *Inr* (*rpv*, *callee'*); ∀ (*z*, *s'*) ∈ *results-gpv* ℐ' *callee'*. ℐ ⊢*g rpv z* √ ∧ *I s'* ⟧
⟹ ∃ (*z*, *s''*) ∈ *results-gpv* ℐ' *callee'*. *x* ∈ *results-gpv* ℐ (*rpv z*) ∧ *I s''* ∧ *I s'*
using *assms*
proof(*induction gvp'*≡*inline-aux callee y arbitrary*: *y gpv s rpv callee'*)
case *Pure* **case** *1*
with *Pure* **show** *?case*
by(*auto simp add*: *inline-aux.sel split*: *sum.split-asm dest*: *inline1-Inl-results-gpv*)
next
case *Pure* **case** *2*
with *Pure* **show** *?case*
by(*clarsimp simp add*: *inline-aux.sel split*: *sum.split-asm*)
(*fastforce split*: *generat.split-asm dest*: *inline1-Inl-results-gpv intro*: *results-gpv.Pure*)+
next
case (*IO out c input*) **case** *1*
with *IO*(*1*) **obtain** *rpv rpv'* **where** *inline1*: *Inr* (*out*, *rpv*, *rpv'*) ∈ *set-spmf* (*inline1 callee gpv s*)
and *c*: *c* = (λ*input*. *inline-aux callee* (*Inr* (*rpv'*, *rpv input*)))
by(*auto simp add*: *inline-aux.sel split*: *sum.split-asm*)
from *inline1*[*THEN inline1-in-sub-gpvs*, *OF* - ‹*input* ∈ *responses-*ℐ ℐ' *out*› - ‹*I s*›] ‹ℐ ⊢*g gpv* √›
have ∀ (*z*, *s'*)∈*results-gpv* ℐ' (*rpv input*). ℐ ⊢*g rpv' z* √ ∧ *I s'*
by(*auto intro*: *WT-sub-gpvsD*)
from *IO*(*5*)[*unfolded c*, *OF refl refl this*] **obtain** *input' s''*
where *input'*: (*input'*, *s''*) ∈ *results-gpv* ℐ' (*rpv input*)
and *x*: *x* ∈ *results-gpv* ℐ (*rpv' input'*) **and** *s''*: *I s''* *I s'*
by *auto*
from *inline1*[*THEN inline1-in-sub-gpvs*, *OF input'* ‹*input* ∈ *responses-*ℐ ℐ' *out*› ‹ℐ ⊢*g gpv* √› ‹*I s*›] *s'' x*
show *?case* **by**(*auto intro*: *in-results-gpv-sub-gvps*)
next
case (*IO out c input*) **case** *2*
from *IO*(*1*) *2*(*1*) **consider** (*Pure*) *input' s'' rpv' rpv''*
where *Pure* (*input'*, *s''*) ∈ *set-spmf* (*the-gpv callee'*) *Inr* (*out*, *rpv'*, *rpv''*) ∈ *set-spmf* (*inline1 callee* (*rpv input'*) *s''*)
c = (λ*input*. *inline-aux callee* (*Inr* (*rpv''*, *rpv' input*)))
| (*Cont*) *rpv'* **where** *IO out rpv'* ∈ *set-spmf* (*the-gpv callee'*) *c* = (λ*input*. *inline-aux callee* (*Inr* (*rpv*, *rpv' input*)))
by(*auto simp add*: *inline-aux.sel split*: *sum.split-asm*; *rename-tac generat*; *case-tac generat*; *clarsimp*)
then show *?case*
proof *cases*
case *Pure*
have *res*: (*input'*, *s''*) ∈ *results-gpv* ℐ' *callee'* **using** *Pure*(*1*) **by**(*rule results-gpv.Pure*)
with *2* **have** *WT*: ℐ ⊢*g rpv input'* √ *I s''* **by** *auto*
have ∀ (*z*, *s'*)∈*results-gpv* ℐ' (*rpv' input*). ℐ ⊢*g rpv'' z* √ ∧ *I s'*

using *inline1-in-sub-gpvs*[*OF Pure*(*2*) - ‹*input* ∈ -› *WT*] *WT* **by**(*auto intro*: *WT-sub-gpvsD*)
from *IO*(*5*)[*unfolded Pure*(*3*), *OF refl refl this*] **obtain** *z s′′′*
where *z*: (*z*, *s′′′*) ∈ *results-gpv* $\mathcal{I}'$ (*rpv′ input*)
and *x*: *x* ∈ *results-gpv* $\mathcal{I}$ (*rpv′′ z*) **and** *s′*: *I s′′′ I s′* **by** *auto*
have *x* ∈ *results-gpv* $\mathcal{I}$ (*rpv input′*) **using** *x inline1-in-sub-gpvs*[*OF Pure*(*2*) *z* ‹*input* ∈ -› *WT*]
by(*auto intro*: *in-results-gpv-sub-gvps*)
then show *?thesis* **using** *res WT s′* **by** *auto*
next
case *Cont*
have ∀ (*z*, *s′*)∈*results-gpv* $\mathcal{I}'$ (*rpv′ input*). $\mathcal{I}$ ⊢*g rpv z* √ ∧ *I s′*
using *Cont 2* ‹*input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out*› **by**(*auto intro*: *results-gpv.IO*)
from *IO*(*5*)[*unfolded Cont*, *OF refl refl this*] **obtain** *z s′′*
where (*z*, *s′′*) ∈ *results-gpv* $\mathcal{I}'$ (*rpv′ input*) *x* ∈ *results-gpv* $\mathcal{I}$ (*rpv z*) *I s′′ I s′* **by** *auto*
then show *?thesis* **using** *Cont*(*1*) ‹*input* ∈ -› **by**(*auto intro*: *results-gpv.IO*)
qed
qed

lemma *results-gpv-inline*:
⟦(*x*, *s′*) ∈ *results-gpv* $\mathcal{I}'$ (*inline callee gpv s*); $\mathcal{I}$ ⊢*g gpv* √; *I s*⟧ ⟹ *x* ∈ *results-gpv* $\mathcal{I}$ *gpv* ∧ *I s′*
unfolding *inline-def* **by**(*rule results-gpv-inline-aux*(*1*)[*OF - refl*])

end

lemma *inline-map-gpv*:
inline callee (*map-gpv f g gpv*) *s* = *map-gpv* (*apfst f*) *id* (*inline* (λ*s x*. *callee s* (*g x*)) *gpv s*)
unfolding *apfst-def*
by(*rule inline-parametric*
[**where** *S*=*BNF-Def.Grp UNIV id* **and** *C*=*BNF-Def.Grp UNIV g* **and** *C′*=*BNF-Def.Grp UNIV id* **and** *A*=*BNF-Def.Grp UNIV f*,
THEN rel-funD, *THEN rel-funD*, *THEN rel-funD*,
unfolded gpv.rel-Grp prod.rel-Grp, *simplified*, *folded eq-alt*, *unfolded Grp-def*, *simplified*])
(*auto simp add*: *rel-fun-def relator-eq*)

4.17 Running GPVs

type-synonym (*′call*, *′ret*, *′s*) *callee* = *′s* ⇒ *′call* ⇒ (*′ret* × *′s*) *spmf*

context fixes *callee* :: (*′call*, *′ret*, *′s*) *callee* **notes** [[*function-internals*]] **begin**

partial-function (*spmf*) *exec-gpv* :: (*′a*, *′call*, *′ret*) *gpv* ⇒ *′s* ⇒ (*′a* × *′s*) *spmf*
where
exec-gpv c s =
the-gpv c ≫=

case-generat (*λx. return-spmf* (*x*, *s*))
(*λout c. callee s out* ≫= (*λ*(*x*, *y*). *exec-gpv* (*c x*) *y*))

abbreviation *run-gpv* :: (*'a*, *'call*, *'ret*) *gpv* ⇒ *'s* ⇒ *'a spmf*
where *run-gpv gpv s* ≡ *map-spmf fst* (*exec-gpv gpv s*)

lemma *exec-gpv-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (*λf*. *P* (*λc s*. *f* (*c*, *s*)))
and *P* (*λ- -. return-pmf None*)
and ⋀*exec-gpv*. *P exec-gpv* ⟹
P (*λc s. the-gpv c* ≫= *case-generat* (*λx. return-spmf* (*x*, *s*)) (*λout c. callee s out* ≫= (*λ*(*x*, *y*). *exec-gpv* (*c x*) *y*)))
shows *P exec-gpv*
using *assms*(*1*)
by(*rule exec-gpv.fixp-induct*[*unfolded curry-conv*[*abs-def*]])(*simp-all add*: *assms*(*2*−))

lemma *exec-gpv-fixp-induct-strong* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (*λf*. *P* (*λc s*. *f* (*c*, *s*)))
and *P* (*λ- -. return-pmf None*)
and ⋀*exec-gpv'*. ⟦ ⋀*c s. ord-spmf* (=) (*exec-gpv' c s*) (*exec-gpv c s*); *P exec-gpv'* ⟧
⟹ *P* (*λc s. the-gpv c* ≫= *case-generat* (*λx. return-spmf* (*x*, *s*)) (*λout c. callee s out* ≫= (*λ*(*x*, *y*). *exec-gpv'* (*c x*) *y*)))
shows *P exec-gpv*
using *assms*
by(*rule spmf.fixp-strong-induct-uc*[**where** *P*=*λf*. *P* (*curry f*) **and** *U*=*case-prod* **and** *C*=*curry*, *OF exec-gpv.mono exec-gpv-def*, *simplified curry-case-prod*, *simplified curry-conv*[*abs-def*] *fun-ord-def split-paired-All prod.case case-prod-eta*, *OF refl*]) *blast*

lemma *exec-gpv-fixp-induct-strong2* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (*λf*. *P* (*λc s*. *f* (*c*, *s*)))
and *P* (*λ- -. return-pmf None*)
and ⋀*exec-gpv'*.
⟦ ⋀*c s. ord-spmf* (=) (*exec-gpv' c s*) (*exec-gpv c s*);
⋀*c s. ord-spmf* (=) (*exec-gpv' c s*) (*the-gpv c* ≫= *case-generat* (*λx. return-spmf* (*x*, *s*)) (*λout c. callee s out* ≫= (*λ*(*x*, *y*). *exec-gpv'* (*c x*) *y*)));
P exec-gpv' ⟧
⟹ *P* (*λc s. the-gpv c* ≫= *case-generat* (*λx. return-spmf* (*x*, *s*)) (*λout c. callee s out* ≫= (*λ*(*x*, *y*). *exec-gpv'* (*c x*) *y*)))
shows *P exec-gpv*
using *assms*
by(*rule spmf.fixp-induct-strong2-uc*[**where** *P*=*λf*. *P* (*curry f*) **and** *U*=*case-prod* **and** *C*=*curry*, *OF exec-gpv.mono exec-gpv-def*, *simplified curry-case-prod*, *simplified curry-conv*[*abs-def*] *fun-ord-def split-paired-All prod.case case-prod-eta*, *OF refl*]) *blast*+

end

lemma *exec-gpv-conv-inline1*:
exec-gpv callee gpv s = map-spmf projl (inline1 (λs c. lift-spmf (callee s c) :: (-, unit, unit) gpv) gpv s)
by(*induction arbitrary*: *gpv s rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf exec-gpv.mono inline1.mono exec-gpv-def inline1-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
(*auto simp add*: *map-spmf-bind-spmf o-def spmf.map-comp bind-map-spmf split-def intro*!: *bind-spmf-cong*[*OF refl*] *split*: *generat.split*)

lemma *exec-gpv-simps*:
exec-gpv callee gpv s =
the-gpv gpv ≫=
case-generat (λx. return-spmf (x, s))
(λout rpv. callee s out ≫= *(λ(x, y). exec-gpv callee (rpv x) y))*
by(*fact exec-gpv.simps*)

lemma *exec-gpv-lift-spmf* [*simp*]:
exec-gpv callee (lift-spmf p) s = bind-spmf p (λx. return-spmf (x, s))
by(*simp add*: *exec-gpv-conv-inline1 spmf.map-comp o-def map-spmf-conv-bind-spmf*)

lemma *exec-gpv-Done* [*simp*]: *exec-gpv callee (Done x) s = return-spmf (x, s)*
by(*simp add*: *exec-gpv-conv-inline1*)

lemma *exec-gpv-Fail* [*simp*]: *exec-gpv callee Fail s = return-pmf None*
by(*simp add*: *exec-gpv-conv-inline1*)

lemma *if-distrib-exec-gpv* [*if-distribs*]:
exec-gpv callee (if b then x else y) s = (if b then exec-gpv callee x s else exec-gpv callee y s)
by *simp*

lemmas *exec-gpv-fixp-parallel-induct* [*case-names adm bottom step*] =
parallel-fixp-induct-2-2[*OF partial-function-definitions-spmf partial-function-definitions-spmf exec-gpv.mono exec-gpv.mono exec-gpv-def exec-gpv-def*, *unfolded lub-spmf-empty*]

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

lemma *exec-gpv-parametric′*:
((S ===> CALL ===> rel-spmf (rel-prod R S)) ===> rel-gpv′′ A CALL R ===> S ===> rel-spmf (rel-prod A S))
exec-gpv exec-gpv
apply(*rule rel-funI*)+
apply(*unfold spmf-rel-map exec-gpv-conv-inline1*)
apply(*rule rel-spmf-mono-strong*)
apply(*erule inline1-parametric′*[*THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*, *rotated*])

prefer *3*
apply(*drule in-set-inline1-lift-spmf1*)+
apply *fastforce*
subgoal by *simp*
subgoal premises [*transfer-rule*]
supply *lift-spmf-parametric'*[*transfer-rule*] **by** *transfer-prover*
done

lemma *exec-gpv-parametric* [*transfer-rule*]:
((*S* ===> *CALL* ===> *rel-spmf* (*rel-prod* ((=) :: *'ret* ⇒ -) *S*)) ===> *rel-gpv* *A CALL* ===> *S* ===> *rel-spmf* (*rel-prod A S*))
exec-gpv exec-gpv
unfolding *rel-gpv-conv-rel-gpv''* **by**(*rule exec-gpv-parametric'*)

end

lemma *exec-gpv-bind*: *exec-gpv callee* (*c* ≫= *f*) *s* = *exec-gpv callee c s* ≫= (λ(*x*, *s'*) ⇒ *exec-gpv callee* (*f x*) *s'*)
by(*auto simp add*: *exec-gpv-conv-inline1 inline1-bind-gpv map-spmf-bind-spmf o-def bind-map-spmf intro*!: *bind-spmf-cong*[*OF refl*] *dest*: *in-set-inline1-lift-spmf1*)

lemma *exec-gpv-map-gpv-id*:
exec-gpv oracle (*map-gpv f id gpv*) σ = *map-spmf* (*apfst f*) (*exec-gpv oracle gpv* σ)
proof(*rule sym*)
define *gpv'* **where** *gpv'* = *map-gpv f id gpv*
have [*transfer-rule*]: *rel-gpv* (λ*x y*. *y* = *f x*) (=) *gpv gpv'*
unfolding *gpv'-def* **by**(*simp add*: *gpv.rel-map gpv.rel-refl*)
have *rel-spmf* (*rel-prod* (λ*x y*. *y* = *f x*) (=)) (*exec-gpv oracle gpv* σ) (*exec-gpv oracle gpv'* σ)
by *transfer-prover*
thus *map-spmf* (*apfst f*) (*exec-gpv oracle gpv* σ) = *exec-gpv oracle* (*map-gpv f id gpv*) σ
unfolding *spmf-rel-eq*[*symmetric*] *gpv'-def spmf-rel-map* **by**(*rule rel-spmf-mono*) *clarsimp*
qed

lemma *exec-gpv-Pause* [*simp*]:
exec-gpv callee (*Pause out f*) *s* = *callee s out* ≫= (λ(*x*, *s'*). *exec-gpv callee* (*f x*) *s'*)
by(*simp add*: *inline1-Pause map-spmf-bind-spmf bind-map-spmf o-def exec-gpv-conv-inline1 split-def*)

lemma *exec-gpv-bind-lift-spmf*:
exec-gpv callee (*bind-gpv* (*lift-spmf p*) *f*) *s* = *bind-spmf p* (λ*x*. *exec-gpv callee* (*f x*) *s*)
by(*simp add*: *exec-gpv-bind*)

lemma *exec-gpv-bind-option* [*simp*]:

exec-gpv oracle (monad.bind-option Fail x f) s = monad.bind-option (return-pmf None) x (λa. exec-gpv oracle (f a) s)
by(*cases x*) *simp-all*

lemma *pred-spmf-exec-gpv*:
— We don't get an equivalence here because states are threaded through in *exec-gpv*.
⟦ *pred-gpv A C gpv*; *pred-fun S (pred-fun C (pred-spmf (pred-prod (λ-. True) S))) callee*; *S s* ⟧
⟹ *pred-spmf (pred-prod A S) (exec-gpv callee gpv s)*
using *exec-gpv-parametric*[*of eq-onp S eq-onp C eq-onp A*, *folded eq-onp-True*]
apply(*unfold prod.rel-eq-onp option.rel-eq-onp pmf.rel-eq-onp gpv.rel-eq-onp*)
apply(*drule rel-funD*[**where** *x=callee* **and** *y=callee*])
subgoal
apply(*rule rel-fun-mono*[**where** *X=eq-onp S*])
apply(*rule rel-fun-eq-onpI*)
apply(*unfold eq-onp-same-args*)
apply *assumption*
apply *simp*
apply(*erule rel-fun-eq-onpI*)
done
apply(*auto dest*!: *rel-funD simp add*: *eq-onp-def*)
done

lemma *exec-gpv-inline*:
fixes *callee* :: (*'c*, *'r*, *'s*) *callee*
and *gpv* :: *'s' ⇒ 'c' ⇒ ('r' × 's', 'c, 'r) gpv*
shows *exec-gpv callee (inline gpv c' s') s =*
map-spmf (λ(x, s', s). ((x, s'), s)) (exec-gpv (λ(s', s) y. map-spmf (λ((x, s'), s). (x, s', s)) (exec-gpv callee (gpv s' y) s)) c' (s', s))
(**is** *?lhs = ?rhs*)
proof −
have *?lhs = map-spmf projl (map-spmf (map-sum (λ(x, s2, y). ((x, s2), y))*
(λ(x, rpv'' :: ('r × 's, unit, unit) rpv, rpv', rpv). (x, rpv'', λr1. bind-gpv (rpv' r1) (λ(r2, y). inline gpv (rpv r2) y))))
(inline2 (λs c. lift-spmf (callee s c)) gpv c' s' s))
unfolding *exec-gpv-conv-inline1* **by**(*simp add*: *inline1-inline-conv-inline2*)
also have ... = *map-spmf (λ(x, s', s). ((x, s'), s)) (map-spmf projl (map-spmf (map-sum id*
(λ(x, rpv'' :: ('r × 's, unit, unit) rpv, rpv', rpv). (x, λr. bind-gpv (rpv'' r) (λ(r1, s1). map-gpv (λ((r2, s2), s1). (r2, s2, s1)) id (inline (λs c. lift-spmf (callee s c)) (rpv' r1) s1)), rpv)))
(inline2 (λs c. lift-spmf (callee s c)) gpv c' s' s)))
unfolding *spmf.map-comp* **by**(*rule map-spmf-cong*[*OF refl*])(*auto dest*!: *in-set-inline2-lift-spmf1*)
also have ... = *?rhs* **unfolding** *exec-gpv-conv-inline1*
by(*subst inline1-inline-conv-inline2'*[*symmetric*])(*simp add*: *spmf.map-comp split-def inline-lift-spmf1 map-lift-spmf*)
finally show *?thesis* **.**
qed

lemma *ord-spmf-exec-gpv*:
assumes *callee*: $\bigwedge s\ x.$ *ord-spmf* (=) (*callee1 s x*) (*callee2 s x*)
shows *ord-spmf* (=) (*exec-gpv callee1 gpv s*) (*exec-gpv callee2 gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *exec-gpv-fixp-parallel-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
next
case (*step exec-gpv1 exec-gpv2*)
show *?case* **using** *step.prems*
by(*clarsimp intro*!: *ord-spmf-bind-reflI ord-spmf-bindI*[*OF assms*] *step.IH split*!: *generat.split*)
qed

context fixes *callee* :: (*'call*, *'ret*, *'s*) *callee* **notes** [[*function-internals*]] **begin**

partial-function (*spmf*) *execp-resumption* :: (*'a*, *'call*, *'ret*) *resumption* ⇒ *'s* ⇒ (*'a* × *'s*) *spmf*
where
execp-resumption r s = (*case r of resumption.Done x* ⇒ *return-pmf* (*map-option* (λ*a*. (*a*, *s*)) *x*)
| *resumption.Pause out c* ⇒ *bind-spmf* (*callee s out*) (λ(*input*, *s'*). *execp-resumption* (*c input*) *s'*))

simps-of-case *execp-resumption-simps* [*simp*]: *execp-resumption.simps*

lemma *execp-resumption-ABORT* [*simp*]: *execp-resumption ABORT s* = *return-pmf None*
by(*simp add*: *ABORT-def*)

lemma *execp-resumption-DONE* [*simp*]: *execp-resumption* (*DONE x*) *s* = *return-spmf* (*x*, *s*)
by(*simp add*: *DONE-def*)

lemma *exec-gpv-lift-resumption*: *exec-gpv callee* (*lift-resumption r*) *s* = *execp-resumption r s*
proof(*induction arbitrary*: *r s rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf exec-gpv.mono execp-resumption.mono exec-gpv-def execp-resumption-def*, *case-names adm bot step*])
case *adm* **show** *?case* **by**(*simp*)
case *bot* **thus** *?case* **by** *simp*
case (*step exec-gpv' execp-resumption'*)
show *?case*
by(*auto split*: *resumption.split option.split simp add*: *lift-resumption.sel intro*: *bind-spmf-cong step*)
qed

lemma *mcont2mcont-execp-resumption* [*THEN spmf.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-execp-resumption*:

mcont resumption-lub resumption-ord lub-spmf (*ord-spmf* (=)) (λ*r*. *execp-resumption r s*)
proof −
have *mcont* (*prod-lub resumption-lub the-Sup*) (*rel-prod resumption-ord* (=)) *lub-spmf* (*ord-spmf* (=)) (*case-prod execp-resumption*)
proof(*rule ccpo.fixp-preserves-mcont2*[*OF ccpo-spmf execp-resumption.mono execp-resumption-def*])
fix *execp-resumption′* :: (′*b*, ′*call*, ′*ret*) *resumption* ⇒ ′*s* ⇒ (′*b* × ′*s*) *spmf*
assume ∗: *mcont* (*prod-lub resumption-lub the-Sup*) (*rel-prod resumption-ord* (=)) *lub-spmf* (*ord-spmf* (=)) (λ(*r*, *s*). *execp-resumption′ r s*)
have [*THEN spmf.mcont2mcont*, *cont-intro*, *simp*]: *mcont resumption-lub resumption-ord lub-spmf* (*ord-spmf* (=)) (λ*r*. *execp-resumption′ r s*)
for *s* **using** ∗ **by** *simp*
have *mcont resumption-lub resumption-ord lub-spmf* (*ord-spmf* (=))
(λ*r*. *case r of resumption.Done x* ⇒ *return-pmf* (*map-option* (λ*a*. (*a*, *s*)) *x*)
| *resumption.Pause out c* ⇒ *bind-spmf* (*callee s out*) (λ(*input*, *s′*). *execp-resumption′* (*c input*) *s′*))
for *s* **by**(*rule mcont-case-resumption*)(*auto simp add*: *ccpo-spmf intro*!: *mcont-bind-spmf*)
thus *mcont* (*prod-lub resumption-lub the-Sup*) (*rel-prod resumption-ord* (=)) *lub-spmf* (*ord-spmf* (=))
(λ(*r*, *s*). *case r of resumption.Done x* ⇒ *return-pmf* (*map-option* (λ*a*. (*a*, *s*)) *x*)
| *resumption.Pause out c* ⇒ *bind-spmf* (*callee s out*) (λ(*input*, *s′*). *execp-resumption′* (*c input*) *s′*))
by *simp*
qed
thus *?thesis* **by** *auto*
qed

lemma *execp-resumption-bind* [*simp*]:
execp-resumption (*r* ≫= *f*) *s* = *execp-resumption r s* ≫= (λ(*x*, *s′*). *execp-resumption* (*f x*) *s′*)
by(*simp add*: *exec-gpv-lift-resumption*[*symmetric*] *lift-resumption-bind exec-gpv-bind*)

lemma *pred-spmf-execp-resumption*:
⋀*A*. ⟦ *pred-resumption A C r*; *pred-fun S* (*pred-fun C* (*pred-spmf* (*pred-prod* (λ-. *True*) *S*))) *callee*; *S s* ⟧
⟹ *pred-spmf* (*pred-prod A S*) (*execp-resumption r s*)
unfolding *exec-gpv-lift-resumption*[*symmetric*]
by(*rule pred-spmf-exec-gpv*) *simp-all*

end

inductive *WT-callee* :: (′*call*, ′*ret*) ℐ ⇒ (′*call* ⇒ (′*ret* × ′*s*) *spmf*) ⇒ *bool* (‹(-) ⊢*c*/ (-) √› [*100*, *0*] *99*)
for ℐ *callee*
where
WT-callee:

⟦ ⋀call ret s. ⟦ call ∈ outs-ℐ ℐ; (ret, s) ∈ set-spmf (callee call) ⟧ ⟹ ret ∈ responses-ℐ ℐ call ⟧
⟹ ℐ ⊢c callee √

lemmas *WT-calleeI* = *WT-callee*
hide-fact *WT-callee*

lemma *WT-calleeD*: ⟦ ℐ ⊢c callee √; (ret, s) ∈ set-spmf (callee out); out ∈ outs-ℐ ℐ ⟧ ⟹ ret ∈ responses-ℐ ℐ out
by(*rule WT-callee.cases*)

lemma *WT-callee-full* [*intro*!, *simp*]: ℐ-*full* ⊢c *callee* √
by(*rule WT-calleeI*) *simp*

lemma *WT-callee-parametric* [*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique R*
shows (*rel-ℐ C R* ===> (*C* ===> *rel-spmf* (*rel-prod R S*)) ===> (=)) *WT-callee WT-callee*
proof −
have ∗: *WT-callee* = (λℐ *callee*. ∀ *call*∈ *outs-ℐ* ℐ. ∀ (*ret*, *s*) ∈ *set-spmf* (*callee call*). *ret* ∈ *responses-ℐ* ℐ *call*)
unfolding *WT-callee.simps* **by** *blast*
show *?thesis* **unfolding** ∗ **by** *transfer-prover*
qed

locale *callee-invariant-on-base* =
fixes *callee* :: ′*s* ⇒ ′*a* ⇒ (′*b* × ′*s*) *spmf*
and *I* :: ′*s* ⇒ *bool*
and ℐ :: (′*a*, ′*b*) ℐ

locale *callee-invariant-on* = *callee-invariant-on-base callee I* ℐ
for *callee* :: ′*s* ⇒ ′*a* ⇒ (′*b* × ′*s*) *spmf*
and *I* :: ′*s* ⇒ *bool*
and ℐ :: (′*a*, ′*b*) ℐ
+
assumes *callee-invariant*: ⋀*s x y s*′. ⟦ (*y*, *s*′) ∈ *set-spmf* (*callee s x*); *I s*; *x* ∈ *outs-ℐ* ℐ ⟧ ⟹ *I s*′
and *WT-callee*: ⋀*s*. *I s* ⟹ ℐ ⊢c *callee s* √
begin

lemma *callee-invariant*′: ⟦ (*y*, *s*′) ∈ *set-spmf* (*callee s x*); *I s*; *x* ∈ *outs-ℐ* ℐ ⟧ ⟹ *I s*′ ∧ *y* ∈ *responses-ℐ* ℐ *x*
by(*auto dest*: *WT-calleeD*[*OF WT-callee*] *callee-invariant*)

lemma *exec-gpv-invariant*′:
⟦ *I s*; ℐ ⊢g *gpv* √ ⟧ ⟹ *set-spmf* (*exec-gpv callee gpv s*) ⊆ {(*x*, *s*′). *I s*′}
proof(*induction arbitrary*: *gpv s rule*: *exec-gpv-fixp-induct*)
case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*)

case *bottom* **show** *?case* **by** *simp*
case *step* **show** *?case* **using** *step.prems*
by(*auto simp add*: *bind-UNION intro*!: *UN-least step.IH del*: *subsetI split*: *generat.split dest*!: *callee-invariant′ elim*: *WT-gpvD*)
qed

lemma *exec-gpv-invariant*:
⟦ (*x*, *s′*) ∈ *set-spmf* (*exec-gpv callee gpv s*); *I s*; ℐ ⊢*g gpv* √ ⟧ ⟹ *I s′*
by(*drule exec-gpv-invariant′*) *blast*+

lemma *interaction-bounded-by-exec-gpv-count′*:
fixes *count*
assumes *bound*: *interaction-bounded-by consider gpv n*
and *count*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s′* ≤ *eSuc* (*count s*)
and *ignore*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s′* ≤ *count s*
and *WT*: ℐ ⊢*g gpv* √
and *I*: *I s*
shows *set-spmf* (*exec-gpv callee gpv s*) ⊆ {(*x*, *s′*). *count s′* ≤ *n* + *count s*}
using *bound I WT*
proof(*induction arbitrary*: *gpv s n rule*: *exec-gpv-fixp-induct*)
case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*)
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv′*)
have *set-spmf* (*exec-gpv′* (*c input*) *s′*) ⊆ {(*x*, *s″*). *count s″* ≤ *n* + *count s*}
if *out*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *input*: (*input*, *s′*) ∈ *set-spmf* (*callee s out*)
and *X*: *out* ∈ *outs-ℐ ℐ*
for *out c input s′*
proof(*cases consider out*)
case *True*
with *step.prems out* **have** *n* > *0*
and *bound′*: *interaction-bounded-by consider* (*c input*) (*n* − *1*)
by(*auto dest*: *interaction-bounded-by-contD*)
note *bound′*
moreover from *input* ‹*I s*› *X* **have** *I s′* **by**(*rule callee-invariant*)
moreover have ℐ ⊢*g c input* √ **using** *step.prems*(*3*) *out WT-calleeD*[*OF WT-callee input*]
by(*rule WT-gpvD*)(*rule step.prems X*)+
ultimately have *set-spmf* (*exec-gpv′* (*c input*) *s′*) ⊆ {(*x*, *s″*). *count s″* ≤ *n* − *1* + *count s′*}
by(*rule step.IH*)
also have ... ⊆ {(*x*, *s″*). *count s″* ≤ *n* + *count s*} **using** ‹*n* > *0*› *count*[*OF input* ‹*I s*› *True X*]
by(*cases n rule*: *co.enat.exhaust*)(*auto*, *metis add-left-mono-trans eSuc-plus iadd-Suc-right*)
finally show *?thesis* **.**
next

case *False*
from *step.prems out this* **have** *bound′*: *interaction-bounded-by consider* (*c input*) *n*
by(*auto dest*: *interaction-bounded-by-contD-ignore*)
from *input* ‹*I s*› *X* **have** *I s′* **by**(*rule callee-invariant*)
note *bound′*
moreover from *input* ‹*I s*› *X* **have** *I s′* **by**(*rule callee-invariant*)
moreover have $\mathcal{I}$ ⊢*g c input* √ **using** *step.prems*(*3*) *out WT-calleeD*[*OF WT-callee input*]
by(*rule WT-gpvD*)(*rule step.prems X*)+
ultimately have *set-spmf* (*exec-gpv′* (*c input*) *s′*) ⊆ {(*x*, *s″*). *count s″* ≤ *n* + *count s′*}
by(*rule step.IH*)
also have ... ⊆ {(*x*, *s″*). *count s″* ≤ *n* + *count s*}
using *ignore*[*OF input* ‹*I s*› *False X*] **by**(*auto elim*: *order-trans*)
finally show *?thesis* **.**
qed
then show *?case* **using** *step.prems*(*3*)
by(*auto 4 3 simp add*: *bind-UNION del*: *subsetI intro*!: *UN-least split*: *generat.split dest*: *WT-gpvD*)
qed

lemma *interaction-bounded-by-exec-gpv-count*:
fixes *count*
assumes *bound*: *interaction-bounded-by consider gpv n*
and *xs′*: (*x*, *s′*) ∈ *set-spmf* (*exec-gpv callee gpv s*)
and *count*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *count s′* ≤ *eSuc* (*count s*)
and *ignore*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *count s′* ≤ *count s*
and *WT*: $\mathcal{I}$ ⊢*g gpv* √
and *I*: *I s*
shows *count s′* ≤ *n* + *count s*
using *bound count ignore WT I*
by(*rule interaction-bounded-by-exec-gpv-count′*[*THEN subsetD*, *OF - - - - - xs′*, *unfolded mem-Collect-eq prod.case*])

lemma *interaction-bounded-by′-exec-gpv-count*:
fixes *count*
assumes *bound*: *interaction-bounded-by′ consider gpv n*
and *xs′*: (*x*, *s′*) ∈ *set-spmf* (*exec-gpv callee gpv s*)
and *count*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *count s′* ≤ *Suc* (*count s*)
and *ignore*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *count s′* ≤ *count s*
and *outs*: $\mathcal{I}$ ⊢*g gpv* √
and *I*: *I s*
shows *count s′* ≤ *n* + *count s*
using *interaction-bounded-by-exec-gpv-count*[*OF bound xs′*, *of count*] *count ignore*

outs I
by(*simp add*: *eSuc-enat*)

lemma *pred-spmf-calleeI*: ⟦ *I s*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *pred-spmf* (*pred-prod* (λ-. *True*) *I*) (*callee s x*)
by(*auto simp add*: *pred-spmf-def dest*: *callee-invariant*)

lemma *lossless-exec-gpv*:
  assumes *gpv*: *lossless-gpv ℐ gpv*
  and *callee*: ⋀*s out*. ⟦ *out* ∈ *outs-ℐ ℐ*; *I s* ⟧ ⟹ *lossless-spmf* (*callee s out*)
  and *WT-gpv*: *ℐ* ⊢*g gpv* √
  and *I*: *I s*
  shows *lossless-spmf* (*exec-gpv callee gpv s*)
using *gpv WT-gpv I*
proof(*induction arbitrary*: *s rule*: *lossless-WT-gpv-induct*)
  case (*lossless-gpv gpv*)
  show *?case* **using** *lossless-gpv.hyps lossless-gpv.prems*
    by(*subst exec-gpv.simps*)(*fastforce split*: *generat.split simp add*: *callee intro*!: *lossless-gpv.IH intro*: *WT-calleeD*[*OF WT-callee*] *elim*!: *callee-invariant*)
qed

lemma *in-set-spmf-exec-gpv-into-results-gpv*:
  assumes ∗: (*x*, *s*′) ∈ *set-spmf* (*exec-gpv callee gpv s*)
  and *WT-gpv* : *ℐ* ⊢*g gpv* √
  and *I*: *I s*
  shows *x* ∈ *results-gpv ℐ gpv*
proof −
  have *set-spmf* (*exec-gpv callee gpv s*) ⊆ *results-gpv ℐ gpv* × *UNIV*
    using *WT-gpv I*
  proof(*induction arbitrary*: *gpv s rule*: *exec-gpv-fixp-induct*)
    { case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*) **}**
    { case *bottom* **show** *?case* **by** *simp* **}**
    case (*step exec-gpv*′)
    { fix *out c ret s*′
      assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
        and *ret*: (*ret*, *s*′) ∈ *set-spmf* (*callee s out*)
      from *step.prems*(*1*) *IO* **have** *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpvD*)
      with *WT-callee*[*OF* ‹*I s*›] *ret* **have** *ret* ∈ *responses-ℐ ℐ out* **by**(*rule WT-calleeD*)
      with *step.prems*(*1*) *IO* **have** *ℐ* ⊢*g c ret* √ **by**(*rule WT-gpvD*)
      moreover from *ret* ‹*I s*› ‹*out* ∈ *outs-ℐ ℐ*› **have** *I s*′ **by**(*rule callee-invariant*)
      ultimately have *set-spmf* (*exec-gpv*′ (*c ret*) *s*′) ⊆ *results-gpv ℐ* (*c ret*) × *UNIV*
        by(*rule step.IH*)
      also have . . . ⊆ *results-gpv ℐ gpv* × *UNIV* **using** *IO* ‹*ret* ∈ -›
        by(*auto intro*: *results-gpv.IO*)
      finally have *set-spmf* (*exec-gpv*′ (*c ret*) *s*′) ⊆ *results-gpv ℐ gpv* × *UNIV* **. }**
    then show *?case* **using** *step.prems*
    by(*auto simp add*: *bind-UNION intro*!: *UN-least del*: *subsetI split*: *generat.split intro*: *results-gpv.Pure*)

qed
thus *x* ∈ *results-gpv* $\mathcal{I}$ *gpv* **using** ∗ **by** *blast*+
qed

end

lemma *callee-invariant-on-alt-def*:
callee-invariant-on = (λ*callee I* $\mathcal{I}$.
(∀ *s* ∈ *Collect I*. ∀ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$. ∀ (*y*, *s*′) ∈ *set-spmf* (*callee s x*). *I s*′) ∧
(∀ *s* ∈ *Collect I*. $\mathcal{I}$ ⊢*c callee s* √))
unfolding *callee-invariant-on-def* **by** *blast*

lemma *callee-invariant-on-parametric* [*transfer-rule*]: **includes** *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique R bi-total S*
shows ((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) ===> (*S* ===> (=)) ===> *rel-*$\mathcal{I}$ *C R* ===> (=))
callee-invariant-on callee-invariant-on
unfolding *callee-invariant-on-alt-def* **by** *transfer-prover*

lemma *callee-invariant-on-cong*:
⟦ *I* = *I*′; *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}$′;
⋀*s x*. ⟦ *I*′ *s*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$′ ⟧ ⟹ *set-spmf* (*callee s x*) ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* × *Collect I*′ ⟷ *set-spmf* (*callee*′ *s x*) ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}$′ *x* × *Collect I*′ ⟧
⟹ *callee-invariant-on callee I* $\mathcal{I}$ = *callee-invariant-on callee*′ *I*′ $\mathcal{I}$′
unfolding *callee-invariant-on-def WT-callee.simps*
by *safe*((*erule meta-allE*)+, (*erule* (*1*) *meta-impE*)+, *force*)+

abbreviation *callee-invariant* :: (′*s* ⇒ ′*a* ⇒ (′*b* × ′*s*) *spmf*) ⇒ (′*s* ⇒ *bool*) ⇒ *bool*
where *callee-invariant callee I* ≡ *callee-invariant-on callee I* $\mathcal{I}$*-full*

interpretation *oi-True*: *callee-invariant-on callee* λ-. *True* $\mathcal{I}$*-full* **for** *callee*
by *unfold-locales* (*simp-all*)

lemma *callee-invariant-on-return-spmf* [*simp*]:
callee-invariant-on (λ*s x*. *return-spmf* (*f s x*)) *I* $\mathcal{I}$ ⟷ (∀ *s*. ∀ *x*∈*outs-*$\mathcal{I}$ $\mathcal{I}$. *I s* ⟶ *I* (*snd* (*f s x*)) ∧ *fst* (*f s x*) ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x*)
by(*auto simp add*: *callee-invariant-on-def split-pairs WT-callee.simps*)

lemma *callee-invariant-return-spmf* [*simp*]:
callee-invariant (λ*s x*. *return-spmf* (*f s x*)) *I* ⟷ (∀ *s x*. *I s* ⟶ *I* (*snd* (*f s x*)))
by(*auto simp add*: *callee-invariant-on-def split-pairs*)

lemma *callee-invariant-restrict-relp*:
includes *lifting-syntax*
assumes (*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) *callee1 callee2*
and *callee-invariant callee1 I1*
and *callee-invariant callee2 I2*
shows ((*S* ↾ *I1* ⊗ *I2*) ===> *C* ===> *rel-spmf* (*rel-prod R* (*S* ↾ *I1* ⊗ *I2*))) *callee1 callee2*

proof −
  interpret *ci1*: *callee-invariant-on callee1 I1 $\mathcal{I}$-full* **by** *fact*
  interpret *ci2*: *callee-invariant-on callee2 I2 $\mathcal{I}$-full* **by** *fact*
  show *?thesis* **using** *assms*(*1*)
    by(*intro rel-funI*)(*auto simp add*: *restrict-rel-prod2 intro*!: *rel-spmf-restrict-relpI intro*: *ci1.pred-spmf-calleeI ci2.pred-spmf-calleeI dest*: *rel-funD rel-setD1 rel-setD2*)
qed

lemma *callee-invariant-on-True* [*simp*]: *callee-invariant-on callee* (λ-. *True*) $\mathcal{I}$ $\longleftrightarrow$ ($\forall$ *s*. $\mathcal{I}$ $\vdash$*c callee s* $\surd$)
by(*simp add*: *callee-invariant-on-def*)

lemma *lossless-exec-gpv*:
  ⟦ *lossless-gpv* $\mathcal{I}$ *gpv*; $\bigwedge$*s out*. *out* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}$ $\Longrightarrow$ *lossless-spmf* (*callee s out*);
    $\mathcal{I}$ $\vdash$*g gpv* $\surd$; $\bigwedge$*s*. $\mathcal{I}$ $\vdash$*c callee s* $\surd$ ⟧
  $\Longrightarrow$ *lossless-spmf* (*exec-gpv callee gpv s*)
by(*rule callee-invariant-on.lossless-exec-gpv*; *simp*)

lemma *in-set-spmf-exec-gpv-into-results′-gpv*:
  assumes $*$: (*x*, *s′*) $\in$ *set-spmf* (*exec-gpv callee gpv s*)
  shows *x* $\in$ *results′-gpv gpv*
using *oi-True.in-set-spmf-exec-gpv-into-results-gpv*[*OF* $*$] **by**(*simp add*: *results-gpv-$\mathcal{I}$-full*)

context fixes $\mathcal{I}$ :: (*′out*, *′in*) $\mathcal{I}$ **begin**

primcorec *restrict-gpv* :: (*′a*, *′out*, *′in*) *gpv* $\Rightarrow$ (*′a*, *′out*, *′in*) *gpv*
where
  restrict-gpv gpv = *GPV* (
  map-pmf (*case-option None* (*case-generat* (*Some* $\circ$ *Pure*)
      (λ*out c*. *if out* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}$ *then Some* (*IO out* (λ*input*. *if input* $\in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *out then restrict-gpv* (*c input*) *else Fail*))
        else None)))
    (*the-gpv gpv*))

lemma *restrict-gpv-Done* [*simp*]: *restrict-gpv* (*Done x*) = *Done x*
by(*rule gpv.expand*)(*simp*)

lemma *restrict-gpv-Fail* [*simp*]: *restrict-gpv Fail* = *Fail*
by(*rule gpv.expand*)(*simp*)

lemma *restrict-gpv-Pause* [*simp*]: *restrict-gpv* (*Pause out c*) = (*if out* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}$ *then Pause out* (λ*input*. *if input* $\in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *out then restrict-gpv* (*c input*) *else Fail*) *else Fail*)
by(*rule gpv.expand*)(*simp*)

lemma *restrict-gpv-bind* [*simp*]: *restrict-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*restrict-gpv gpv*) (λ*x*. *restrict-gpv* (*f x*))
apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)

apply(*auto 4 3 simp del*: *bind-gpv-sel′ simp add*: *bind-gpv.sel bind-spmf-def pmf.rel-map bind-map-pmf rel-fun-def intro*!: *rel-pmf-bind-reflI rel-pmf-reflI split*!: *option.split generat.split split*: *if-split-asm*)
done

lemma *WT-restrict-gpv* [*simp*]: $\mathcal{I} \vdash g$ *restrict-gpv gpv* $\surd$
apply(*coinduction arbitrary*: *gpv*)
apply(*clarsimp split*: *option.split-asm*)
apply(*split generat.split-asm*; *auto split*: *if-split-asm*)
done

lemma *exec-gpv-restrict-gpv*:
  assumes $\mathcal{I} \vdash g$ *gpv* $\surd$ **and** *WT-callee*: $\bigwedge s.\ \mathcal{I} \vdash c$ *callee s* $\surd$
  shows *exec-gpv callee* (*restrict-gpv gpv*) *s* = *exec-gpv callee gpv s*
using *assms*(*1*)
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*
    by(*auto 4 3 simp add*: *bind-spmf-def bind-map-pmf in-set-spmf*[*symmetric*] *WT-gpv-OutD*[*OF step.prems*] *WT-calleeD*[*OF WT-callee*] *intro*!: *bind-pmf-cong*[*OF refl*] *step.IH split*!: *option.split generat.split intro*: *WT-gpv-ContD*[*OF step.prems*])
qed

lemma *in-outs′-restrict-gpvD*: $x \in$ *outs′-gpv* (*restrict-gpv gpv*) $\Longrightarrow x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$
apply(*induction gpv′≡restrict-gpv gpv arbitrary*: *gpv rule*: *outs′-gpv-induct*)
apply(*clarsimp split*: *option.split-asm*; *split generat.split-asm*; *clarsimp split*: *if-split-asm*)+
done

lemma *outs′-restrict-gpv*: *outs′-gpv* (*restrict-gpv gpv*) $\subseteq$ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*blast intro*: *in-outs′-restrict-gpvD*)

lemma *lossless-restrict-gpvI*: ⟦ *lossless-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ $\Longrightarrow$ *lossless-gpv* $\mathcal{I}$ (*restrict-gpv gpv*)
apply(*induction rule*: *lossless-gpv-induct*)
apply(*rule lossless-gpvI*)
subgoal by(*clarsimp simp add*: *lossless-map-pmf lossless-iff-set-pmf-None in-set-spmf*[*symmetric*] *WT-gpv-OutD split*: *option.split-asm generat.split-asm if-split-asm*)
subgoal by(*clarsimp split*: *option.split-asm*; *split generat.split-asm*; *force simp add*: *fun-eq-iff in-set-spmf*[*symmetric*] *split*: *if-split-asm intro*: *WT-gpv-ContD*)
done

lemma *lossless-restrict-gpvD*: ⟦ *lossless-gpv* $\mathcal{I}$ (*restrict-gpv gpv*); $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ $\Longrightarrow$ *lossless-gpv* $\mathcal{I}$ *gpv*
proof(*induction gpv′≡restrict-gpv gpv arbitrary*: *gpv rule*: *lossless-gpv-induct*)
  case (*lossless-gpv p*)
  from *lossless-gpv.hyps*(*4*) **have** *p*: *p* = *the-gpv* (*restrict-gpv gpv*) **by**(*cases restrict-gpv gpv*) *simp*
  show *?case*

proof(*rule lossless-gpvI*)
from *lossless-gpv.hyps*(*1*) **show** *lossless-spmf* (*the-gpv gpv*)
by(*auto simp add*: *p lossless-iff-set-pmf-None intro*: *rev-image-eqI*)

fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) **and** *input*: *input* ∈ *responses-ℐ ℐ out*
from *lossless-gpv.prems*(*1*) *IO* **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpv-OutD*)
hence *IO out* (λ*input. if input* ∈ *responses-ℐ ℐ out then restrict-gpv* (*c input*) *else Fail*) ∈ *set-spmf p* **using** *IO*
by(*auto simp add*: *p in-set-spmf intro*: *rev-bexI*)
from *lossless-gpv.hyps*(*3*)[*OF this input*, *of c input*] *WT-gpvD*[*OF lossless-gpv.prems IO*] *input*
show *lossless-gpv ℐ* (*c input*) **by** *simp*
qed
qed

lemma *colossless-restrict-gpvD*:
⟦ *colossless-gpv ℐ* (*restrict-gpv gpv*); *ℐ* ⊢*g gpv* √ ⟧ ⟹ *colossless-gpv ℐ gpv*
proof(*coinduction arbitrary*: *gpv*)
case (*colossless-gpv gpv*)
have *?lossless-spmf* **using** *colossless-gpv*(*1*)[*THEN colossless-gpv-lossless-spmfD*]
by(*auto simp add*: *lossless-iff-set-pmf-None intro*: *rev-image-eqI*)
moreover have *?continuation*
proof(*intro strip disjI1*)
fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) **and** *input*: *input* ∈ *responses-ℐ ℐ out*
from *colossless-gpv*(*2*) *IO* **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpv-OutD*)
hence *IO out* (λ*input. if input* ∈ *responses-ℐ ℐ out then restrict-gpv* (*c input*) *else Fail*) ∈ *set-spmf* (*the-gpv* (*restrict-gpv gpv*))
using *IO* **by**(*auto simp add*: *in-set-spmf intro*: *rev-bexI*)
from *colossless-gpv-continuationD*[*OF colossless-gpv*(*1*) *this input*] *input WT-gpv-ContD*[*OF colossless-gpv*(*2*) *IO input*]
show ∃ *gpv. c input* = *gpv* ∧ *colossless-gpv ℐ* (*restrict-gpv gpv*) ∧ *ℐ* ⊢*g gpv* √ **by** *simp*
qed
ultimately show *?case* **..**
qed

lemma *colossless-restrict-gpvI*:
⟦ *colossless-gpv ℐ gpv*; *ℐ* ⊢*g gpv* √ ⟧ ⟹ *colossless-gpv ℐ* (*restrict-gpv gpv*)
proof(*coinduction arbitrary*: *gpv*)
case (*colossless-gpv gpv*)
have *?lossless-spmf* **using** *colossless-gpv*(*1*)[*THEN colossless-gpv-lossless-spmfD*]
by(*auto simp add*: *lossless-iff-set-pmf-None in-set-spmf*[*symmetric*] *split*: *option.split-asm generat.split-asm if-split-asm dest*: *WT-gpv-OutD*[*OF colossless-gpv*(*2*)])
moreover have *?continuation*
proof(*intro strip disjI1*)

fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv* (*restrict-gpv gpv*)) **and** *input*: *input* ∈ *responses-ℐ ℐ out*
then obtain *c′* **where** *out*: *out* ∈ *outs-ℐ ℐ*
and *c*: *c* = (*λinput. if input* ∈ *responses-ℐ ℐ out then restrict-gpv* (*c′ input*) *else Fail*)
and *IO′*: *IO out c′* ∈ *set-spmf* (*the-gpv gpv*)
by(*clarsimp split*: *option.split-asm*; *split generat.split-asm*; *clarsimp simp add*: *in-set-spmf split*: *if-split-asm*)
with *input WT-gpv-ContD*[*OF colossless-gpv*(*2*) *IO′ input*] *colossless-gpv-continuationD*[*OF colossless-gpv*(*1*) *IO′ input*]
show ∃ *gpv. c input* = *restrict-gpv gpv* ∧ *colossless-gpv ℐ gpv* ∧ *ℐ* ⊢*g gpv* √ **by**(*auto*)
qed
ultimately show *?case* **..**
qed

lemma *gen-colossless-restrict-gpv* [*simp*]:
ℐ ⊢*g gpv* √ ⟹ *gen-lossless-gpv b ℐ* (*restrict-gpv gpv*) ⟷ *gen-lossless-gpv b ℐ gpv*
by(*cases b*)(*auto intro*: *lossless-restrict-gpvI lossless-restrict-gpvD colossless-restrict-gpvI colossless-restrict-gpvD*)

lemma *interaction-bound-restrict-gpv*:
interaction-bound consider (*restrict-gpv gpv*) ≤ *interaction-bound consider 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* **using** *step.hyps*(*1*)[*of Fail*]
by(*fastforce simp add*: *SUP-UNION set-spmf-def bind-UNION intro*: *SUP-mono rev-bexI step.IH split*: *option.split generat.split*)
qed

lemma *interaction-bounded-by-restrict-gpvI* [*interaction-bound*, *simp*]:
interaction-bounded-by consider gpv n ⟹ *interaction-bounded-by consider* (*restrict-gpv gpv*) *n*
using *interaction-bound-restrict-gpv*[*of consider gpv*] **by**(*simp add*: *interaction-bounded-by.simps*)

end

lemma *restrict-gpv-parametric′*:
includes *lifting-syntax*
notes [*transfer-rule*] = *the-gpv-parametric′ Fail-parametric′ corec-gpv-parametric′*
assumes [*transfer-rule*]: *bi-unique C bi-unique R*
shows (*rel-ℐ C R* ===> *rel-gpv′′ A C R* ===> *rel-gpv′′ A C R*) *restrict-gpv restrict-gpv*
unfolding *restrict-gpv-def* **by** *transfer-prover*

lemma *restrict-gpv-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
bi-unique C $\Longrightarrow$ (*rel-$\mathcal{I}$ C* (=) ===> *rel-gpv A C* ===> *rel-gpv A C*) *restrict-gpv restrict-gpv*
using *restrict-gpv-parametric'*[*of C* (=) *A*]
by(*simp add*: *bi-unique-eq rel-gpv-conv-rel-gpv''*)

lemma *map-restrict-gpv*: *map-gpv f id* (*restrict-gpv $\mathcal{I}$ gpv*) = *restrict-gpv $\mathcal{I}$* (*map-gpv f id gpv*)
for *gpv* :: (*'a*, *'out*, *'ret*) *gpv*
using *restrict-gpv-parametric*[*of BNF-Def.Grp UNIV* (*id* :: *'out* $\Rightarrow$ *'out*) *BNF-Def.Grp UNIV f*, **where** *?'c='ret*]
unfolding *gpv.rel-Grp* **by**(*simp add*: *eq-alt*[*symmetric*] *rel-$\mathcal{I}$-eq rel-fun-def bi-unique-eq*)(*simp add*: *Grp-def*)

lemma (**in** *callee-invariant-on*) *exec-gpv-restrict-gpv-invariant*:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$ **and** *I s*
shows *exec-gpv callee* (*restrict-gpv $\mathcal{I}$ gpv*) *s* = *exec-gpv callee gpv s*
using *assms*
proof(*induction arbitrary*: *gpv s rule*: *exec-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv'*) **show** *?case* **using** *step.prems*(*2*)
by(*auto 4 3 simp add*: *bind-spmf-def bind-map-pmf in-set-spmf*[*symmetric*] *WT-gpv-OutD*[*OF step.prems*(*1*)] *WT-calleeD*[*OF WT-callee*[*OF step.prems*(*2*)]] *intro*!: *bind-pmf-cong*[*OF refl*] *step.IH split*!: *option.split generat.split intro*: *WT-gpv-ContD*[*OF step.prems*(*1*)] *callee-invariant*)
qed

lemma *in-results-gpv-restrict-gpvD*:
assumes $x \in$ *results-gpv $\mathcal{I}$* (*restrict-gpv $\mathcal{I}'$ gpv*)
shows $x \in$ *results-gpv $\mathcal{I}$ gpv*
using *assms*
apply(*induction gpv'$\equiv$restrict-gpv $\mathcal{I}'$ gpv arbitrary*: *gpv*)
apply(*clarsimp split*: *option.split-asm simp add*: *in-set-spmf*[*symmetric*])
subgoal for ... *y* **by**(*cases y*)(*auto intro*: *results-gpv.intros split*: *if-split-asm*)
apply(*clarsimp split*: *option.split-asm simp add*: *in-set-spmf*[*symmetric*])
subgoal for ... *y* **by**(*cases y*)(*auto intro*: *results-gpv.intros split*: *if-split-asm*)
done

lemma *results-gpv-restrict-gpv*:
results-gpv $\mathcal{I}$ (*restrict-gpv $\mathcal{I}'$ gpv*) $\subseteq$ *results-gpv $\mathcal{I}$ gpv*
by(*blast intro*: *in-results-gpv-restrict-gpvD*)

lemma *in-results'-gpv-restrict-gpvD*:
$x \in$ *results'-gpv* (*restrict-gpv $\mathcal{I}'$ gpv*) $\Longrightarrow$ $x \in$ *results'-gpv gpv*
by(*rule in-results-gpv-restrict-gpvD*[**where** $\mathcal{I}$ = *$\mathcal{I}$-full*, *unfolded results-gpv-$\mathcal{I}$-full*])

primcorec *enforce-$\mathcal{I}$-gpv* :: (*'out*, *'in*) $\mathcal{I}$ $\Rightarrow$ (*'a*, *'out*, *'in*) *gpv* $\Rightarrow$ (*'a*, *'out*, *'in*) *gpv*
where

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

lemma *enforce-$\mathcal{I}$-gpv-Done* [*simp*]: *enforce-$\mathcal{I}$-gpv* $\mathcal{I}$ (*Done x*) = *Done x*
by(*rule gpv.expand*) *simp*

lemma *enforce-$\mathcal{I}$-gpv-Fail* [*simp*]: *enforce-$\mathcal{I}$-gpv* $\mathcal{I}$ *Fail* = *Fail*
by(*rule gpv.expand*) *simp*

lemma *enforce-$\mathcal{I}$-gpv-Pause* [*simp*]:
enforce-$\mathcal{I}$-gpv $\mathcal{I}$ (*Pause out rpv*) =
(*if out* ∈ *outs-$\mathcal{I}$* $\mathcal{I}$ *then Pause out* (λ*input. if input* ∈ *responses-$\mathcal{I}$* $\mathcal{I}$ *out then enforce-$\mathcal{I}$-gpv* $\mathcal{I}$ (*rpv input*) *else Fail*) *else Fail*)
by(*rule gpv.expand*)(*simp add*: *fun-eq-iff*)

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

lemma *enforce-$\mathcal{I}$-gpv-bind-gpv* [*simp*]:
enforce-$\mathcal{I}$-gpv $\mathcal{I}$ (*bind-gpv gpv f*) = *bind-gpv* (*enforce-$\mathcal{I}$-gpv* $\mathcal{I}$ *gpv*) (*enforce-$\mathcal{I}$-gpv* $\mathcal{I}$ ∘ *f*)
by(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
(*auto 4 3 simp add*: *bind-gpv.sel spmf-rel-map bind-map-spmf o-def pred-generat-def elim*!: *generat.set-cases intro*!: *generat.rel-refl-strong rel-spmf-bind-reflI rel-spmf-reflI rel-funI split*!: *if-splits generat.split-asm*)

lemma *enforce-$\mathcal{I}$-gpv-parametric′*:
includes *lifting-syntax*
notes [*transfer-rule*] = *corec-gpv-parametric′ the-gpv-parametric′ Fail-parametric′*
assumes [*transfer-rule*]: *bi-unique C bi-unique R*
shows (*rel-$\mathcal{I}$ C R* ===> *rel-gpv′′ A C R* ===> *rel-gpv′′ A C R*) *enforce-$\mathcal{I}$-gpv enforce-$\mathcal{I}$-gpv*
unfolding *enforce-$\mathcal{I}$-gpv-def top-fun-def* **by**(*transfer-prover*)

lemma *enforce-$\mathcal{I}$-gpv-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
bi-unique C ⟹ (*rel-$\mathcal{I}$ C* (=) ===> *rel-gpv A C* ===> *rel-gpv A C*) *enforce-$\mathcal{I}$-gpv enforce-$\mathcal{I}$-gpv*
unfolding *rel-gpv-conv-rel-gpv′′* **by**(*rule enforce-$\mathcal{I}$-gpv-parametric′*[*OF - bi-unique-eq*])

lemma *WT-enforce-$\mathcal{I}$-gpv* [*simp*]: $\mathcal{I}$ ⊢*g enforce-$\mathcal{I}$-gpv* $\mathcal{I}$ *gpv* √
by(*coinduction arbitrary*: *gpv*)(*auto split*: *generat.split-asm*)

context fixes $\mathcal{I}$ :: (′*out*, ′*in*) $\mathcal{I}$ **begin**

inductive *finite-gpv* :: (′*a*, ′*out*, ′*in*) *gpv* ⇒ *bool*
where

finite-gpvI:
(⋀*out c input*. ⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-ℐ ℐ out* ⟧ ⟹ *finite-gpv* (*c input*)) ⟹ *finite-gpv gpv*

lemmas *finite-gpv-induct*[*consumes 1*, *case-names finite-gpv*, *induct pred*] = *finite-gpv.induct*

lemma *finite-gpvD*: ⟦ *finite-gpv gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-ℐ ℐ out* ⟧ ⟹ *finite-gpv* (*c input*)
by(*auto elim*: *finite-gpv.cases*)

lemma *finite-gpv-Fail* [*simp*]: *finite-gpv Fail*
by(*auto intro*: *finite-gpvI*)

lemma *finite-gpv-Done* [*simp*]: *finite-gpv* (*Done x*)
by(*auto intro*: *finite-gpvI*)

lemma *finite-gpv-Pause* [*simp*]: *finite-gpv* (*Pause x c*) ⟷ (∀ *input* ∈ *responses-ℐ ℐ x*. *finite-gpv* (*c input*))
by(*auto dest*: *finite-gpvD intro*: *finite-gpvI*)

lemma *finite-gpv-lift-spmf* [*simp*]: *finite-gpv* (*lift-spmf p*)
by(*auto intro*: *finite-gpvI*)

lemma *finite-gpv-bind* [*simp*]:
finite-gpv (*gpv* ≫= *f*) ⟷ *finite-gpv gpv* ∧ (∀ *x*∈*results-gpv ℐ gpv*. *finite-gpv* (*f x*))
(**is** *?lhs* = *?rhs*)
proof(*intro iffI conjI ballI*; (*elim conjE*)*?*)
  show *finite-gpv gpv* **if** *?lhs* **using** *that*
  proof(*induction gpv′*≡*gpv* ≫= *f arbitrary*: *gpv*)
    case *finite-gpv*
    show *?case*
    proof(*rule finite-gpvI*)
      fix *out c input*
      assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
        and *input*: *input* ∈ *responses-ℐ ℐ out*
      have *IO out* (λ*input*. *c input* ≫= *f*) ∈ *set-spmf* (*the-gpv* (*gpv* ≫= *f*))
        using *IO* **by**(*auto intro*: *rev-bexI*)
      thus *finite-gpv* (*c input*) **using** *input* **by**(*rule finite-gpv.hyps*) *simp*
    qed
  qed
  show *finite-gpv* (*f x*) **if** *x* ∈ *results-gpv ℐ gpv ?lhs* **for** *x* **using** *that*
  proof(*induction*)
    case (*Pure gpv*)
    show *?case*
    proof
      fix *out c input*
      assume *IO out c* ∈ *set-spmf* (*the-gpv* (*f x*)) *input* ∈ *responses-ℐ ℐ out*

with *Pure* **have** *IO out c* ∈ *set-spmf* (*the-gpv* (*gpv* ≫= *f*)) **by**(*auto intro*: *rev-bexI*)
with *Pure.prems* **show** *finite-gpv* (*c input*) **by**(*rule finite-gpvD*) *fact*
qed
next
case (*IO out c gpv input*)
with *IO.hyps* **have** *IO out* (λ*input. c input* ≫= *f*) ∈ *set-spmf* (*the-gpv* (*gpv* ≫= *f*))
by(*auto intro*: *rev-bexI*)
with *IO.prems* **have** *finite-gpv* (*c input* ≫= *f*) **using** *IO.hyps*(*2*) **by**(*rule finite-gpvD*)
thus *?case* **by**(*rule IO.IH*)
qed
show *?lhs* **if** *finite-gpv gpv* ∀ *x*∈*results-gpv* ℐ *gpv. finite-gpv* (*f x*) **using** *that*
proof *induction*
case (*finite-gpv gpv*)
show *?case*
proof(*rule finite-gpvI*)
fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv* (*gpv* ≫= *f*)) **and** *input*: *input* ∈ *responses-ℐ* ℐ *out*
then obtain *generat* **where** *generat*: *generat* ∈ *set-spmf* (*the-gpv gpv*)
and *IO*: *IO out c* ∈ *set-spmf* (*if is-Pure generat then the-gpv* (*f* (*result generat*)) *else*
return-spmf (*IO* (*output generat*) (λ*input. continuation generat input* ≫= *f*)))
by(*auto*)
show *finite-gpv* (*c input*)
proof(*cases generat*)
case (*Pure x*)
with *generat IO* **have** *x* ∈ *results-gpv* ℐ *gpv IO out c* ∈ *set-spmf* (*the-gpv* (*f x*))
by(*auto intro*: *results-gpv.Pure*)
thus *?thesis* **using** *finite-gpv.prems input* **by**(*auto dest*: *finite-gpvD*)
next
case ∗: (*IO out′ c′*)
with *IO generat finite-gpv.prems input* **show** *?thesis*
by(*auto 4 4 intro*: *finite-gpv.IH results-gpv.IO*)
qed
qed
qed
qed

end

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

lemma *finite-gpv-rel′′D1*:
assumes *rel-gpv′′ A C R gpv gpv′* **and** *finite-gpv* ℐ *gpv* **and** ℐ: *rel-ℐ C R* ℐ ℐ′

shows *finite-gpv* $\mathcal{I}'$ *gpv'*
using *assms*(*2*,*1*)
proof(*induction arbitrary*: *gpv'*)
case (*finite-gpv gpv*)
note *finite-gpv.prems*[*transfer-rule*]
show *?case*
proof(*rule finite-gpvI*)
fix *out' c' input'*
assume *IO*: *IO out' c'* ∈ *set-spmf* (*the-gpv gpv'*) **and** *input'*: *input'* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*
have *rel-set* (*rel-generat A C* (*R* ===> (*rel-gpv'' A C R*))) (*set-spmf* (*the-gpv gpv*)) (*set-spmf* (*the-gpv gpv'*))
supply *the-gpv-parametric'*[*transfer-rule*] **by** *transfer-prover*
with *IO input' responses-*$\mathcal{I}$*-parametric*[*THEN rel-funD*, *OF* $\mathcal{I}$] **obtain** *out c input*
where *IO out c* ∈ *set-spmf* (*the-gpv gpv*) *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out rel-gpv'' A C R* (*c input*) (*c' input'*)
by(*auto 4 3 dest*!: *rel-setD2 elim*!: *generat.rel-cases dest*: *rel-funD*)
then show *finite-gpv* $\mathcal{I}'$ (*c' input'*) **by**(*rule finite-gpv.IH*)
qed
qed

lemma *finite-gpv-relD1*: ⟦ *rel-gpv A C gpv gpv'*; *finite-gpv* $\mathcal{I}$ *gpv*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *finite-gpv* $\mathcal{I}$ *gpv'*
using *finite-gpv-rel''D1*[*of A C* (=) *gpv gpv'* $\mathcal{I}$ $\mathcal{I}$] **by**(*simp add*: *rel-gpv-conv-rel-gpv''*)

lemma *finite-gpv-rel''D2*: ⟦ *rel-gpv'' A C R gpv gpv'*; *finite-gpv* $\mathcal{I}$ *gpv'*; *rel-*$\mathcal{I}$ *C R* $\mathcal{I}'$ $\mathcal{I}$ ⟧ ⟹ *finite-gpv* $\mathcal{I}'$ *gpv*
using *finite-gpv-rel''D1*[*of* A^{-1-1} C^{-1-1} R^{-1-1} *gpv' gpv* $\mathcal{I}$ $\mathcal{I}'$] **by**(*simp add*: *rel-gpv''-conversep*)

lemma *finite-gpv-relD2*: ⟦ *rel-gpv A C gpv gpv'*; *finite-gpv* $\mathcal{I}$ *gpv'*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *finite-gpv* $\mathcal{I}$ *gpv*
using *finite-gpv-rel''D2*[*of A C* (=) *gpv gpv'* $\mathcal{I}$ $\mathcal{I}$] **by**(*simp add*: *rel-gpv-conv-rel-gpv''*)

lemma *finite-gpv-parametric'*: (*rel-*$\mathcal{I}$ *C R* ===> *rel-gpv'' A C R* ===> (=)) *finite-gpv finite-gpv*
by(*blast dest*: *finite-gpv-rel''D2 finite-gpv-rel''D1*)

lemma *finite-gpv-parametric* [*transfer-rule*]: (*rel-*$\mathcal{I}$ *C* (=) ===> *rel-gpv A C* ===> (=)) *finite-gpv finite-gpv*
using *finite-gpv-parametric'*[*of C* (=) *A*] **by**(*simp add*: *rel-gpv-conv-rel-gpv''*)

end

lemma *finite-gpv-map* [*simp*]: *finite-gpv* $\mathcal{I}$ (*map-gpv f id gpv*) = *finite-gpv* $\mathcal{I}$ *gpv*
using *finite-gpv-parametric*[*of BNF-Def.Grp UNIV id BNF-Def.Grp UNIV f*]
unfolding *gpv.rel-Grp* **by**(*auto simp add*: *rel-fun-def BNF-Def.Grp-def eq-commute rel-*$\mathcal{I}$*-eq*)

lemma *finite-gpv-assert* [*simp*]: *finite-gpv* $\mathcal{I}$ (*assert-gpv b*)
by(*cases b*) *simp-all*

lemma *finite-gpv-try* [*simp*]:
finite-gpv $\mathcal{I}$ (*TRY gpv ELSE gpv′*) ⟷ *finite-gpv* $\mathcal{I}$ *gpv* ∧ (*colossless-gpv* $\mathcal{I}$ *gpv* ∨ *finite-gpv* $\mathcal{I}$ *gpv′*)
(**is** *?lhs* = -)
proof(*intro iffI conjI*; (*elim conjE disjE*)*?*)
show *1*: *finite-gpv* $\mathcal{I}$ *gpv* **if** *?lhs* **using** *that*
proof(*induction gpv″*≡*TRY gpv ELSE gpv′ arbitrary*: *gpv*)
case (*finite-gpv gpv*)
show *?case*
proof(*rule finite-gpvI*)
fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) **and** *input*: *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*
from *IO* **have** *IO out* (λ*input*. *TRY c input ELSE gpv′*) ∈ *set-spmf* (*the-gpv* (*TRY gpv ELSE gpv′*))
by(*auto simp add*: *image-image generat.map-comp o-def intro*: *rev-image-eqI*)
thus *finite-gpv* $\mathcal{I}$ (*c input*) **using** *input* **by**(*rule finite-gpv.hyps*) *simp*
qed
qed
have *finite-gpv* $\mathcal{I}$ *gpv′* **if** *?lhs* ¬ *colossless-gpv* $\mathcal{I}$ *gpv* **using** *that*
proof(*induction gpv″*≡*TRY gpv ELSE gpv′ arbitrary*: *gpv*)
case (*finite-gpv gpv*)
show *?case*
proof(*cases lossless-spmf* (*the-gpv gpv*))
case *True*
have ∃ *out c input*. *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ∧ *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* ∧ ¬ *colossless-gpv* $\mathcal{I}$ (*c input*)
using *finite-gpv.prems* **by**(*rule contrapos-np*)(*auto intro*: *colossless-gpvI simp add*: *True*)
then obtain *out c input* **where** *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *co′*: ¬ *colossless-gpv* $\mathcal{I}$ (*c input*)
and *input*: *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* **by** *blast*
from *IO* **have** *IO out* (λ*input*. *TRY c input ELSE gpv′*) ∈ *set-spmf* (*the-gpv* (*TRY gpv ELSE gpv′*))
by(*auto simp add*: *image-image generat.map-comp o-def intro*: *rev-image-eqI*)
with *co′* **show** *?thesis* **using** *input* **by**(*blast intro*: *finite-gpv.hyps*(*2*))
next
case *False*
show *?thesis*
proof(*rule finite-gpvI*)
fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv′*) **and** *input*: *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*
from *IO False* **have** *IO out c* ∈ *set-spmf* (*the-gpv* (*TRY gpv ELSE gpv′*)) **by**(*auto intro*: *rev-image-eqI*)

then show *finite-gpv* $\mathcal{I}$ (*c input*) **using** *input* **by**(*rule finite-gpv.hyps*)
qed
qed
qed
then show *colossless-gpv* $\mathcal{I}$ *gpv* ∨ *finite-gpv* $\mathcal{I}$ *gpv′* **if** *?lhs* **using** *that* **by** *blast*

show *?lhs* **if** *finite-gpv* $\mathcal{I}$ *gpv finite-gpv* $\mathcal{I}$ *gpv′* **using** *that*(*1*)
proof *induction*
case (*finite-gpv gpv*)
show *?case*
proof
fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv* (*TRY gpv ELSE gpv′*))
and *input*: *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*
then consider (*gpv*) *c′* **where** *IO out c′* ∈ *set-spmf* (*the-gpv gpv*) *c* = (λ*input*. *TRY c′ input ELSE gpv′*)
| (*gpv′*) *IO out c* ∈ *set-spmf* (*the-gpv gpv′*) **by**(*auto split*: *if-split-asm*)
then show *finite-gpv* $\mathcal{I}$ (*c input*) **using** *input*
by *cases*(*auto intro*: *finite-gpv.IH finite-gpvD*[*OF that*(*2*)])
qed
qed
show *?lhs* **if** *finite-gpv* $\mathcal{I}$ *gpv colossless-gpv* $\mathcal{I}$ *gpv* **using** *that*
proof *induction*
case (*finite-gpv gpv*)
show *?case*
by(*rule finite-gpvI*)(*use finite-gpv.prems* **in** ‹*fastforce split*: *if-split-asm dest*: *colossless-gpvD intro*: *finite-gpv.IH*›)
qed
qed

lemma *lossless-gpv-conv-finite*:
lossless-gpv $\mathcal{I}$ *gpv* ⟷ *finite-gpv* $\mathcal{I}$ *gpv* ∧ *colossless-gpv* $\mathcal{I}$ *gpv*
(**is** *?loss* ⟷ *?fin* ∧ *?co*)
proof(*intro iffI conjI*; (*elim conjE*)*?*)
show *?fin* **if** *?loss* **using** *that* **by** *induction*(*auto intro*: *finite-gpvI*)
show *?co* **if** *?loss* **using** *that* **by** *induction*(*auto intro*: *colossless-gpvI*)
show *?loss* **if** *?fin ?co* **using** *that*
proof *induction*
case (*finite-gpv gpv*)
from *finite-gpv.prems finite-gpv.IH* **show** *?case*
by *cases*(*auto intro*: *lossless-gpvI*)
qed
qed

lemma *colossless-gpv-try* [*simp*]:
colossless-gpv $\mathcal{I}$ (*TRY gpv ELSE gpv′*) ⟷ *colossless-gpv* $\mathcal{I}$ *gpv* ∨ *colossless-gpv* $\mathcal{I}$ *gpv′*
(**is** *?lhs* ⟷ *?gpv* ∨ *?gpv′*)
proof(*intro iffI disjCI*; (*elim disjE*)*?*)

show *?gpv* **if** *?lhs* ¬ *?gpv′* **using** *that*(*1*)
proof(*coinduction arbitrary*: *gpv*)
case (*colossless-gpv gpv*)
have *?lossless-spmf*
proof(*rule ccontr*)
assume *loss*: ¬ *?lossless-spmf*
with *colossless-gpv-lossless-spmfD*[*OF colossless-gpv*(*1*)]
have *gpv′*: *lossless-spmf* (*the-gpv gpv′*) **by** *auto*
have ∃ *out c input*. *IO out c* ∈ *set-spmf* (*the-gpv gpv′*) ∧ *input* ∈ *responses-ℐ ℐ out* ∧ ¬ *colossless-gpv ℐ* (*c input*)
using *that*(*2*) **by**(*rule contrapos-np*)(*auto intro*: *colossless-gpvI gpv′*)
then obtain *out c input*
where *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv′*)
and *co′*: ¬ *colossless-gpv ℐ* (*c input*)
and *input*: *input* ∈ *responses-ℐ ℐ out* **by** *blast*
from *IO loss* **have** *IO out c* ∈ *set-spmf* (*the-gpv* (*TRY gpv ELSE gpv′*))
by(*auto intro*: *rev-image-eqI*)
with *colossless-gpv*(*1*) **have** *colossless-gpv ℐ* (*c input*) **using** *input*
by(*rule colossless-gpv-continuationD*)
with *co′* **show** *False* **by** *contradiction*
qed
moreover have *?continuation*
proof(*intro strip disjI1*; *simp*)
fix *out c input*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) **and** *input*: *input* ∈ *responses-ℐ ℐ out*
hence *IO out* (*λinput*. *TRY c input ELSE gpv′*) ∈ *set-spmf* (*the-gpv* (*TRY gpv ELSE gpv′*))
by(*auto intro*: *rev-image-eqI*)
with *colossless-gpv* **show** *colossless-gpv ℐ* (*TRY c input ELSE gpv′*)
by(*rule colossless-gpv-continuationD*)(*simp add*: *input*)
qed
ultimately show *?case* **..**
qed
show *?lhs* **if** *?gpv′*
proof(*coinduction arbitrary*: *gpv*)
case *colossless-gpv*
show *?case* **using** *colossless-gpvD*[*OF that*] **by**(*auto 4 3*)
qed
show *?lhs* **if** *?gpv* **using** *that*
proof(*coinduction arbitrary*: *gpv*)
case *colossless-gpv*
show *?case* **using** *colossless-gpvD*[*OF colossless-gpv*] **by**(*auto 4 3*)
qed
qed

lemma *lossless-gpv-try* [*simp*]:
lossless-gpv ℐ (*TRY gpv ELSE gpv′*) ⟷
finite-gpv ℐ gpv ∧ (*lossless-gpv ℐ gpv* ∨ *lossless-gpv ℐ gpv′*)

by(*auto simp add*: *lossless-gpv-conv-finite*)

lemma *interaction-any-bounded-by-imp-finite*:
  assumes *interaction-any-bounded-by gpv* (*enat n*)
  shows *finite-gpv* $\mathcal{I}$*-full gpv*
using *assms*
proof(*induction n arbitrary*: *gpv*)
  case *0*
 then show *?case* **by**(*auto intro*: *finite-gpv.intros dest*: *interaction-bounded-by-contD simp add*: *zero-enat-def*[*symmetric*])
next
  case (*Suc n*)
  from *Suc.prems* **show** *?case* **unfolding** *eSuc-enat*[*symmetric*]
    by(*auto 4 4 intro*: *finite-gpv.intros Suc.IH dest*: *interaction-bounded-by-contD*)
qed

lemma *finite-restrict-gpvI* [*simp*]: *finite-gpv* $\mathcal{I}'$ *gpv* $\Longrightarrow$ *finite-gpv* $\mathcal{I}'$ (*restrict-gpv* $\mathcal{I}$ *gpv*)
by(*induction rule*: *finite-gpv-induct*)(*rule finite-gpvI*; *clarsimp split*: *option.split-asm*; *split generat.split-asm*; *clarsimp split*: *if-split-asm simp add*: *in-set-spmf*)

lemma *interaction-bounded-by-exec-gpv-bad-count*:
  fixes *count* **and** *bad* **and** *n* :: *enat* **and** *k* :: *real*
  assumes *bound*: *interaction-bounded-by consider gpv n*
  and *good*: $\neg$ *bad s*
  and *count*: $\bigwedge$*s x y s'*. $[\![$ (*y*, *s'*) $\in$ *set-spmf* (*callee s x*); *consider x*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!]$ $\Longrightarrow$ *count s'* $\leq$ *Suc* (*count s*)
  and *ignore*: $\bigwedge$*s x y s'*. $[\![$ (*y*, *s'*) $\in$ *set-spmf* (*callee s x*); $\neg$ *consider x*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!]$ $\Longrightarrow$ *count s'* $\leq$ *count s*
  and *bad*: $\bigwedge$*s' x*. $[\![$ $\neg$ *bad s'*; *count s'* $<$ *n* $+$ *count s*; *consider x*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!]$ $\Longrightarrow$ *spmf* (*map-spmf* (*bad* $\circ$ *snd*) (*callee s' x*)) *True* $\leq$ *k*
  and *consider*: $\bigwedge$*s x y s'*. $[\![$ (*y*, *s'*) $\in$ *set-spmf* (*callee s x*); $\neg$ *bad s*; *bad s'*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!]$ $\Longrightarrow$ *consider x*
  and *k-nonneg*: *k* $\geq$ *0*
  and *WT-gpv*: $\mathcal{I}$ $\vdash$*g gpv* $\surd$
  and *WT-callee*: $\bigwedge$*s*. $\mathcal{I}$ $\vdash$*c callee s* $\surd$
  shows *spmf* (*map-spmf* (*bad* $\circ$ *snd*) (*exec-gpv callee gpv s*)) *True* $\leq$ *ennreal k* $*$ *n*
using *bound good bad WT-gpv*
proof(*induction arbitrary*: *gpv s n rule*: *exec-gpv-fixp-induct*)
  case *adm* **show** *?case* **by**(*rule cont-intro ccpo-class.admissible-leI*)+
 case *bottom* **show** *?case* **using** *k-nonneg* **by**(*simp add*: *zero-ereal-def*[*symmetric*])
next
  case (*step exec-gpv'*)
  let *?M* = *restrict-space* (*measure-spmf* (*the-gpv gpv*)) {*IO out c*|*out c*. *True*}
 have *ennreal* (*spmf* (*map-spmf* (*bad* $\circ$ *snd*) (*bind-spmf* (*the-gpv gpv*) (*case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out c*. *bind-spmf* (*callee s out*) (λ(*x*, *y*). *exec-gpv'* (*c x*) *y*))))) *True*) =
    ennreal (*spmf* (*bind-spmf* (*the-gpv gpv*) (λ*generat*. *case generat of Pure x* $\Rightarrow$

return-spmf (*bad s*) |
IO out rpv ⇒ *bind-spmf* (*callee s out*) (λ(*x*, *s*′). *map-spmf* (*bad* ∘ *snd*) (*exec-gpv*′ (*rpv x*) *s*′)))) *True*)
(**is** - = *ennreal* (*spmf* (*bind-spmf* - (*case-generat* - *?io*)) -))
by(*simp add*: *map-spmf-bind-spmf o-def generat.case-distrib*[**where** *h*=*map-spmf* -] *split-def cong del*: *generat.case-cong-weak*)
also have ... = ∫ ⁺ *generat*. ∫ ⁺ (*x*, *s*′). *spmf* (*map-spmf* (*bad* ∘ *snd*) (*exec-gpv*′ (*continuation generat x*) *s*′)) *True* ∂*measure-spmf* (*callee s* (*output generat*)) ∂*?M*
using *step.prems*(*2*) **by**(*auto simp add*: *ennreal-spmf-bind nn-integral-restrict-space intro*!: *nn-integral-cong split*: *generat.split*)
also have ... ≤ ∫ ⁺ *generat*. ∫ ⁺ (*x*, *s*′). (*if bad s*′ *then 1 else ennreal k* ∗ (*if consider* (*output generat*) *then n* − *1 else n*)) ∂*measure-spmf* (*callee s* (*output generat*)) ∂*?M*
proof(*clarsimp intro*!: *nn-integral-mono-AE simp add*: *AE-restrict-space-iff split del*: *if-split cong del*: *if-cong*)
show *ennreal* (*spmf* (*map-spmf* (*bad* ∘ *snd*) (*exec-gpv*′ (*rpv ret*) *s*′)) *True*)
≤ (*if bad s*′ *then 1 else ennreal k* ∗ *ennreal-of-enat* (*if consider out then n* − *1 else n*))
if *IO*: *IO out rpv* ∈ *set-spmf* (*the-gpv gpv*)
and *call*: (*ret*, *s*′) ∈ *set-spmf* (*callee s out*)
for *out rpv ret s*′
proof(*cases bad s*′)
case *True*
then show *?thesis* **by**(*simp add*: *pmf-le-1*)
next
case *False*
let *?n*′ = *if consider out then n* − *1 else n*
have *out*: *out* ∈ *outs-ℐ ℐ* **using** *IO step.prems*(*4*) **by**(*simp add*: *WT-gpv-OutD*)
have *bound*′: *interaction-bounded-by consider* (*rpv ret*) *?n*′
using *interaction-bounded-by-contD*[*OF step.prems*(*1*) *IO*]
interaction-bounded-by-contD-ignore[*OF step.prems*(*1*) *IO*] **by**(*auto*)
have *ret* ∈ *responses-ℐ ℐ out* **using** *WT-callee call out* **by**(*rule WT-calleeD*)
with *step.prems*(*4*) *IO* **have** *WT*′: *ℐ* ⊢*g rpv ret* √ **by**(*rule WT-gpv-ContD*)
have *bad*′: *spmf* (*map-pmf* (*map-option* (*bad* ∘ *snd*)) (*callee s*′′ *x*)) *True* ≤ *k*
if ¬ *bad s*′′ **and** *count*′: *count s*′′ < *?n*′ + *count s*′ **and** *consider x* **and** *x* ∈ *outs-ℐ ℐ*
for *s*′′ *x* **using** ‹¬ *bad s*′′› - ‹*consider x*› ‹*x* ∈ *outs-ℐ ℐ*›
proof(*rule step.prems*)
show *count s*′′ < *n* + *count s*
proof(*cases consider out*)
case *True*
with *count*[*OF call True out*] *count*′ *interaction-bounded-by-contD*[*OF step.prems*(*1*) *IO*, *of undefined*]
show *?thesis* **by**(*cases n*)(*auto simp add*: *one-enat-def*)
next
case *False*
with *ignore*[*OF call - out*] *count*′ **show** *?thesis* **by**(*cases n*)*auto*
qed
qed

from *step.IH*[*OF bound′ False this*] *False WT′* **show** *?thesis* **by**(*auto simp add*: *o-def*)
qed
qed
also have ... $= \int^+$ *generat*. $\int^+$ *b*. *indicator* {*True*} *b* + *ennreal k* ∗ (*if consider* (*output generat*) *then n* − *1 else n*) ∗ *indicator* {*False*} *b* ∂*measure-spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s* (*output generat*))) ∂*?M*
(**is** - = $\int^+$ *generat*. $\int^+$ -. - ∂*?O′ generat* ∂-)
by(*auto intro*!: *nn-integral-cong*)
also have ... $= \int^+$ *generat*. ($\int^+$ *b*. *indicator* {*True*} *b* ∂*?O′ generat*) + *ennreal k* ∗ (*if consider* (*output generat*) *then n* − *1 else n*) ∗ $\int^+$ *b*. *indicator* {*False*} *b* ∂*?O′ generat* ∂*?M*
by(*subst nn-integral-add*)(*simp-all add*: *k-nonneg nn-integral-cmult o-def*)
also have ... $= \int^+$ *generat*. *ennreal* (*spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s* (*output generat*))) *True*) + *ennreal k* ∗ (*if consider* (*output generat*) *then n* − *1 else n*) ∗ *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s* (*output generat*))) *False* ∂*?M*
by(*simp del*: *nn-integral-map-spmf add*: *emeasure-spmf-single ereal-of-enat-mult*)
also have ... $\leq \int^+$ *generat*. *ennreal k* ∗ *n* ∂*?M*
proof(*intro nn-integral-mono-AE*, *clarsimp intro*!: *nn-integral-mono-AE simp add*: *AE-restrict-space-iff not-is-Pure-conv split del*: *if-split*)
fix *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
with *step.prems*(*4*) **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpv-OutD*)
show *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s out*)) *True* +
ennreal k ∗ (*if consider out then n* − *1 else n*) ∗ *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s out*)) *False*
≤ *ennreal k* ∗ *n*
proof(*cases consider out*)
case *True*
with *IO* **have** *n* > *0* **using** *interaction-bounded-by-contD*[*OF step.prems*(*1*)] **by**(*blast dest*: *interaction-bounded-by-contD*)
have *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s out*)) *True* ≤ *k* (**is** *?o True* ≤ -)
using ‹¬ *bad s*› *True* ‹*n* > *0*› *out* **by**(*intro step.prems*)(*simp*)
hence *ennreal* (*?o True*) ≤ *k* **using** *k-nonneg* **by**(*simp del*: *o-apply*)
hence *?o True* + *ennreal k* ∗ (*n* − *1*) ∗ *?o False* ≤ *ennreal k* + *ennreal k* ∗ (*n* − *1*) ∗ *ennreal 1*
by(*rule add-mono*)(*rule mult-left-mono*, *simp-all add*: *pmf-le-1 k-nonneg*)
also have ... ≤ *ennreal k* ∗ *n* **using** ‹*n* > *0*›
by(*cases n*)(*auto simp add*: *zero-enat-def ennreal-top-mult gr0-conv-Suc eSuc-enat*[*symmetric*] *field-simps*)
finally show *?thesis* **using** *True* **by**(*simp del*: *o-apply add*: *ereal-of-enat-mult*)
next
case *False*
hence *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s out*)) *True* = *0* **using** ‹¬ *bad s*› *out*
unfolding *spmf-eq-0-set-spmf* **by**(*auto dest*: *consider*)
with *False k-nonneg pmf-le-1*[*of map-spmf* (*bad* ∘ *snd*) (*callee s out*) *Some False*]
show *?thesis* **by**(*simp add*: *mult-left-mono*[*THEN order-trans*, **where** *?b1*=*1*])

qed
qed
also have ... $\leq$ *ennreal k* $*$ *n*
by(*simp add*: *k-nonneg emeasure-restrict-space measure-spmf.emeasure-eq-measure space-restrict-space measure-spmf.subprob-measure-le-1 mult-left-mono*[*THEN order-trans*, **where** *?b1=1*])
finally show *?case* **by**(*simp del*: *o-apply*)
qed

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

lemma *interaction-bounded-by-exec-gpv-bad-count*:
includes *lifting-syntax*
fixes *count* **and** *bad* **and** *n* :: *enat*
assumes *bound*: *interaction-bounded-by consider gpv n*
and *I*: *I s*
and *good*: $\neg$ *bad s*
and *count*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; *consider x*; $x \in$ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *count s'* $\leq$ *Suc* (*count s*)
and *ignore*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; $\neg$ *consider x*; $x \in$ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *count s'* $\leq$ *count s*
and *bad*: $\bigwedge s'\ x$. ⟦ *I s'*; $\neg$ *bad s'*; *count s'* $<$ *n* $+$ *count s*; *consider x*; $x \in$ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *spmf* (*map-spmf* (*bad* $\circ$ *snd*) (*callee s' x*)) *True* $\leq$ *k*
and *consider*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; $\neg$ *bad s*; *bad s'*; $x \in$ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *consider x*
and *k-nonneg*: $k \geq 0$
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *spmf* (*map-spmf* (*bad* $\circ$ *snd*) (*exec-gpv callee gpv s*)) *True* $\leq$ *ennreal k* $*$ *n*
proof $-$
{ **assume** $\exists$(*Rep* :: *'s'* $\Rightarrow$ *'s*) *Abs*. *type-definition Rep Abs* {*s*. *I s*}
then obtain *Rep* :: *'s'* $\Rightarrow$ *'s* **and** *Abs* **where** *td*: *type-definition Rep Abs* {*s*. *I s*} **by** *blast*
then interpret *td*: *type-definition Rep Abs* {*s*. *I s*} **.**
define *cr* **where** *cr* $\equiv$ $\lambda x\ y.\ x =$ *Rep y*
have [*transfer-rule*]: *bi-unique cr right-total cr* **using** *td cr-def* **by**(*rule typedef-bi-unique typedef-right-total*)+
have [*transfer-domain-rule*]: *Domainp cr* $=$ *I* **using** *type-definition-Domainp*[*OF td cr-def*] **by** *simp*

let *?C* $=$ *eq-onp* ($\lambda x.\ x \in$ *outs-$\mathcal{I}$ $\mathcal{I}$*)

define *callee'* **where** *callee'* $\equiv$ (*Rep* −−−> *id* −−−> *map-spmf* (*map-prod id Abs*)) *callee*
have [*transfer-rule*]: (*cr* ===> *?C* ===> *rel-spmf* (*rel-prod* (=) *cr*)) *callee callee'*
by(*auto simp add*: *callee'-def rel-fun-def cr-def spmf-rel-map prod.rel-map td.Abs-inverse eq-onp-def intro*!: *rel-spmf-reflI intro*: *td.Rep*[*simplified*] *dest*: *callee-invariant*)
define *s'* **where** *s'* $\equiv$ *Abs s*

have [*transfer-rule*]: *cr s s'* **using** *I* **by**(*simp add*: *cr-def s'-def td.Abs-inverse*)
define *bad'* **where** *bad'* ≡ (*Rep* −−−> *id*) *bad*
have [*transfer-rule*]: (*cr* ===> (=)) *bad bad'* **by**(*simp add*: *rel-fun-def bad'-def cr-def*)
define *count'* **where** *count'* ≡ (*Rep* −−−> *id*) *count*
have [*transfer-rule*]: (*cr* ===> (=)) *count count'* **by**(*simp add*: *rel-fun-def count'-def cr-def*)

have [*transfer-rule*]: (*?C* ===> (=)) *consider consider* **by**(*simp add*: *eq-onp-def rel-fun-def*)
have [*transfer-rule*]: *rel-ℐ ?C* (=) *ℐ ℐ*
by(*rule rel-ℐI*)(*auto simp add*: *rel-set-eq set-relator-eq-onp eq-onp-same-args dest*: *eq-onp-to-eq*)
note [*transfer-rule*] = *bi-unique-eq-onp bi-unique-eq*

define *gpv'* **where** *gpv'* ≡ *restrict-gpv ℐ gpv*
have [*transfer-rule*]: *rel-gpv* (=) *?C gpv' gpv'*
by(*fold eq-onp-top-eq-eq*)(*auto simp add*: *gpv.rel-eq-onp eq-onp-same-args pred-gpv-def gpv'-def dest*: *in-outs'-restrict-gpvD*)

have *interaction-bounded-by consider gpv' n* **using** *bound* **by**(*simp add*: *gpv'-def*)
moreover have ¬ *bad' s'* **using** *good* **by** *transfer*
moreover have [*rule-format*, *rotated*]:
⋀*s y s'*. ∀ *x* ∈ *outs-ℐ ℐ*. (*y*, *s'*) ∈ *set-spmf* (*callee' s x*) ⟶ *consider x* ⟶ *count' s'* ≤ *Suc* (*count' s*)
by(*transfer fixing*: *consider*)(*blast intro*: *count*)
moreover have [*rule-format*, *rotated*]:
⋀*s y s'*. ∀ *x* ∈ *outs-ℐ ℐ*. (*y*, *s'*) ∈ *set-spmf* (*callee' s x*) ⟶ ¬ *consider x* ⟶ *count' s'* ≤ *count' s*
by(*transfer fixing*: *consider*)(*blast intro*: *ignore*)
moreover have [*rule-format*, *rotated*]:
⋀*s''*. ∀ *x* ∈ *outs-ℐ ℐ*. ¬ *bad' s''* ⟶ *count' s''* < *n* + *count' s'* ⟶ *consider x* ⟶ *spmf* (*map-spmf* (*bad'* ∘ *snd*) (*callee' s'' x*)) *True* ≤ *k*
by(*transfer fixing*: *consider k n*)(*blast intro*: *bad*)
moreover have [*rule-format*, *rotated*]:
⋀*s y s'*. ∀ *x* ∈ *outs-ℐ ℐ*. (*y*, *s'*) ∈ *set-spmf* (*callee' s x*) ⟶ ¬ *bad' s* ⟶ *bad' s'* ⟶ *consider x*
by(*transfer fixing*: *consider*)(*blast intro*: *consider*)
moreover note *k-nonneg*
moreover have *ℐ* ⊢*g gpv'* √ **by**(*simp add*: *gpv'-def*)
moreover have ⋀*s*. *ℐ* ⊢*c callee' s* √ **by** *transfer*(*rule WT-callee*)
ultimately have ∗∗: *spmf* (*map-spmf* (*bad'* ∘ *snd*) (*exec-gpv callee' gpv' s'*)) *True* ≤ *ennreal k* ∗ *n*
by(*rule interaction-bounded-by-exec-gpv-bad-count*)
have [*transfer-rule*]: ((=) ===> *?C* ===> *rel-spmf* (*rel-prod* (=) (=))) *callee callee*
by(*simp add*: *rel-fun-def eq-onp-def prod.rel-eq*)
have *spmf* (*map-spmf* (*bad* ∘ *snd*) (*exec-gpv callee gpv' s*)) *True* ≤ *ennreal k* ∗ *n* **using** ∗∗

by(*transfer*)
also have *exec-gpv callee gpv′ s* = *exec-gpv callee gpv s*
unfolding *gpv′-def* **using** *WT-gpv I* **by**(*rule exec-gpv-restrict-gpv-invariant*)
finally have *?thesis* **. }**
from *this*[*cancel-type-definition*] *I* **show** *?thesis* **by** *blast*
qed

lemma *interaction-bounded-by′-exec-gpv-bad-count*:
fixes *count* **and** *bad* **and** *n* :: *nat*
assumes *bound*: *interaction-bounded-by′ consider gpv n*
and *I*: *I s*
and *good*: ¬ *bad s*
and *count*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s′* ≤ *Suc* (*count s*)
and *ignore*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s′* ≤ *count s*
and *bad*: ⋀*s′ x*. ⟦ *I s′*; ¬ *bad s′*; *count s′* < *n* + *count s*; *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s′ x*)) *True* ≤ *k*
and *consider*: ⋀*s x y s′*. ⟦ (*y*, *s′*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *bad s*; *bad s′*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *consider x*
and *k-nonneg*: *k* ≥ *0*
and *WT-gpv*: *ℐ* ⊢g *gpv* √
shows *spmf* (*map-spmf* (*bad* ∘ *snd*) (*exec-gpv callee gpv s*)) *True* ≤ *k* ∗ *n*
apply(*subst ennreal-le-iff*[*symmetric*], *simp-all add*: *k-nonneg ennreal-mult ennreal-real-conv-ennreal-of-enat del*: *ennreal-of-enat-enat ennreal-le-iff*)
apply(*rule interaction-bounded-by-exec-gpv-bad-count*[*OF bound I - count ignore bad consider k-nonneg WT-gpv*, *OF good*])
apply *simp-all*
done

lemma *interaction-bounded-by-exec-gpv-bad*:
assumes *interaction-any-bounded-by gpv n*
and *I s* ¬ *bad s*
and *bad*: ⋀*s x*. ⟦ *I s*; ¬ *bad s*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s x*)) *True* ≤ *k*
and *k-nonneg*: *0* ≤ *k*
and *WT-gpv*: *ℐ* ⊢g *gpv* √
shows *spmf* (*map-spmf* (*bad* ∘ *snd*) (*exec-gpv callee gpv s*)) *True* ≤ *k* ∗ *n*
using *interaction-bounded-by-exec-gpv-bad-count*[**where** *bad*=*bad*, *OF assms*(*1*) *assms*(*2*−*3*), **where** *?count* = λ-. *0*, *OF - - bad - k-nonneg*] *k-nonneg WT-gpv*
by(*simp add*: *ennreal-real-conv-ennreal-of-enat*[*symmetric*] *ennreal-mult*[*symmetric*] *del*: *ennreal-of-enat-enat*)

end

end

5 Oracle combinators

theory *Computational-Model* **imports**
Generative-Probabilistic-Value
begin

type-synonym *security* = *nat*
type-synonym *advantage* = *security* ⇒ *real*

type-synonym ('σ, 'call, 'ret) *oracle'* = 'σ ⇒ 'call ⇒ ('ret × 'σ) *spmf*
type-synonym ('σ, 'call, 'ret) *oracle* = *security* ⇒ ('σ, 'call, 'ret) *oracle'* × 'σ

print-translation — pretty printing for ('σ, 'call, 'ret) *oracle* ‹

```
  let
    fun tr' [Const (@{type-syntax nat}, -),
      Const (@{type-syntax prod}, -) $
        (Const (@{type-syntax fun}, -) $ s1 $
          (Const (@{type-syntax fun}, -) $ call $
            (Const (@{type-syntax pmf}, -) $
              (Const (@{type-syntax option}, -) $
                (Const (@{type-syntax prod}, -) $ ret $ s2))))) $
        s3] =
      if s1 = s2 andalso s1 = s3 then Syntax.const @{type-syntax oracle} $ s1 $
call $ ret
      else raise Match;
  in [(@{type-syntax fun}, K tr')]
  end
```

›
typ ('σ, 'call, 'ret) *oracle*

5.1 Shared state

context includes *𝓘.lifting* **and** *lifting-syntax* **begin**

lift-definition *plus-𝓘* :: ('out, 'ret) 𝓘 ⇒ ('out', 'ret') 𝓘 ⇒ ('out + 'out', 'ret + 'ret') 𝓘 (**infix** ‹⊕$_{\mathcal{I}}$› *500*)
is *λresp1 resp2. λout. case out of Inl out' ⇒ Inl ‘ resp1 out' | Inr out' ⇒ Inr ‘ resp2 out'* **.**

lemma *plus-𝓘-sel* [*simp*]:
  shows *outs-plus-𝓘*: *outs-𝓘 (plus-𝓘 𝓘l 𝓘r) = outs-𝓘 𝓘l <+> outs-𝓘 𝓘r*
  and *responses-plus-𝓘-Inl*: *responses-𝓘 (plus-𝓘 𝓘l 𝓘r) (Inl x) = Inl ‘ responses-𝓘 𝓘l x*
  and *responses-plus-𝓘-Inr*: *responses-𝓘 (plus-𝓘 𝓘l 𝓘r) (Inr y) = Inr ‘ responses-𝓘 𝓘r y*
by(*transfer*; *auto split*: *sum.split-asm*; *fail*)+

lemma *vimage-Inl-Plus* [*simp*]: *Inl −‘ (A <+> B) = A*
  and *vimage-Inr-Plus* [*simp*]: *Inr −‘ (A <+> B) = B*
by *auto*

lemma *vimage-Inl-image-Inr*: *Inl* −‘ *Inr* ‘ *A* = {}
and *vimage-Inr-image-Inl*: *Inr* −‘ *Inl* ‘ *A* = {}
by *auto*

lemma *plus-$\mathcal{I}$-parametric* [*transfer-rule*]:
(*rel-$\mathcal{I}$ C R* ===> *rel-$\mathcal{I}$ C′ R′* ===> *rel-$\mathcal{I}$* (*rel-sum C C′*) (*rel-sum R R′*)) *plus-$\mathcal{I}$* *plus-$\mathcal{I}$*
apply(*rule rel-funI rel-$\mathcal{I}$I*)+
subgoal premises [*transfer-rule*] **by**(*simp*; *rule conjI*; *transfer-prover*)
apply(*erule rel-sum.cases*; *clarsimp simp add*: *inj-vimage-image-eq vimage-Inl-image-Inr empty-transfer vimage-Inr-image-Inl*)
subgoal premises [*transfer-rule*] **by** *transfer-prover*
subgoal premises [*transfer-rule*] **by** *transfer-prover*
done

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

lemma *$\mathcal{I}$-trivial-plus-$\mathcal{I}$* [*simp*]: *$\mathcal{I}$-trivial* ($\mathcal{I}_1 \oplus_{\mathcal{I}} \mathcal{I}_2$) ⟷ *$\mathcal{I}$-trivial* $\mathcal{I}_1$ ∧ *$\mathcal{I}$-trivial* $\mathcal{I}_2$
by(*auto simp add*: *$\mathcal{I}$-trivial-def*)

end

lemma *map-$\mathcal{I}$-plus-$\mathcal{I}$* [*simp*]:
map-$\mathcal{I}$ (*map-sum f1 f2*) (*map-sum g1 g2*) ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) = *map-$\mathcal{I}$ f1 g1 $\mathcal{I}$1* $\oplus_{\mathcal{I}}$ *map-$\mathcal{I}$ f2 g2 $\mathcal{I}$2*
proof(*rule $\mathcal{I}$-eqI*[*OF Set.set-eqI*], *goal-cases*)
case (*1 x*)
then show *?case* **by**(*cases x*) *auto*
qed (*auto simp add*: *image-image*)

lemma *le-plus-$\mathcal{I}$-iff* [*simp*]:
$\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \leq \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2'$ ⟷ $\mathcal{I}1 \leq \mathcal{I}1'$ ∧ $\mathcal{I}2 \leq \mathcal{I}2'$
by(*auto 4 4 simp add*: *le-$\mathcal{I}$-def dest*: *bspec*[**where** *x*=*Inl* -] *bspec*[**where** *x*=*Inr* -])

lemma *$\mathcal{I}$-full-le-plus-$\mathcal{I}$*: *$\mathcal{I}$-full* ≤ *plus-$\mathcal{I}$ $\mathcal{I}$1 $\mathcal{I}$2* **if** *$\mathcal{I}$-full* ≤ *$\mathcal{I}$1* *$\mathcal{I}$-full* ≤ *$\mathcal{I}$2*
using *that* **by**(*auto simp add*: *le-$\mathcal{I}$-def top-unique*)

lemma *plus-$\mathcal{I}$-mono*: *plus-$\mathcal{I}$ $\mathcal{I}$1 $\mathcal{I}$2* ≤ *plus-$\mathcal{I}$ $\mathcal{I}$1′ $\mathcal{I}$2′* **if** $\mathcal{I}1 \leq \mathcal{I}1'$ $\mathcal{I}2 \leq \mathcal{I}2'$
using *that* **by**(*fastforce simp add*: *le-$\mathcal{I}$-def*)

context
fixes *left* :: (′*s*, ′*a*, ′*b*) *oracle′*
and *right* :: (′*s*,′*c*, ′*d*) *oracle′*
and *s* :: ′*s*
begin

primrec *plus-oracle* :: $'a + 'c \Rightarrow (('b + 'd) \times 's)$ *spmf*
where
plus-oracle (*Inl a*) = *map-spmf* (*apfst Inl*) (*left s a*)
| *plus-oracle* (*Inr b*) = *map-spmf* (*apfst Inr*) (*right s b*)

lemma *lossless-plus-oracleI* [*intro*, *simp*]:
$[\![\bigwedge a.\ x = Inl\ a \Longrightarrow lossless\text{-}spmf\ (left\ s\ a);$
$\bigwedge b.\ x = Inr\ b \Longrightarrow lossless\text{-}spmf\ (right\ s\ b)\ ]\!]$
$\Longrightarrow lossless\text{-}spmf\ (plus\text{-}oracle\ x)$
by(*cases x*) *simp-all*

lemma *plus-oracle-split*:
$P\ (plus\text{-}oracle\ lr) \longleftrightarrow$
$(\forall x.\ lr = Inl\ x \longrightarrow P\ (map\text{-}spmf\ (apfst\ Inl)\ (left\ s\ x))) \wedge$
$(\forall y.\ lr = Inr\ y \longrightarrow P\ (map\text{-}spmf\ (apfst\ Inr)\ (right\ s\ y)))$
by(*cases lr*) *auto*

lemma *plus-oracle-split-asm*:
$P\ (plus\text{-}oracle\ lr) \longleftrightarrow$
$\neg\ ((\exists x.\ lr = Inl\ x \wedge \neg\ P\ (map\text{-}spmf\ (apfst\ Inl)\ (left\ s\ x))) \vee$
$(\exists y.\ lr = Inr\ y \wedge \neg\ P\ (map\text{-}spmf\ (apfst\ Inr)\ (right\ s\ y))))$
by(*cases lr*) *auto*

end

notation *plus-oracle* (**infix** ‹$\oplus_O$› *500*)

context
fixes *left* :: ($'s$, $'a$, $'b$) *oracle'*
and *right* :: ($'s$,$'c$, $'d$) *oracle'*
begin

lemma *WT-plus-oracleI* [*intro*!]:
$[\![\mathcal{I}l \vdash c\ left\ s\ \surd;\ \mathcal{I}r \vdash c\ right\ s\ \surd\ ]\!] \Longrightarrow \mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c\ (left \oplus_O right)\ s\ \surd$
by(*rule WT-calleeI*)(*auto elim*!: *WT-calleeD simp add*: *inj-image-mem-iff*)

lemma *WT-plus-oracleD1*:
assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c\ (left \oplus_O right)\ s\ \surd$ (**is** $?\mathcal{I} \vdash c\ ?callee\ s\ \surd$)
shows $\mathcal{I}l \vdash c\ left\ s\ \surd$
proof(*rule WT-calleeI*)
fix *call ret s'*
assume $call \in outs\text{-}\mathcal{I}\ \mathcal{I}l\ (ret, s') \in set\text{-}spmf\ (left\ s\ call)$
hence $(Inl\ ret, s') \in set\text{-}spmf\ (?callee\ s\ (Inl\ call))\ Inl\ call \in outs\text{-}\mathcal{I}\ (\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r)$
by(*auto intro*: *rev-image-eqI*)
hence $Inl\ ret \in responses\text{-}\mathcal{I}\ ?\mathcal{I}\ (Inl\ call)$ **by**(*rule WT-calleeD*[*OF assms*])
then show $ret \in responses\text{-}\mathcal{I}\ \mathcal{I}l\ call$ **by**(*simp add*: *inj-image-mem-iff*)
qed

lemma *WT-plus-oracleD2*:
  assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c$ (*left* $\oplus_O$ *right*) *s* $\surd$ (**is** *?*$\mathcal{I}$ $\vdash c$ *?callee s* $\surd$)
  shows $\mathcal{I}r \vdash c$ *right s* $\surd$
proof(*rule WT-calleeI*)
  fix *call ret s′*
  assume *call* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}r$ (*ret*, *s′*) ∈ *set-spmf* (*right s call*)
  hence (*Inr ret*, *s′*) ∈ *set-spmf* (*?callee s* (*Inr call*)) *Inr call* ∈ *outs-*$\mathcal{I}$ ($\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$)
    by(*auto intro*: *rev-image-eqI*)
  hence *Inr ret* ∈ *responses-*$\mathcal{I}$ *?*$\mathcal{I}$ (*Inr call*) **by**(*rule WT-calleeD*[*OF assms*])
  then show *ret* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}r$ *call* **by**(*simp add*: *inj-image-mem-iff*)
qed

lemma *WT-plus-oracle-iff* [*simp*]: $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c$ (*left* $\oplus_O$ *right*) *s* $\surd$ ⟷ $\mathcal{I}l \vdash c$ *left s* $\surd$ ∧ $\mathcal{I}r \vdash c$ *right s* $\surd$
by(*blast dest*: *WT-plus-oracleD1 WT-plus-oracleD2*)

lemma *callee-invariant-on-plus-oracle* [*simp*]:
  callee-invariant-on (*left* $\oplus_O$ *right*) *I* ($\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$) ⟷
   callee-invariant-on left I $\mathcal{I}l$ ∧ *callee-invariant-on right I* $\mathcal{I}r$
  (**is** *?lhs* ⟷ *?rhs*)
proof(*intro iffI conjI*)
  assume *?lhs*
  then interpret *plus*: *callee-invariant-on left* $\oplus_O$ *right I* $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$ **.**
  show *callee-invariant-on left I* $\mathcal{I}l$
  proof
    fix *s x y s′*
    assume (*y*, *s′*) ∈ *set-spmf* (*left s x*) **and** *I s* **and** *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}l$
    then have (*Inl y*, *s′*) ∈ *set-spmf* ((*left* $\oplus_O$ *right*) *s* (*Inl x*))
      by(*auto intro*: *rev-image-eqI*)
   then show *I s′* **using** ‹*I s*› **by**(*rule plus.callee-invariant*)(*simp add*: ‹*x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}l$›)
  next
    show $\mathcal{I}l \vdash c$ *left s* $\surd$ **if** *I s* **for** *s* **using** *plus.WT-callee*[*OF that*] **by** *simp*
  qed
  show *callee-invariant-on right I* $\mathcal{I}r$
  proof
    fix *s x y s′*
    assume (*y*, *s′*) ∈ *set-spmf* (*right s x*) **and** *I s* **and** *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}r$
    then have (*Inr y*, *s′*) ∈ *set-spmf* ((*left* $\oplus_O$ *right*) *s* (*Inr x*))
      by(*auto intro*: *rev-image-eqI*)
   then show *I s′* **using** ‹*I s*› **by**(*rule plus.callee-invariant*)(*simp add*: ‹*x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}r$›)
  next
    show $\mathcal{I}r \vdash c$ *right s* $\surd$ **if** *I s* **for** *s* **using** *plus.WT-callee*[*OF that*] **by** *simp*
  qed
next
  assume *?rhs*
  interpret *left*: *callee-invariant-on left I* $\mathcal{I}l$ **using** ‹*?rhs*› **by** *simp*
  interpret *right*: *callee-invariant-on right I* $\mathcal{I}r$ **using** ‹*?rhs*› **by** *simp*

show *?lhs*
proof
fix *s x y s'*
assume $(y,\ s') \in$ *set-spmf* ((*left* $\oplus_O$ *right*) *s x*) **and** *I s* **and** $x \in$ *outs-$\mathcal{I}$* ($\mathcal{I}l$ $\oplus_{\mathcal{I}}$ $\mathcal{I}r$)
then have (*projl y*, *s'*) $\in$ *set-spmf* (*left s* (*projl x*)) $\wedge$ *projl x* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}l$ $\vee$
(*projr y*, *s'*) $\in$ *set-spmf* (*right s* (*projr x*)) $\wedge$ *projr x* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}r$
by (*cases x*) *auto*
then show *I s'* **using** ‹*I s*›
by (*auto dest*: *left.callee-invariant right.callee-invariant*)
next
show $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c$ (*left* $\oplus_O$ *right*) *s* $\surd$ **if** *I s* **for** *s*
using *left.WT-callee*[*OF that*] *right.WT-callee*[*OF that*] **by** *simp*
qed
qed

lemma *callee-invariant-plus-oracle* [*simp*]:
callee-invariant (*left* $\oplus_O$ *right*) *I* $\longleftrightarrow$
callee-invariant left I $\wedge$ *callee-invariant right I*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof −
have *?lhs* $\longleftrightarrow$ *callee-invariant-on* (*left* $\oplus_O$ *right*) *I* ($\mathcal{I}$*-full* $\oplus_{\mathcal{I}}$ $\mathcal{I}$*-full*)
by(*rule callee-invariant-on-cong*)(*auto split*: *plus-oracle-split-asm*)
also have ... $\longleftrightarrow$ *?rhs* **by**(*rule callee-invariant-on-plus-oracle*)
finally show *?thesis* **.**
qed

lemma *plus-oracle-parametric* [*transfer-rule*]:
includes *lifting-syntax* **shows**
((*S* ===> *A* ===> *rel-spmf* (*rel-prod B S*))
===> (*S* ===> *C* ===> *rel-spmf* (*rel-prod D S*))
===> *S* ===> *rel-sum A C* ===> *rel-spmf* (*rel-prod* (*rel-sum B D*) *S*))
plus-oracle plus-oracle
unfolding *plus-oracle-def*[*abs-def*] **by** *transfer-prover*

lemma *rel-spmf-plus-oracle*:
⟦ $\bigwedge$*q1' q2'*. ⟦ *q1* = *Inl q1'*; *q2* = *Inl q2'* ⟧ $\Longrightarrow$ *rel-spmf* (*rel-prod B S*) (*left1 s1 q1'*) (*left2 s2 q2'*);
$\bigwedge$*q1' q2'*. ⟦ *q1* = *Inr q1'*; *q2* = *Inr q2'* ⟧ $\Longrightarrow$ *rel-spmf* (*rel-prod D S*) (*right1 s1 q1'*) (*right2 s2 q2'*);
S s1 s2; *rel-sum A C q1 q2* ⟧
$\Longrightarrow$ *rel-spmf* (*rel-prod* (*rel-sum B D*) *S*) ((*left1* $\oplus_O$ *right1*) *s1 q1*) ((*left2* $\oplus_O$ *right2*) *s2 q2*)
apply(*erule rel-sum.cases*; *clarsimp*)
apply(*erule meta-allE*)+
apply(*erule meta-impE*, *rule refl*)+
subgoal premises [*transfer-rule*] **by** *transfer-prover*
apply(*erule meta-allE*)+
apply(*erule meta-impE*, *rule refl*)+

subgoal premises [*transfer-rule*] **by** *transfer-prover*
done

end

5.2 Shared state with aborts

context
  fixes *left* :: (*'s*, *'a*, *'b option*) *oracle'*
  and *right* :: (*'s*,*'c*, *'d option*) *oracle'*
  and *s* :: *'s*
begin

primrec *plus-oracle-stop* :: *'a* + *'c* ⇒ ((*'b* + *'d*) *option* × *'s*) *spmf*
where
  plus-oracle-stop (*Inl a*) = *map-spmf* (*apfst* (*map-option Inl*)) (*left s a*)
| *plus-oracle-stop* (*Inr b*) = *map-spmf* (*apfst* (*map-option Inr*)) (*right s b*)

lemma *lossless-plus-oracle-stopI* [*intro*, *simp*]:
  ⟦ ⋀*a*. *x* = *Inl a* ⟹ *lossless-spmf* (*left s a*);
    ⋀*b*. *x* = *Inr b* ⟹ *lossless-spmf* (*right s b*) ⟧
  ⟹ *lossless-spmf* (*plus-oracle-stop x*)
by(*cases x*) *simp-all*

lemma *plus-oracle-stop-split*:
  P (*plus-oracle-stop lr*) ⟷
  (∀ *x*. *lr* = *Inl x* ⟶ *P* (*map-spmf* (*apfst* (*map-option Inl*)) (*left s x*))) ∧
  (∀ *y*. *lr* = *Inr y* ⟶ *P* (*map-spmf* (*apfst* (*map-option Inr*)) (*right s y*)))
by(*cases lr*) *auto*

lemma *plus-oracle-stop-split-asm*:
  P (*plus-oracle-stop lr*) ⟷
  ¬ ((∃ *x*. *lr* = *Inl x* ∧ ¬ *P* (*map-spmf* (*apfst* (*map-option Inl*)) (*left s x*))) ∨
    (∃ *y*. *lr* = *Inr y* ∧ ¬ *P* (*map-spmf* (*apfst* (*map-option Inr*)) (*right s y*))))
by(*cases lr*) *auto*

end

notation *plus-oracle-stop* (**infix** ‹$\oplus_O{}^S$› *500*)

5.3 Disjoint state

context
  fixes *left* :: (*'s1*, *'a*, *'b*) *oracle'*
  and *right* :: (*'s2*, *'c*, *'d*) *oracle'*
begin

fun *parallel-oracle* :: (*'s1* × *'s2*, *'a* + *'c*, *'b* + *'d*) *oracle'*
where

parallel-oracle (*s1*, *s2*) (*Inl a*) = *map-spmf* (*map-prod Inl* (λ*s1*′. (*s1*′, *s2*))) (*left s1 a*)
| *parallel-oracle* (*s1*, *s2*) (*Inr b*) = *map-spmf* (*map-prod Inr* (*Pair s1*)) (*right s2 b*)

lemma *parallel-oracle-def*:
parallel-oracle = (λ(*s1*, *s2*). *case-sum* (λ*a*. *map-spmf* (*map-prod Inl* (λ*s1*′. (*s1*′, *s2*))) (*left s1 a*)) (λ*b*. *map-spmf* (*map-prod Inr* (*Pair s1*)) (*right s2 b*)))
by(*auto intro*!: *ext split*: *sum.split*)

lemma *lossless-parallel-oracle* [*simp*]:
lossless-spmf (*parallel-oracle s12 xy*) ⟷
(∀ *x*. *xy* = *Inl x* ⟶ *lossless-spmf* (*left* (*fst s12*) *x*)) ∧
(∀ *y*. *xy* = *Inr y* ⟶ *lossless-spmf* (*right* (*snd s12*) *y*))
by(*cases s12*; *cases xy*) *simp-all*

lemma *parallel-oracle-split*:
P (*parallel-oracle s1s2 lr*) ⟷
(∀ *s1 s2 x*. *s1s2* = (*s1*, *s2*) ⟶ *lr* = *Inl x* ⟶ *P* (*map-spmf* (*map-prod Inl* (λ*s1*′. (*s1*′, *s2*))) (*left s1 x*))) ∧
(∀ *s1 s2 y*. *s1s2* = (*s1*, *s2*) ⟶ *lr* = *Inr y* ⟶ *P* (*map-spmf* (*map-prod Inr* (*Pair s1*)) (*right s2 y*)))
by(*cases s1s2*; *cases lr*) *auto*

lemma *parallel-oracle-split-asm*:
P (*parallel-oracle s1s2 lr*) ⟷
¬ ((∃ *s1 s2 x*. *s1s2* = (*s1*, *s2*) ∧ *lr* = *Inl x* ∧ ¬ *P* (*map-spmf* (*map-prod Inl* (λ*s1*′. (*s1*′, *s2*))) (*left s1 x*))) ∨
(∃ *s1 s2 y*. *s1s2* = (*s1*, *s2*) ∧ *lr* = *Inr y* ∧ ¬ *P* (*map-spmf* (*map-prod Inr* (*Pair s1*)) (*right s2 y*))))
by(*cases s1s2*; *cases lr*) *auto*

lemma *WT-parallel-oracle* [*intro*!, *simp*]:
⟦ $\mathcal{I}l$ ⊢*c left sl* √; $\mathcal{I}r$ ⊢*c right sr* √ ⟧ ⟹ *plus-*$\mathcal{I}$ $\mathcal{I}l$ $\mathcal{I}r$ ⊢*c parallel-oracle* (*sl*, *sr*) √
by(*rule WT-calleeI*)(*auto elim*!: *WT-calleeD simp add*: *inj-image-mem-iff*)

lemma *callee-invariant-parallel-oracleI* [*simp*, *intro*]:
assumes *callee-invariant-on left Il* $\mathcal{I}l$ *callee-invariant-on right Ir* $\mathcal{I}r$
shows *callee-invariant-on parallel-oracle* (*pred-prod Il Ir*) ($\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$)
proof
interpret *left*: *callee-invariant-on left Il* $\mathcal{I}l$ **by** *fact*
interpret *right*: *callee-invariant-on right Ir* $\mathcal{I}r$ **by** *fact*

show *pred-prod Il Ir s12*′
if (*y*, *s12*′) ∈ *set-spmf* (*parallel-oracle s12 x*) **and** *pred-prod Il Ir s12* **and** *x* ∈ *outs-*$\mathcal{I}$ ($\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$)
for *s12 x y s12*′ **using** *that*
by(*cases s12*; *cases s12*; *cases x*)(*auto dest*: *left.callee-invariant right.callee-invariant*)

show $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c$ *local.parallel-oracle s* $\surd$ **if** *pred-prod Il Ir s* **for** *s* **using** *that*
by(*cases s*)(*simp add*: *left.WT-callee right.WT-callee*)
qed

end

lemma *parallel-oracle-parametric*:
includes *lifting-syntax* **shows**
((*S1* ===> *CALL1* ===> *rel-spmf* (*rel-prod* (=) *S1*))
===> (*S2* ===> *CALL2* ===> *rel-spmf* (*rel-prod* (=) *S2*))
===> *rel-prod S1 S2* ===> *rel-sum CALL1 CALL2* ===> *rel-spmf* (*rel-prod* (=) (*rel-prod S1 S2*)))
parallel-oracle parallel-oracle
unfolding *parallel-oracle-def*[*abs-def*] **by** (*fold relator-eq*)*transfer-prover*

5.4 Indexed oracles

definition *family-oracle* :: ($'i \Rightarrow ('s, 'a, 'b)$ *oracle*$'$) $\Rightarrow$ ($'i \Rightarrow 's, 'i \times 'a, 'b$) *oracle*$'$
where *family-oracle f s* = (λ(*i*, *x*). *map-spmf* (λ(*y*, *s*$'$). (*y*, *s*(*i* := *s*$'$))) (*f i* (*s i*) *x*))

lemma *family-oracle-apply* [*simp*]:
family-oracle f s (*i*, *x*) = *map-spmf* (*apsnd* (*fun-upd s i*)) (*f i* (*s i*) *x*)
by(*simp add*: *family-oracle-def apsnd-def map-prod-def*)

lemma *lossless-family-oracle*:
lossless-spmf (*family-oracle f s ix*) $\longleftrightarrow$ *lossless-spmf* (*f* (*fst ix*) (*s* (*fst ix*)) (*snd ix*))
by(*simp add*: *family-oracle-def split-beta*)

5.5 State extension

definition *extend-state-oracle* :: ($'call, 'ret, 's$) *callee* $\Rightarrow$ ($'call, 'ret, 's' \times 's$) *callee* (‹†-› [*1000*] *1000*)
where *extend-state-oracle callee* = (λ(*s*$'$, *s*) *x*. *map-spmf* (λ(*y*, *s*). (*y*, (*s*$'$, *s*))) (*callee s x*))

lemma *extend-state-oracle-simps* [*simp*]:
extend-state-oracle callee (*s*$'$, *s*) *x* = *map-spmf* (λ(*y*, *s*). (*y*, (*s*$'$, *s*))) (*callee s x*)
by(*simp add*: *extend-state-oracle-def*)

context includes *lifting-syntax* **begin**
lemma *extend-state-oracle-parametric* [*transfer-rule*]:
((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) ===> *rel-prod S*$'$ *S* ===> *C* ===> *rel-spmf* (*rel-prod R* (*rel-prod S*$'$ *S*)))
extend-state-oracle extend-state-oracle
unfolding *extend-state-oracle-def*[*abs-def*] **by** *transfer-prover*

lemma *extend-state-oracle-transfer*:

((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*))
===> *rel-prod2 S* ===> *C* ===> *rel-spmf* (*rel-prod R* (*rel-prod2 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
end

lemma *callee-invariant-extend-state-oracle-const* [*simp*]:
callee-invariant †*oracle* (λ(*s*′, *s*). *I s*′)
by *unfold-locales auto*

lemma *callee-invariant-extend-state-oracle-const*′:
callee-invariant †*oracle* (λ*s*. *I* (*fst s*))
by *unfold-locales auto*

definition *lift-stop-oracle* :: (′*call*, ′*ret*, ′*s*) *callee* ⇒ (′*call*, ′*ret option*, ′*s*) *callee*
where *lift-stop-oracle oracle s x* = *map-spmf* (*apfst Some*) (*oracle s x*)

lemma *lift-stop-oracle-apply* [*simp*]: *lift-stop-oracle oracle s x* = *map-spmf* (*apfst Some*) (*oracle s x*)
by(*fact lift-stop-oracle-def*)

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

lemma *lift-stop-oracle-transfer*:
((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) ===> (*S* ===> *C* ===> *rel-spmf* (*rel-prod* (*pcr-Some R*) *S*)))
(λ*x*. *x*) *lift-stop-oracle*
unfolding *lift-stop-oracle-def*
apply(*rule rel-funI*)+
apply(*drule* (*1*) *rel-funD*)+
apply(*simp add*: *spmf-rel-map apfst-def prod.rel-map*)
done

end

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

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

lemma *extend-state-oracle2-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
((*S* ===> *C* ===> *rel-spmf* (*rel-prod R S*)) ===> *rel-prod S S′* ===> *C* ===> *rel-spmf* (*rel-prod R* (*rel-prod S S′*)))
extend-state-oracle2 extend-state-oracle2
unfolding *extend-state-oracle2-def*[*abs-def*] **by** *transfer-prover*

lemma *callee-invariant-extend-state-oracle2-const* [*simp*]:
callee-invariant oracle† (λ(*s*, *s′*). *I s′*)
by *unfold-locales auto*

lemma *callee-invariant-extend-state-oracle2-const′*:
callee-invariant oracle† (λ*s*. *I* (*snd s*))
by *unfold-locales auto*

lemma *extend-state-oracle2-plus-oracle*:
extend-state-oracle2 (*plus-oracle oracle1 oracle2*) = *plus-oracle* (*extend-state-oracle2 oracle1*) (*extend-state-oracle2 oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases s*; *cases q*) (*simp-all add*: *apfst-def spmf.map-comp o-def split-def*)
qed

lemma *parallel-oracle-conv-plus-oracle*:
parallel-oracle oracle1 oracle2 = *plus-oracle* (*oracle1*†) (†*oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases s*; *cases q*) (*auto simp add*: *spmf.map-comp apfst-def o-def split-def map-prod-def*)
qed

lemma *map-sum-parallel-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *map-sum f g* −−−> *map-spmf* (*map-prod* (*map-sum h k*) *id*)) (*parallel-oracle oracle1 oracle2*)
= *parallel-oracle* ((*id* −−−> *f* −−−> *map-spmf* (*map-prod h id*)) *oracle1*) ((*id* −−−> *g* −−−> *map-spmf* (*map-prod k id*)) *oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases s*; *cases q*) (*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed

lemma *map-sum-plus-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *map-sum f g* −−−> *map-spmf* (*map-prod* (*map-sum h k*) *id*)) (*plus-oracle oracle1 oracle2*)
= *plus-oracle* ((*id* −−−> *f* −−−> *map-spmf* (*map-prod h id*)) *oracle1*) ((*id* −−−> *g* −−−> *map-spmf* (*map-prod k id*)) *oracle2*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)

then show *?case* **by** (*cases q*) (*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed

lemma *map-rsuml-plus-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *rsuml* −−−> (*map-spmf* (*map-prod lsumr id*))) (*oracle1* $\oplus_O$ (*oracle2* $\oplus_O$ *oracle3*)) =
((*oracle1* $\oplus_O$ *oracle2*) $\oplus_O$ *oracle3*)
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case*
proof(*cases q*)
case (*Inl ql*)
then show *?thesis* **by**(*cases ql*)(*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed (*simp add*: *spmf.map-comp o-def apfst-def prod.map-comp id-def*)
qed

lemma *map-lsumr-plus-oracle*: **includes** *lifting-syntax* **shows**
(*id* −−−> *lsumr* −−−> (*map-spmf* (*map-prod rsuml id*))) ((*oracle1* $\oplus_O$ *oracle2*) $\oplus_O$ *oracle3*) =
(*oracle1* $\oplus_O$ (*oracle2* $\oplus_O$ *oracle3*))
proof((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case*
proof(*cases q*)
case (*Inr qr*)
then show *?thesis* **by**(*cases qr*)(*simp-all add*: *spmf.map-comp o-def apfst-def prod.map-comp*)
qed (*simp add*: *spmf.map-comp o-def apfst-def prod.map-comp id-def*)
qed

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

definition *lift-state-oracle*
:: (('s ⇒ 'a ⇒ (('b × 't) × 's) *spmf*) ⇒ ('s' ⇒ 'a ⇒ (('b × 't) × 's') *spmf*))
⇒ ('t × 's ⇒ 'a ⇒ ('b × 't × 's) *spmf*) ⇒ ('t × 's' ⇒ 'a ⇒ ('b × 't × 's') *spmf*) **where**
lift-state-oracle F oracle =
(λ(*t*, *s'*) *a*. *map-spmf rprodl* (*F* ((*Pair t* −−−> *id* −−−> *map-spmf lprodr*) *oracle*) *s' a*))

lemma *lift-state-oracle-simps* [*simp*]:
lift-state-oracle F oracle (*t*, *s'*) *a* = *map-spmf rprodl* (*F* ((*Pair t* −−−> *id* −−−> *map-spmf lprodr*) *oracle*) *s' a*)
by(*simp add*: *lift-state-oracle-def*)

lemma *lift-state-oracle-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(((*S* ===> *A* ===> *rel-spmf* (*rel-prod* (*rel-prod B T*) *S*)) ===> *S'* ===>

A ===> *rel-spmf* (*rel-prod* (*rel-prod B T*) *S'*))
===> (*rel-prod T S* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-prod T S*)))
===> *rel-prod T S'* ===> *A* ===> *rel-spmf* (*rel-prod B* (*rel-prod T S'*)))
lift-state-oracle lift-state-oracle
unfolding *lift-state-oracle-def map-fun-def o-def* **by** *transfer-prover*

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

shows *lift-state-oracle F* (*extend-state-oracle oracle*) = *extend-state-oracle* (*G oracle*)
unfolding *lift-state-oracle-def extend-state-oracle-def*
apply(*clarsimp simp add*: *fun-eq-iff map-fun-def o-def spmf.map-comp split-def rprodl-def*)
subgoal for *t s a*
apply(*rule sym*)
apply(*fold spmf-rel-eq*)
apply(*simp add*: *spmf-rel-map*)
apply(*rule rel-spmf-mono*)
apply(*rule assms*[*unfolded Rel-def*, **where** *B*=λ*x* (*y*, *z*). *x* = *y* ∧ *z* = *t*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
apply(*auto simp add*: *rel-fun-def spmf-rel-map intro*!: *rel-spmf-reflI*)
done
done

lemma *lift-state-oracle-compose*:
lift-state-oracle F (*lift-state-oracle G oracle*) = *lift-state-oracle* (*F* ∘ *G*) *oracle*
by(*simp add*: *lift-state-oracle-def map-fun-def o-def split-def spmf.map-comp*)

lemma *lift-state-oracle-id* [*simp*]: *lift-state-oracle id* = *id*
by(*simp add*: *fun-eq-iff spmf.map-comp o-def*)

lemma *rprodl-extend-state-oracle*: **includes** *lifting-syntax* **shows**
(*rprodl* −−−> *id* −−−> *map-spmf* (*map-prod id lprodr*)) (*extend-state-oracle* (*extend-state-oracle oracle*)) =
extend-state-oracle oracle
by(*simp add*: *fun-eq-iff spmf.map-comp o-def split-def*)

end

6 Combining GPVs

6.1 Shared state without interrupts

context
fixes *left* :: *'s* ⇒ *'x1* ⇒ (*'y1* × *'s*, *'call*, *'ret*) *gpv*
and *right* :: *'s* ⇒ *'x2* ⇒ (*'y2* × *'s*, *'call*, *'ret*) *gpv*

begin

primrec *plus-intercept* :: *'s* ⇒ *'x1* + *'x2* ⇒ ((*'y1* + *'y2*) × *'s*, *'call*, *'ret*) *gpv*
where
  plus-intercept s (*Inl x*) = *map-gpv* (*apfst Inl*) *id* (*left s x*)
| *plus-intercept s* (*Inr x*) = *map-gpv* (*apfst Inr*) *id* (*right s x*)

end

lemma *plus-intercept-parametric* [*transfer-rule*]:
  includes *lifting-syntax* **shows**
  ((*S* ===> *X1* ===> *rel-gpv* (*rel-prod Y1 S*) *C*)
  ===> (*S* ===> *X2* ===> *rel-gpv* (*rel-prod Y2 S*) *C*)
  ===> *S* ===> *rel-sum X1 X2* ===> *rel-gpv* (*rel-prod* (*rel-sum Y1 Y2*) *S*) *C*)
  plus-intercept plus-intercept
unfolding *plus-intercept-def*[*abs-def*] **by** *transfer-prover*

lemma *interaction-bounded-by-plus-intercept* [*interaction-bound*]:
  fixes *left right*
  shows ⟦ ⋀*x'*. *x* = *Inl x'* ⟹ *interaction-bounded-by P* (*left s x'*) (*n x'*);
    ⋀*y*. *x* = *Inr y* ⟹ *interaction-bounded-by P* (*right s y*) (*m y*) ⟧
  ⟹ *interaction-bounded-by P* (*plus-intercept left right s x*) (*case x of Inl x* ⇒ *n x* | *Inr y* ⇒ *m y*)
by(*simp split*!: *sum.split add*: *interaction-bounded-by-map-gpv-id*)

6.2 Shared state with interrupts

context
  fixes *left* :: *'s* ⇒ *'x1* ⇒ (*'y1 option* × *'s*, *'call*, *'ret*) *gpv*
  and *right* :: *'s* ⇒ *'x2* ⇒ (*'y2 option* × *'s*, *'call*, *'ret*) *gpv*
begin

primrec *plus-intercept-stop* :: *'s* ⇒ *'x1* + *'x2* ⇒ ((*'y1* + *'y2*) *option* × *'s*, *'call*, *'ret*) *gpv*
where
  plus-intercept-stop s (*Inl x*) = *map-gpv* (*apfst* (*map-option Inl*)) *id* (*left s x*)
| *plus-intercept-stop s* (*Inr x*) = *map-gpv* (*apfst* (*map-option Inr*)) *id* (*right s x*)

end

lemma *plus-intercept-stop-parametric* [*transfer-rule*]:
  includes *lifting-syntax* **shows**
  ((*S* ===> *X1* ===> *rel-gpv* (*rel-prod* (*rel-option Y1*) *S*) *C*)
  ===> (*S* ===> *X2* ===> *rel-gpv* (*rel-prod* (*rel-option Y2*) *S*) *C*)
  ===> *S* ===> *rel-sum X1 X2* ===> *rel-gpv* (*rel-prod* (*rel-option* (*rel-sum Y1 Y2*)) *S*) *C*)
  plus-intercept-stop plus-intercept-stop
unfolding *plus-intercept-stop-def* **by** *transfer-prover*

6.3 One-sided shifts

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

abbreviation *left-rpv* :: (*'a*, *'out*, *'in*) *rpv* ⇒ (*'a*, *'out* + *'out'*, *'in* + *'in'*) *rpv* **where**
left-rpv rpv ≡ *λinput. case input of Inl input'* ⇒ *left-gpv* (*rpv input'*) | - ⇒ *Fail*

primcorec (*transfer*) *right-gpv* :: (*'a*, *'out*, *'in*) *gpv* ⇒ (*'a*, *'out'* + *'out*, *'in'* + *'in*) *gpv* **where**
the-gpv (*right-gpv gpv*) =
map-spmf (*map-generat id Inr* (*λrpv input. case input of Inr input'* ⇒ *right-gpv* (*rpv input'*) | - ⇒ *Fail*)) (*the-gpv gpv*)

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

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

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

lemma *left-gpv-parametric'*:
(*rel-gpv'' A C R* ===> *rel-gpv'' A* (*rel-sum C C'*) (*rel-sum R R'*)) *left-gpv left-gpv*
unfolding *left-gpv-def* **by** *transfer-prover*

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

lemma *right-gpv-parametric'*:
(*rel-gpv'' A C' R'* ===> *rel-gpv'' A* (*rel-sum C C'*) (*rel-sum R R'*)) *right-gpv right-gpv*
unfolding *right-gpv-def* **by** *transfer-prover*

end

lemma *left-gpv-Done* [*simp*]: *left-gpv* (*Done x*) = *Done x*
by(*rule gpv.expand*) *simp*

lemma *right-gpv-Done* [*simp*]: *right-gpv* (*Done x*) = *Done x*
by(*rule gpv.expand*) *simp*

lemma *left-gpv-Pause* [*simp*]:

left-gpv (*Pause x rpv*) = *Pause* (*Inl x*) (*λinput. case input of Inl input′* ⇒ *left-gpv* (*rpv input′*) | - ⇒ *Fail*)
by(*rule gpv.expand*) *simp*

lemma *right-gpv-Pause* [*simp*]:
right-gpv (*Pause x rpv*) = *Pause* (*Inr x*) (*λinput. case input of Inr input′* ⇒ *right-gpv* (*rpv input′*) | - ⇒ *Fail*)
by(*rule gpv.expand*) *simp*

lemma *left-gpv-map*: *left-gpv* (*map-gpv f g gpv*) = *map-gpv f* (*map-sum g h*) (*left-gpv gpv*)
using *left-gpv.transfer*[*of BNF-Def.Grp UNIV f BNF-Def.Grp UNIV g BNF-Def.Grp UNIV h*]
unfolding *sum.rel-Grp gpv.rel-Grp*
by(*auto simp add*: *rel-fun-def Grp-def*)

lemma *right-gpv-map*: *right-gpv* (*map-gpv f g gpv*) = *map-gpv f* (*map-sum h g*) (*right-gpv gpv*)
using *right-gpv.transfer*[*of BNF-Def.Grp UNIV f BNF-Def.Grp UNIV g BNF-Def.Grp UNIV h*]
unfolding *sum.rel-Grp gpv.rel-Grp*
by(*auto simp add*: *rel-fun-def Grp-def*)

lemma *results′-gpv-left-gpv* [*simp*]:
results′-gpv (*left-gpv gpv* :: (*′a*, *′out* + *′out′*, *′in* + *′in′*) *gpv*) = *results′-gpv gpv* (**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv′≡left-gpv gpv* :: (*′a*, *′out* + *′out′*, *′in* + *′in′*) *gpv arbitrary*: *gpv*)
(*fastforce simp add*: *elim*!: *generat.set-cases intro*: *results′-gpvI split*: *sum.splits*)+
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that*
by(*induction*)
(*auto 4 3 elim*!: *generat.set-cases intro*: *results′-gpv-Pure rev-image-eqI results′-gpv-Cont*[**where** *input*=*Inl* -])
qed

lemma *results′-gpv-right-gpv* [*simp*]:
results′-gpv (*right-gpv gpv* :: (*′a*, *′out′* + *′out*, *′in′* + *′in*) *gpv*) = *results′-gpv gpv* (**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
by(*induction gpv′≡right-gpv gpv* :: (*′a*, *′out′* + *′out*, *′in′* + *′in*) *gpv arbitrary*: *gpv*)
(*fastforce simp add*: *elim*!: *generat.set-cases intro*: *results′-gpvI split*: *sum.splits*)+
show *x* ∈ *?lhs* **if** *x* ∈ *?rhs* **for** *x* **using** *that*
by(*induction*)
(*auto 4 3 elim*!: *generat.set-cases intro*: *results′-gpv-Pure rev-image-eqI results′-gpv-Cont*[**where** *input*=*Inr* -])

qed

lemma *left-gpv-Inl-transfer*: *rel-gpv''* (=) (*λl r. l* = *Inl r*) (*λl r. l* = *Inl r*) (*left-gpv gpv*) *gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto simp add*: *spmf-rel-map generat.rel-map del*: *rel-funI intro*!: *rel-spmf-reflI generat.rel-refl-strong rel-funI*)

lemma *right-gpv-Inr-transfer*: *rel-gpv''* (=) (*λl r. l* = *Inr r*) (*λl r. l* = *Inr r*) (*right-gpv gpv*) *gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto simp add*: *spmf-rel-map generat.rel-map del*: *rel-funI intro*!: *rel-spmf-reflI generat.rel-refl-strong rel-funI*)

lemma *exec-gpv-plus-oracle-left*: *exec-gpv* (*plus-oracle oracle1 oracle2*) (*left-gpv gpv*) *s* = *exec-gpv oracle1 gpv s*
unfolding *spmf-rel-eq*[*symmetric*] *prod.rel-eq*[*symmetric*]
by(*rule exec-gpv-parametric'*[**where** *A*=(=) **and** *S*=(=) **and** *CALL*=*λl r. l* = *Inl r* **and** *R*=*λl r. l* = *Inl r*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
(*auto intro*!: *rel-funI simp add*: *spmf-rel-map apfst-def map-prod-def rel-prod-conv intro*: *rel-spmf-reflI left-gpv-Inl-transfer*)

lemma *exec-gpv-plus-oracle-right*: *exec-gpv* (*plus-oracle oracle1 oracle2*) (*right-gpv gpv*) *s* = *exec-gpv oracle2 gpv s*
unfolding *spmf-rel-eq*[*symmetric*] *prod.rel-eq*[*symmetric*]
by(*rule exec-gpv-parametric'*[**where** *A*=(=) **and** *S*=(=) **and** *CALL*=*λl r. l* = *Inr r* **and** *R*=*λl r. l* = *Inr r*, *THEN rel-funD*, *THEN rel-funD*, *THEN rel-funD*])
(*auto intro*!: *rel-funI simp add*: *spmf-rel-map apfst-def map-prod-def rel-prod-conv intro*: *rel-spmf-reflI right-gpv-Inr-transfer*)

lemma *left-gpv-bind-gpv*: *left-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*left-gpv gpv*) (*left-gpv* ∘ *f*)
by(*coinduction arbitrary*:*gpv f rule*: *gpv.coinduct-strong*)
(*auto 4 4 simp add*: *bind-map-spmf spmf-rel-map intro*!: *rel-spmf-reflI rel-spmf-bindI*[*of* (=)] *generat.rel-refl rel-funI split*: *sum.splits*)

lemma *inline1-left-gpv*:
inline1 (*λs q. left-gpv* (*callee s q*)) *gpv s* =
map-spmf (*map-sum id* (*map-prod Inl* (*map-prod left-rpv id*))) (*inline1 callee gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf inline1.mono inline1.mono inline1-def inline1-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step inline1' inline1''*)
then show *?case*
by(*auto simp add*: *map-spmf-bind-spmf o-def bind-map-spmf intro*!: *ext bind-spmf-cong split*: *generat.split*)

qed

lemma *left-gpv-inline*: *left-gpv* (*inline callee gpv s*) = *inline* (λ*s q*. *left-gpv* (*callee s q*)) *gpv s*
  by(*coinduction arbitrary*: *callee gpv s rule*: *gpv-coinduct-bind*)
    (*fastforce simp add*: *inline-sel spmf-rel-map inline1-left-gpv left-gpv-bind-gpv o-def split-def intro*!: *rel-spmf-reflI split*: *sum.split intro*!: *rel-funI gpv.rel-refl-strong*)

lemma *right-gpv-bind-gpv*: *right-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*right-gpv gpv*) (*right-gpv* ∘ *f*)
  by(*coinduction arbitrary*:*gpv f rule*: *gpv.coinduct-strong*)
   (*auto 4 4 simp add*: *bind-map-spmf spmf-rel-map intro*!: *rel-spmf-reflI rel-spmf-bindI*[*of* (=)] *generat.rel-refl rel-funI split*: *sum.splits*)

lemma *inline1-right-gpv*:
  inline1 (λ*s q*. *right-gpv* (*callee s q*)) *gpv s* =
   map-spmf (*map-sum id* (*map-prod Inr* (*map-prod right-rpv id*))) (*inline1 callee gpv s*)
proof(*induction arbitrary*: *gpv s rule*: *parallel-fixp-induct-2-2*[*OF partial-function-definitions-spmf partial-function-definitions-spmf inline1.mono inline1.mono inline1-def inline1-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
  case *adm* **show** *?case* **by** *simp*
  case *bottom* **show** *?case* **by** *simp*
  case (*step inline1′ inline1″*)
  then show *?case*
   by(*auto simp add*: *map-spmf-bind-spmf o-def bind-map-spmf intro*!: *ext bind-spmf-cong split*: *generat.split*)
qed

lemma *right-gpv-inline*: *right-gpv* (*inline callee gpv s*) = *inline* (λ*s q*. *right-gpv* (*callee s q*)) *gpv s*
  by(*coinduction arbitrary*: *callee gpv s rule*: *gpv-coinduct-bind*)
    (*fastforce simp add*: *inline-sel spmf-rel-map inline1-right-gpv right-gpv-bind-gpv o-def split-def intro*!: *rel-spmf-reflI split*: *sum.split intro*!: *rel-funI gpv.rel-refl-strong*)

lemma *WT-gpv-left-gpv*: $\mathcal{I}1$ ⊢*g gpv* √ ⟹ $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$ ⊢*g left-gpv gpv* √
  by(*coinduction arbitrary*: *gpv*)(*auto 4 4 dest*: *WT-gpvD*)

lemma *WT-gpv-right-gpv*: $\mathcal{I}2$ ⊢*g gpv* √ ⟹ $\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$ ⊢*g right-gpv gpv* √
  by(*coinduction arbitrary*: *gpv*)(*auto 4 4 dest*: *WT-gpvD*)

lemma *results-gpv-left-gpv* [*simp*]: *results-gpv* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) (*left-gpv gpv*) = *results-gpv* $\mathcal{I}1$ *gpv*
  (**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
  show *x* ∈ *?rhs* **if** *x* ∈ *?lhs* **for** *x* **using** *that*
   by(*induction gpv′*≡*left-gpv gpv* :: (′*a*, ′*b* + ′*c*, ′*d* + ′*e*) *gpv arbitrary*: *gpv rule*: *results-gpv.induct*)
     (*fastforce intro*: *results-gpv.intros*)+

show $x \in$ *?lhs* **if** $x \in$ *?rhs* **for** x **using** *that*
by(*induction*)(*fastforce intro*: *results-gpv.intros*)+
qed

lemma *results-gpv-right-gpv* [*simp*]: *results-gpv* ($\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2$) (*right-gpv gpv*) = *results-gpv* $\mathcal{I}2$ *gpv*
(**is** *?lhs* = *?rhs*)
proof(*rule Set.set-eqI iffI*)+
show $x \in$ *?rhs* **if** $x \in$ *?lhs* **for** x **using** *that*
by(*induction gpv′≡right-gpv gpv* :: (′*a*, ′*b* + ′*c*, ′*d* + ′*e*) *gpv arbitrary*: *gpv rule*: *results-gpv.induct*)
(*fastforce intro*: *results-gpv.intros*)+
show $x \in$ *?lhs* **if** $x \in$ *?rhs* **for** x **using** *that*
by(*induction*)(*fastforce intro*: *results-gpv.intros*)+
qed

lemma *left-gpv-Fail* [*simp*]: *left-gpv Fail* = *Fail*
by(*rule gpv.expand*) *auto*

lemma *right-gpv-Fail* [*simp*]: *right-gpv Fail* = *Fail*
by(*rule gpv.expand*) *auto*

lemma *rsuml-lsumr-left-gpv-left-gpv*:*map-gpv′ id rsuml lsumr* (*left-gpv* (*left-gpv gpv*)) = *left-gpv gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *lsumr.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *rsuml-lsumr-left-gpv-right-gpv*: *map-gpv′ id rsuml lsumr* (*left-gpv* (*right-gpv gpv*)) = *right-gpv* (*left-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *lsumr.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *rsuml-lsumr-right-gpv*: *map-gpv′ id rsuml lsumr* (*right-gpv gpv*) = *right-gpv* (*right-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *lsumr.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *map-gpv′-map-gpv-swap*:
map-gpv′ f g h (*map-gpv f′ id gpv*) = *map-gpv* ($f \circ f'$) *id* (*map-gpv′ id g h gpv*)
by(*simp add*: *map-gpv-conv-map-gpv′ map-gpv′-comp*)

lemma *lsumr-rsuml-left-gpv*: *map-gpv′ id lsumr rsuml* (*left-gpv gpv*) = *left-gpv* (*left-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split intro*: *exI*[**where** *x*=*Fail*])

lemma *lsumr-rsuml-right-gpv-left-gpv*:
map-gpv′ id lsumr rsuml (*right-gpv* (*left-gpv gpv*)) = *left-gpv* (*right-gpv gpv*)
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split intro*: *exI*[**where** *x*=*Fail*])

lemma *lsumr-rsuml-right-gpv-right-gpv*:
map-gpv′ id lsumr rsuml (*right-gpv* (*right-gpv gpv*)) = *right-gpv gpv*
by(*coinduction arbitrary*: *gpv*)
(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI rel-generat-reflI rel-funI split*!: *sum.split elim*!: *rsuml.elims intro*: *exI*[**where** *x*=*Fail*])

lemma *in-set-spmf-extend-state-oracle* [*simp*]:
$x \in$ *set-spmf* (*extend-state-oracle oracle s y*) $\longleftrightarrow$
fst (*snd x*) = *fst s* $\wedge$ (*fst x*, *snd* (*snd x*)) $\in$ *set-spmf* (*oracle* (*snd s*) *y*)
by(*auto 4 4 simp add*: *extend-state-oracle-def split-beta intro*: *rev-image-eqI prod.expand*)

lemma *extend-state-oracle-plus-oracle*:
extend-state-oracle (*plus-oracle oracle1 oracle2*) = *plus-oracle* (*extend-state-oracle oracle1*) (*extend-state-oracle oracle2*)
proof ((*rule ext*)+; *goal-cases*)
case (*1 s q*)
then show *?case* **by** (*cases s*; *cases q*) (*simp-all add*: *apfst-def spmf.map-comp o-def split-def*)
qed

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

lemma *stateless-callee-parametric′*:
includes *lifting-syntax* **notes** [*transfer-rule*] = *map-gpv-parametric′* **shows**
((*A* ===> *rel-gpv′′ B C R*) ===> *S* ===> *A* ===> (*rel-gpv′′* (*rel-prod B S*) *C R*))
stateless-callee stateless-callee
unfolding *stateless-callee-def* **by** *transfer-prover*

lemma *id-oralce-alt-def*: *id-oracle* = *stateless-callee* (λ*x*. *Pause x Done*)
by(*simp add*: *id-oracle-def fun-eq-iff stateless-callee-def*)

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

fun *parallel-intercept* :: ′*s1* × ′*s2* ⇒ ′*x1* + ′*x2* ⇒ ((′*y1* + ′*y2*) × (′*s1* × ′*s2*),

'call1 + 'call2, 'ret1 + 'ret2) gpv
  where
  parallel-intercept (s1, s2) (Inl a) = left-gpv (map-gpv (map-prod Inl (λs1'. (s1', s2))) id (left s1 a))
  | parallel-intercept (s1, s2) (Inr b) = right-gpv (map-gpv (map-prod Inr (Pair s1)) id (right s2 b))

end

end

6.4 Expectation transformer semantics

theory *GPV-Expectation* **imports**
  Computational-Model
begin

lemma *le-enn2realI*: ⟦ *ennreal x* ≤ *y*; *y* = ⊤ ⟹ *x* ≤ *0* ⟧ ⟹ *x* ≤ *enn2real y*
by(*cases y*) *simp-all*

lemma *enn2real-leD*: ⟦ *enn2real x* < *y*; *x* ≠ ⊤ ⟧ ⟹ *x* < *ennreal y*
by(*cases x*)(*simp-all add*: *ennreal-lessI*)

lemma *ennreal-mult-le-self2I*: ⟦ *y* > *0* ⟹ *x* ≤ *1* ⟧ ⟹ *x* ∗ *y* ≤ *y* **for** *x y* :: *ennreal*
apply(*cases x*; *cases y*)
apply(*auto simp add*: *top-unique ennreal-top-mult ennreal-mult*[*symmetric*] *intro*: *ccontr*)
using *mult-left-le-one-le* **by** *force*

lemma *ennreal-leI*: *x* ≤ *enn2real y* ⟹ *ennreal x* ≤ *y*
by(*cases y*) *simp-all*

lemma *enn2real-INF*: ⟦ *A* ≠ {}; ∀ *x*∈*A*. *f x* < ⊤ ⟧ ⟹ *enn2real* (*INF x*∈*A*. *f x*) = (*INF x*∈*A*. *enn2real* (*f x*))
apply(*rule antisym*)
 apply(*rule cINF-greatest*)
  apply *simp*
 apply(*rule enn2real-mono*)
  apply(*erule INF-lower*)
 apply *simp*
apply(*rule le-enn2realI*)
 apply *simp-all*
apply(*rule INF-greatest*)
apply(*rule ennreal-leI*)
apply(*rule cINF-lower*)
apply(*rule bdd-belowI*[**where** *m=0*])
apply *auto*
done

lemma *monotone-times-ennreal1*: *monotone* (≤) (≤) (λ*x*. *x* ∗ *y* :: *ennreal*)
by(*auto intro*!: *monotoneI mult-right-mono*)

lemma *monotone-times-ennreal2*: *monotone* (≤) (≤) (λ*x*. *y* ∗ *x* :: *ennreal*)
by(*auto intro*!: *monotoneI mult-left-mono*)

lemma *mono2mono-times-ennreal*[*THEN lfp.mono2mono2*, *cont-intro*, *simp*]:
  shows *monotone-times-ennreal*: *monotone* (*rel-prod* (≤) (≤)) (≤) (λ(*x*, *y*). *x* ∗ *y* :: *ennreal*)
by(*simp add*: *monotone-times-ennreal1 monotone-times-ennreal2*)

lemma *mcont-times-ennreal1*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *x* ∗ *y* :: *ennreal*)
by(*auto intro*!: *mcontI contI simp add*: *SUP-mult-left-ennreal*[*symmetric*])

lemma *mcont-times-ennreal2*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *y* ∗ *x* :: *ennreal*)
by(*subst mult.commute*)(*rule mcont-times-ennreal1*)

lemma *mcont2mcont-times-ennreal* [*cont-intro*, *simp*]:
  ⟦ *mcont lub ord Sup* (≤) (λ*x*. *f x*);
    mcont lub ord Sup (≤) (λ*x*. *g x*) ⟧
  ⟹ *mcont lub ord Sup* (≤) (λ*x*. *f x* ∗ *g x* :: *ennreal*)
by(*best intro*: *ccpo.mcont2mcont'*[*OF complete-lattice-ccpo*] *mcont-times-ennreal1 mcont-times-ennreal2 ccpo.mcont-const*[*OF complete-lattice-ccpo*])

lemma *ereal-INF-cmult*: *0* < *c* ⟹ (*INF i*∈*I*. *c* ∗ *f i*) = *ereal c* ∗ (*INF i*∈*I*. *f i*)
using *ereal-Inf-cmult*[**where** *P*=λ*x*. ∃ *i*∈*I*. *x* = *f i*, *of c*]
by(*rule box-equals*)(*auto intro*!: *arg-cong*[**where** *f*=*Inf*] *arg-cong2*[**where** *f*=(∗)])

lemma *ereal-INF-multc*: *0* < *c* ⟹ (*INF i*∈*I*. *f i* ∗ *c*) = (*INF i*∈*I*. *f i*) ∗ *ereal c*
using *ereal-INF-cmult*[*of c f I*] **by**(*simp add*: *mult.commute*)

lemma *INF-mult-left-ennreal*:
  assumes *I* = {} ⟹ *c* ≠ *0*
  and ⟦ *c* = ⊤; ∃ *i*∈*I*. *f i* > *0* ⟧ ⟹ ∃ *p*>*0*. ∀ *i*∈*I*. *f i* ≥ *p*
  shows *c* ∗ (*INF i*∈*I*. *f i*) = (*INF i*∈*I*. *c* ∗ *f i* ::*ennreal*)
proof −
  consider (*empty*) *I* = {} | (*top*) *c* = ⊤ | (*zero*) *c* = *0* | (*normal*) *I* ≠ {} *c* ≠ ⊤ *c* ≠ *0* **by** *auto*
  then show *?thesis*
  proof *cases*
    case *empty* **then show** *?thesis* **by**(*simp add*: *ennreal-mult-top assms*(*1*))
  next
    case *top*
    show *?thesis*
    proof(*cases* ∃ *i*∈*I*. *f i* > *0*)
      case *True*
      with *assms*(*2*) *top* **obtain** *p* **where** *p* > *0* **and** *p*: ⋀*i*. *i* ∈ *I* ⟹ *f i* ≥ *p* **by** *auto*
      then have ∗: ⋀*i*. *i* ∈ *I* ⟹ *f i* > *0* **by**(*auto intro*: *less-le-trans*)

note ‹*0* < *p*› **also from** *p* **have** *p* ≤ (*INF i*∈*I*. *f i*) **by**(*rule INF-greatest*)
finally show *?thesis* **using** *top* **by**(*auto simp add*: *ennreal-top-mult dest*: ∗)
next
case *False*
hence *f i* = *0* **if** *i* ∈ *I* **for** *i* **using** *that* **by** *auto*
thus *?thesis* **using** *top* **by**(*simp add*: *INF-constant ennreal-mult-top*)
qed
next
case *zero*
then show *?thesis* **using** *assms*(*1*) **by**(*auto simp add*: *INF-constant*)
next
case *normal*
then show *?thesis* **including** *ennreal.lifting*
apply *transfer*
subgoal for *I c f* **by**(*cases c*)(*simp-all add*: *top-ereal-def ereal-INF-cmult*)
done
qed
qed

lemma *pmf-map-spmf-None*: *pmf* (*map-spmf f p*) *None* = *pmf p None*
by(*simp add*: *pmf-None-eq-weight-spmf*)

lemma *nn-integral-try-spmf*:
nn-integral (*measure-spmf* (*try-spmf p q*)) *f* = *nn-integral* (*measure-spmf p*) *f* + *nn-integral* (*measure-spmf q*) *f* ∗ *pmf p None*
by(*simp add*: *nn-integral-measure-spmf spmf-try-spmf distrib-right nn-integral-add ennreal-mult mult.assoc nn-integral-cmult*)
(*simp add*: *mult.commute*)

lemma *INF-UNION*: (*INF z* ∈ ⋃*x*∈*A*. *B x*. *f z*) = (*INF x*∈*A*. *INF z*∈*B x*. *f z*)
for *f* :: - ⇒ ′*b*::*complete-lattice*
by(*auto intro*!: *antisym INF-greatest intro*: *INF-lower2*)

definition *nn-integral-spmf* :: ′*a spmf* ⇒ (′*a* ⇒ *ennreal*) ⇒ *ennreal* **where**
nn-integral-spmf p = *nn-integral* (*measure-spmf p*)

lemma *nn-integral-spmf-parametric* [*transfer-rule*]:
includes *lifting-syntax*
shows (*rel-spmf A* ===> (*A* ===> (=)) ===> (=)) *nn-integral-spmf nn-integral-spmf*
unfolding *nn-integral-spmf-def*
proof(*rule rel-funI*)+
fix *p q* **and** *f g* :: - ⇒ *ennreal*
assume *pq*: *rel-spmf A p q* **and** *fg*: (*A* ===> (=)) *f g*
from *pq* **obtain** *pq* **where** *pq* [*rule-format*]: ∀ (*x*, *y*)∈*set-spmf pq*. *A x y*
and *p*: *p* = *map-spmf fst pq* **and** *q*: *q* = *map-spmf snd pq*
by(*cases rule*: *rel-spmfE*) *auto*
show *nn-integral* (*measure-spmf p*) *f* = *nn-integral* (*measure-spmf q*) *g*
by(*simp add*: *p q*)(*auto simp add*: *nn-integral-measure-spmf spmf-eq-0-set-spmf*

dest!: *pq rel-funD*[*OF fg*] *intro*: *ennreal-mult-left-cong intro*!: *nn-integral-cong*)
qed

lemma *weight-spmf-mcont2mcont* [*THEN lfp.mcont2mcont*, *cont-intro*]:
shows *weight-spmf-mcont*: *mcont* (*lub-spmf*) (*ord-spmf* (=)) *Sup* (≤) (λ*p*. *ennreal* (*weight-spmf p*))
apply(*simp add*: *mcont-def cont-def weight-spmf-def measure-spmf.emeasure-eq-measure*[*symmetric*] *emeasure-lub-spmf*)
apply(*rule call-mono*[*THEN lfp.mono2mono*])
apply(*unfold fun-ord-def*)
apply(*rule monotone-emeasure-spmf*[*unfolded le-fun-def*])
done

lemma *mono2mono-nn-integral-spmf* [*THEN lfp.mono2mono*, *cont-intro*]:
shows *monotone-nn-integral-spmf*: *monotone* (*ord-spmf* (=)) (≤) (λ*p*. $integral^N$ (*measure-spmf p*) *f*)
by(*rule monotoneI*)(*auto simp add*: *nn-integral-measure-spmf intro*!: *nn-integral-mono mult-right-mono dest*: *monotone-spmf*[*THEN monotoneD*])

lemma *cont-nn-integral-spmf*:
cont lub-spmf (*ord-spmf* (=)) *Sup* (≤) (λ*p* :: ′*a spmf*. *nn-integral* (*measure-spmf p*) *f*)
proof
fix *Y* :: ′*a spmf set*
assume *Y*: *Complete-Partial-Order.chain* (*ord-spmf* (=)) *Y Y* ≠ {}
let *?M* = *count-space* (*set-spmf* (*lub-spmf Y*))
have *nn-integral* (*measure-spmf* (*lub-spmf Y*)) *f* = ∫⁺ *x*. *ennreal* (*spmf* (*lub-spmf Y*) *x*) ∗ *f x* ∂*?M*
by(*simp add*: *nn-integral-measure-spmf* ′)
also have ... = ∫⁺ *x*. (*SUP p*∈*Y*. *ennreal* (*spmf p x*) ∗ *f x*) ∂*?M*
by(*simp add*: *spmf-lub-spmf Y ennreal-SUP*[*OF SUP-spmf-neq-top*′] *SUP-mult-right-ennreal*)
also have ... = (*SUP p*∈*Y*. ∫⁺ *x*. *ennreal* (*spmf p x*) ∗ *f x* ∂*?M*)
proof(*rule nn-integral-monotone-convergence-SUP-countable*)
show *Complete-Partial-Order.chain* (≤) ((λ*i x*. *ennreal* (*spmf i x*) ∗ *f x*) ‘ *Y*)
using *Y*(*1*) **by**(*rule chain-imageI*)(*auto simp add*: *le-fun-def intro*!: *mult-right-mono dest*: *monotone-spmf*[*THEN monotoneD*])
qed(*simp-all add*: *Y*(*2*))
also have ... = (*SUP p*∈*Y*. *nn-integral* (*measure-spmf p*) *f*)
by(*auto simp add*: *nn-integral-measure-spmf Y nn-integral-count-space-indicator set-lub-spmf spmf-eq-0-set-spmf split*: *split-indicator intro*!: *SUP-cong nn-integral-cong*)
finally show *nn-integral* (*measure-spmf* (*lub-spmf Y*)) *f* = (*SUP p*∈*Y*. *nn-integral* (*measure-spmf p*) *f*) **.**
qed

lemma *mcont2mcont-nn-integral-spmf* [*THEN lfp.mcont2mcont*, *cont-intro*]:
shows *mcont-nn-integral-spmf*:
mcont lub-spmf (*ord-spmf* (=)) *Sup* (≤) (λ*p* :: ′*a spmf*. *nn-integral* (*measure-spmf p*) *f*)
by(*rule mcontI*)(*simp-all add*: *cont-nn-integral-spmf*)

lemma *nn-integral-mono2mono*:
  assumes $\bigwedge x.\ x \in space\ M \Longrightarrow monotone\ ord\ (\leq)\ (\lambda f.\ F\ f\ x)$
  shows $monotone\ ord\ (\leq)\ (\lambda f.\ nn\text{-}integral\ M\ (F\ f))$
  by(*rule monotoneI nn-integral-mono monotoneD*[*OF assms*])+

lemma *nn-integral-mono-lfp* [*partial-function-mono*]:
  — `Partial_Function.mono_tac` does not like conditional assumptions (more precisely the case splitter)
  $(\bigwedge x.\ lfp.mono\text{-}body\ (\lambda f.\ F\ f\ x)) \Longrightarrow lfp.mono\text{-}body\ (\lambda f.\ nn\text{-}integral\ M\ (F\ f))$
  by(*rule nn-integral-mono2mono*)

lemma *INF-mono-lfp* [*partial-function-mono*]:
  $(\bigwedge x.\ lfp.mono\text{-}body\ (\lambda f.\ F\ f\ x)) \Longrightarrow lfp.mono\text{-}body\ (\lambda f.\ INF\ x{\in}M.\ F\ f\ x)$
  by(*rule monotoneI*)(*blast dest*: *monotoneD intro*: *INF-mono*)

lemmas *parallel-fixp-induct-1-2* = *parallel-fixp-induct-uc*[
  of - - - - $\lambda x.\ x$ - $\lambda x.\ x$ *case-prod* - *curry*,
  where $P{=}\lambda f\ g.\ P\ f\ (curry\ g)$,
  unfolded case-prod-curry curry-case-prod curry-K,
  OF - - - - - - *refl refl*]
  for *P*

lemma *monotone-ennreal-add1*: $monotone\ (\leq)\ (\leq)\ (\lambda x.\ x + y :: ennreal)$
by(*auto intro*!: *monotoneI*)

lemma *monotone-ennreal-add2*: $monotone\ (\leq)\ (\leq)\ (\lambda y.\ x + y :: ennreal)$
by(*auto intro*!: *monotoneI*)

lemma *mono2mono-ennreal-add*[*THEN lfp.mono2mono2*, *cont-intro*, *simp*]:
  shows *monotone-eadd*: $monotone\ (rel\text{-}prod\ (\leq)\ (\leq))\ (\leq)\ (\lambda(x,\ y).\ x + y :: ennreal)$
by(*simp add*: *monotone-ennreal-add1 monotone-ennreal-add2*)

lemma *ennreal-add-partial-function-mono* [*partial-function-mono*]:
  $[\![\ monotone\ (fun\text{-}ord\ (\leq))\ (\leq)\ f;\ monotone\ (fun\text{-}ord\ (\leq))\ (\leq)\ g\ ]\!]$
  $\Longrightarrow monotone\ (fun\text{-}ord\ (\leq))\ (\leq)\ (\lambda x.\ f\ x + g\ x :: ennreal)$
by(*rule mono2mono-ennreal-add*)

context
  fixes *fail* :: *ennreal*
  and $\mathcal{I}$ :: ('*out*, '*ret*) $\mathcal{I}$
  and *f* :: '*a* ⇒ *ennreal*
  notes [[*function-internals*]]
begin

partial-function (*lfp-strong*) *expectation-gpv* :: ('*a*, '*out*, '*ret*) *gpv* ⇒ *ennreal* **where**
  expectation-gpv gpv =

(∫ + generat. (case generat of Pure x ⇒ f x
| IO out c ⇒ INF r∈responses-ℐ ℐ out. expectation-gpv (c r))
∂measure-spmf (the-gpv gpv))
+ fail ∗ pmf (the-gpv gpv) None

lemma *expectation-gpv-fixp-induct* [*case-names adm bottom step*]:
assumes *lfp.admissible P*
and *P* (λ-. *0*)
and ⋀*expectation-gpv′*. ⟦ ⋀*gpv*. *expectation-gpv′ gpv* ≤ *expectation-gpv gpv*; *P expectation-gpv′* ⟧ ⟹
P (λ*gpv*. (∫ + *generat*. (*case generat of Pure x* ⇒ *f x* | *IO out c* ⇒ *INF r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ∂*measure-spmf* (*the-gpv gpv*)) + *fail* ∗ *pmf* (*the-gpv gpv*) *None*)
shows *P expectation-gpv*
by(*rule expectation-gpv.fixp-induct*)(*simp-all add*: *bot-ennreal-def assms fun-ord-def*)

lemma *expectation-gpv-Done* [*simp*]: *expectation-gpv* (*Done x*) = *f x*
by(*subst expectation-gpv.simps*)(*simp add*: *measure-spmf-return-spmf nn-integral-return*)

lemma *expectation-gpv-Fail* [*simp*]: *expectation-gpv Fail* = *fail*
by(*subst expectation-gpv.simps*) *simp*

lemma *expectation-gpv-lift-spmf* [*simp*]:
expectation-gpv (*lift-spmf p*) = (∫ + *x*. *f x* ∂*measure-spmf p*) + *fail* ∗ *pmf p None*
by(*subst expectation-gpv.simps*)(*auto simp add*: *o-def pmf-map vimage-def measure-pmf-single*)

lemma *expectation-gpv-Pause* [*simp*]:
expectation-gpv (*Pause out c*) = (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv* (*c r*))
by(*subst expectation-gpv.simps*)(*simp add*: *measure-spmf-return-spmf nn-integral-return*)

end

context begin
private definition *weight-spmf′ p* = *weight-spmf p*
lemmas *weight-spmf′-parametric* = *weight-spmf-parametric*[*folded weight-spmf′-def*]
lemma *expectation-gpv-parametric′*:
includes *lifting-syntax* **notes** *weight-spmf′-parametric*[*transfer-rule*]
shows ((=) ===> *rel-ℐ C R* ===> (*A* ===> (=)) ===> *rel-gpv′′ A C R* ===> (=)) *expectation-gpv expectation-gpv*
unfolding *expectation-gpv-def*
apply(*rule rel-funI*)
apply(*rule rel-funI*)
apply(*rule rel-funI*)
apply(*rule fixp-lfp-parametric-eq*[*OF expectation-gpv.mono expectation-gpv.mono*])
apply(*fold nn-integral-spmf-def Set.is-empty-def pmf-None-eq-weight-spmf*[*symmetric*])
apply(*simp only*: *weight-spmf′-def*[*symmetric*])
subgoal premises [*transfer-rule*] **supply** *the-gpv-parametric′*[*transfer-rule*] **by**

transfer-prover
done
end

lemma *expectation-gpv-parametric* [*transfer-rule*]:
includes *lifting-syntax*
shows ((=) ===> *rel-*$\mathcal{I}$ *C* (=) ===> (*A* ===> (=)) ===> *rel-gpv A C* ===> (=)) *expectation-gpv expectation-gpv*
using *expectation-gpv-parametric*′[*of C* (=) *A*] **by**(*simp add*: *rel-gpv-conv-rel-gpv*′′)

lemma *expectation-gpv-cong*:
fixes *fail fail*′
assumes *fail*: *fail* = *fail*′
and $\mathcal{I}$: $\mathcal{I} = \mathcal{I}'$
and *gpv*: *gpv* = *gpv*′
and *f*: $\bigwedge x.\ x \in$ *results-gpv* $\mathcal{I}'$ *gpv*′ $\Longrightarrow$ *f x* = *g x*
shows *expectation-gpv fail* $\mathcal{I}$ *f gpv* = *expectation-gpv fail*′ $\mathcal{I}'$ *g gpv*′
using *f* **unfolding** $\mathcal{I}$[*symmetric*] *gpv*[*symmetric*] *fail*[*symmetric*]
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv*′ *expectation-gpv*′′) **show** *?case*
by(*rule arg-cong2*[**where** *f*=(+)] *nn-integral-cong-AE*)+(*clarsimp simp add*: *step.prems results-gpv.intros split*!: *generat.split intro*!: *INF-cong*[*OF refl*] *step.IH*)+
qed

lemma *expectation-gpv-cong-fail*:
colossless-gpv $\mathcal{I}$ *gpv* $\Longrightarrow$ *expectation-gpv fail* $\mathcal{I}$ *f gpv* = *expectation-gpv fail*′ $\mathcal{I}$ *f gpv* **for** *fail*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv*′ *expectation-gpv*′′)
from *colossless-gpv-lossless-spmfD*[*OF step.prems*] **show** *?case*
by(*auto simp add*: *lossless-iff-pmf-None intro*!: *nn-integral-cong-AE INF-cong step.IH intro*: *colossless-gpv-continuationD*[*OF step.prems*] *split*: *generat.split*)
qed

lemma *expectation-gpv-mono*:
fixes *fail fail*′
assumes *fail*: *fail* ≤ *fail*′
and *fg*: *f* ≤ *g*
shows *expectation-gpv fail* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv fail*′ $\mathcal{I}$ *g gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono*

expectation-gpv-def expectation-gpv-def, *case-names adm bottom step*])
  case *adm* **show** *?case* **by** *simp*
  case *bottom* **show** *?case* **by** *simp*
  case (*step expectation-gpv′ expectation-gpv″*)
  show *?case*
    by(*intro add-mono mult-right-mono fail nn-integral-mono-AE*)
      (*auto split*: *generat.split simp add*: *fg*[*THEN le-funD*] *INF-mono rev-bexI step.IH*)
qed

lemma *expectation-gpv-mono-strong*:
  fixes *fail fail′*
  assumes *fail*: $\neg$ *colossless-gpv* $\mathcal{I}$ *gpv* $\Longrightarrow$ *fail* $\leq$ *fail′*
  and *fg*: $\bigwedge x.$ $x \in$ *results-gpv* $\mathcal{I}$ *gpv* $\Longrightarrow$ *f x* $\leq$ *g x*
  shows *expectation-gpv fail* $\mathcal{I}$ *f gpv* $\leq$ *expectation-gpv fail′* $\mathcal{I}$ *g gpv*
proof −
  let *?fail* = *if colossless-gpv* $\mathcal{I}$ *gpv then fail′ else fail*
    and *?f* = $\lambda x.$ *if* $x \in$ *results-gpv* $\mathcal{I}$ *gpv then f x else g x*
  have *expectation-gpv fail* $\mathcal{I}$ *f gpv* = *expectation-gpv ?fail* $\mathcal{I}$ *f gpv* **by**(*simp cong*: *expectation-gpv-cong-fail*)
  also have ... = *expectation-gpv ?fail* $\mathcal{I}$ *?f gpv* **by**(*rule expectation-gpv-cong*; *simp*)
  also have ... $\leq$ *expectation-gpv fail′* $\mathcal{I}$ *g gpv* **using** *assms* **by**(*simp add*: *expectation-gpv-mono le-fun-def*)
  finally show *?thesis* **.**
qed

lemma *expectation-gpv-bind* [*simp*]:
  fixes $\mathcal{I}$ *f g fail*
  defines *expectation-gpv1* $\equiv$ *expectation-gpv fail* $\mathcal{I}$ *f*
  and *expectation-gpv2* $\equiv$ *expectation-gpv fail* $\mathcal{I}$ (*expectation-gpv fail* $\mathcal{I}$ *f* $\circ$ *g*)
  shows *expectation-gpv1* (*bind-gpv gpv g*) = *expectation-gpv2 gpv* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  note [*simp*] = *case-map-generat o-def*
    and [*cong del*] = *generat.case-cong-weak*
  show *?lhs* $\leq$ *?rhs* **unfolding** *expectation-gpv1-def*
  proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
    case *adm* **show** *?case* **by** *simp*
    case *bottom* **show** *?case* **by** *simp*
    case (*step expectation-gpv′*)
    show *?case* **unfolding** *expectation-gpv2-def*
      apply(*rewrite bind-gpv.sel*)
      apply(*simp add*: *map-spmf-bind-spmf measure-spmf-bind*)
      apply(*rewrite nn-integral-bind*[**where** *B*=*measure-spmf* -])
        apply(*simp-all add*: *space-subprob-algebra*)
      apply(*rewrite expectation-gpv.simps*)
    apply(*simp add*: *pmf-bind-spmf-None distrib-left nn-integral-eq-integral*[*symmetric*] *measure-spmf.integrable-const-bound*[**where** *B*=*1*] *pmf-le-1 nn-integral-cmult*[*symmetric*] *nn-integral-add*[*symmetric*])

apply(*rule disjI2*)
apply(*rule nn-integral-mono*)
apply(*clarsimp split*!: *generat.split*)
apply(*rewrite expectation-gpv.simps*)
apply *simp*
apply(*rule disjI2*)
apply(*rule nn-integral-mono*)
apply(*clarsimp split*: *generat.split*)
apply(*rule INF-mono*)
apply(*erule rev-bexI*)
apply(*rule step.hyps*)
apply(*clarsimp simp add*: *measure-spmf-return-spmf nn-integral-return*)
apply(*rule INF-mono*)
apply(*erule rev-bexI*)
apply(*rule step.IH*[*unfolded expectation-gpv2-def o-def*])
done
qed
show *?rhs* ≤ *?lhs* **unfolding** *expectation-gpv2-def*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv′*)
show *?case* **unfolding** *expectation-gpv1-def*
apply(*rewrite* **in** - ≤ ⌑ *expectation-gpv.simps*)
apply(*rewrite bind-gpv.sel*)
apply(*simp add*: *measure-spmf-bind*)
apply(*rewrite nn-integral-bind*[**where** *B*=*measure-spmf* -])
apply(*simp-all add*: *space-subprob-algebra*)
apply(*simp add*: *pmf-bind-spmf-None distrib-left nn-integral-eq-integral*[*symmetric*] *measure-spmf.integrable-const-bound*[**where** *B*=*1*] *pmf-le-1 nn-integral-cmult*[*symmetric*] *nn-integral-add*[*symmetric*])
apply(*rule disjI2*)
apply(*rule nn-integral-mono*)
apply(*clarsimp split*!: *generat.split*)
apply(*rewrite expectation-gpv.simps*)
apply(*simp cong del*: *if-weak-cong add*: *generat.map-comp id-def*[*symmetric*] *generat.map-id*)
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return*)
apply(*rule INF-mono*)
apply(*erule rev-bexI*)
apply(*rule step.IH*[*unfolded expectation-gpv1-def*])
done
qed
qed

lemma *expectation-gpv-try-gpv* [*simp*]:
fixes *fail* $\mathcal{I}$ *f gpv′*
defines *expectation-gpv1* ≡ *expectation-gpv fail* $\mathcal{I}$ *f*
and *expectation-gpv2* ≡ *expectation-gpv* (*expectation-gpv fail* $\mathcal{I}$ *f gpv′*) $\mathcal{I}$ *f*

shows *expectation-gpv1* (*try-gpv gpv gpv′*) = *expectation-gpv2 gpv*
proof(*rule antisym*)
show *expectation-gpv1* (*try-gpv gpv gpv′*) ≤ *expectation-gpv2 gpv* **unfolding** *expectation-gpv1-def*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case *step* [*unfolded expectation-gpv2-def*]: (*step expectation-gpv′*)
show *?case* **unfolding** *expectation-gpv2-def*
apply(*rewrite expectation-gpv.simps*)
apply(*rewrite* **in** - ≤ - + ¤ *expectation-gpv.simps*)
apply(*simp add*: *pmf-map-spmf-None nn-integral-try-spmf o-def generat.map-comp case-map-generat distrib-right cong del*: *generat.case-cong-weak*)
apply(*simp add*: *mult-ac add.assoc ennreal-mult*)
apply(*intro disjI2 add-mono mult-left-mono nn-integral-mono*; *clarsimp split*: *generat.split intro*!: *INF-mono step elim*!: *rev-bexI*)
done
qed
show *expectation-gpv2 gpv* ≤ *expectation-gpv1* (*try-gpv gpv gpv′*) **unfolding** *expectation-gpv2-def*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case *step* [*unfolded expectation-gpv1-def*]: (*step expectation-gpv′*)
show *?case* **unfolding** *expectation-gpv1-def*
apply(*rewrite* **in** - ≤ ¤ *expectation-gpv.simps*)
apply(*rewrite* **in** ¤ ≤ - *expectation-gpv.simps*)
apply(*simp add*: *pmf-map-spmf-None nn-integral-try-spmf o-def generat.map-comp case-map-generat distrib-left ennreal-mult mult-ac id-def*[*symmetric*] *generat.map-id cong del*: *generat.case-cong-weak*)
apply(*rule disjI2 nn-integral-mono*)+
apply(*clarsimp split*: *generat.split intro*!: *INF-mono step*(*2*) *elim*!: *rev-bexI*)
done
qed
qed

lemma *expectation-gpv-restrict-gpv*:
$\mathcal{I}$ ⊢*g gpv* √ ⟹ *expectation-gpv fail* $\mathcal{I}$ *f* (*restrict-gpv* $\mathcal{I}$ *gpv*) = *expectation-gpv fail* $\mathcal{I}$ *f gpv* **for** *fail*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv″*)
show *?case*
apply(*simp add*: *pmf-map vimage-def*)
apply(*rule arg-cong2*[**where** *f*=(+)])
subgoal by(*clarsimp simp add*: *measure-spmf-def nn-integral-distr nn-integral-restrict-space step.IH WT-gpv-ContD*[*OF step.prems*] *AE-measure-pmf-iff in-set-spmf*[*symmetric*] *WT-gpv-OutD*[*OF step.prems*] *split*!: *option.split generat.split intro*!: *nn-integral-cong-AE*

INF-cong[*OF refl*])
apply(*simp add*: *measure-pmf-single*[*symmetric*])
apply(*rule arg-cong*[**where** *f*=*λx. - ∗ ennreal x*])
apply(*rule measure-pmf.finite-measure-eq-AE*)
apply(*auto simp add*: *AE-measure-pmf-iff in-set-spmf*[*symmetric*] *intro*: *WT-gpv-OutD*[*OF step.prems*] *split*: *option.split-asm generat.split-asm if-split-asm*)
done
qed

lemma *expectation-gpv-const-le*: $\mathcal{I} \vdash g$ *gpv* $\surd \Longrightarrow$ *expectation-gpv fail* $\mathcal{I}$ (λ-. *c*) *gpv* ≤ *max c fail* **for** *fail*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv′*)
have $integral^N$ (*measure-spmf* (*the-gpv gpv*)) (*case-generat* (λ*x*. *c*) (λ*out c*. *INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv′* (*c r*))) ≤ $integral^N$ (*measure-spmf* (*the-gpv gpv*)) (λ-. *max c fail*)
using *step.prems*
by(*intro nn-integral-mono-AE*)(*auto 4 4 split*: *generat.split intro*: *INF-lower2 step.IH WT-gpv-ContD*[*OF step.prems*] *dest*!: *WT-gpv-OutD simp add*: *in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$)
also have . . . + *fail* ∗ *pmf* (*the-gpv gpv*) *None* ≤ . . . + *max c fail* ∗ *pmf* (*the-gpv gpv*) *None*
by(*intro add-left-mono mult-right-mono*) *simp-all*
also have . . . ≤ *max c fail*
by(*simp add*: *measure-spmf.emeasure-eq-measure pmf-None-eq-weight-spmf ennreal-minus*[*symmetric*])
(*metis* (*no-types*, *opaque-lifting*) *add-diff-eq-iff-ennreal distrib-left ennreal-le-1 le-max-iff-disj max.cobounded2 mult.commute mult.left-neutral weight-spmf-le-1*)
finally show *?case* **by**(*simp add*: *add-mono*)
qed

lemma *expectation-gpv-no-results*:
⟦ *results-gpv* $\mathcal{I}$ *gpv* = {}; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ $\Longrightarrow$ *expectation-gpv 0* $\mathcal{I}$ *f gpv* = *0*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv′*)
have *results-gpv* $\mathcal{I}$ (*c x*) = {} **if** *IO out c* ∈ *set-spmf* (*the-gpv gpv*) *x* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*
for *out c x* **using** *that step.prems*(*1*) **by**(*auto intro*: *results-gpv.IO*)
then show *?case* **using** *step.prems*
by(*auto 4 4 intro*!: *nn-integral-zero′ split*: *generat.split intro*: *results-gpv.Pure cong*: *INF-cong simp add*: *step.IH WT-gpv-ContD INF-constant in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$ *dest*: *WT-gpv-OutD*)
qed

lemma *expectation-gpv-cmult*:
fixes *fail*

assumes *0* < *c* **and** *c* ≠ ⊤
shows *c* ∗ *expectation-gpv fail* $\mathcal{I}$ *f gpv* = *expectation-gpv* (*c* ∗ *fail*) $\mathcal{I}$ (λ*x*. *c* ∗ *f x*) *gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by**(*simp add*: *bot-ennreal-def*)
case (*step expectation-gpv′ expectation-gpv′′*)
show *?case* **using** *assms*
apply(*simp add*: *distrib-left mult-ac nn-integral-cmult*[*symmetric*] *generat.case-distrib*[**where** *h*=(∗) -]])
apply(*subst INF-mult-left-ennreal*, *simp-all add*: *step.IH*)
done
qed

lemma *expectation-gpv-le-exec-gpv*:
assumes *callee*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-spmf* (*callee s x*)
and *WT-gpv*: $\mathcal{I}$ ⊢*g gpv* √
and *WT-callee*: ⋀*s*. $\mathcal{I}$ ⊢*c callee s* √
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* ≤ ∫ + (*x*, *s*). *f x* ∂*measure-spmf* (*exec-gpv callee gpv s*)
using *WT-gpv*
proof(*induction arbitrary*: *gpv s rule*: *parallel-fixp-induct-1-2*[*OF complete-lattice-partial-function-definitions partial-function-definitions-spmf expectation-gpv.mono exec-gpv.mono expectation-gpv-def exec-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by**(*simp add*: *bot-ennreal-def*)
case (*step expectation-gpv′′ exec-gpv′*)
have ∗: (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv′′* (*c r*)) ≤ ∫ + (*x*, *s*). *f x* ∂*measure-spmf* (*bind-spmf* (*callee s out*) (λ(*r*, *s′*). *exec-gpv′* (*c r*) *s′*)) (**is** *?lhs* ≤ *?rhs*)
if *IO out c* ∈ *set-spmf* (*the-gpv gpv*) **for** *out c*
proof −
from *step.prems that* **have** *out*: *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*rule WT-gpvD*)
have *?lhs* = ∫ + -. *?lhs* ∂*measure-spmf* (*callee s out*) **using** *callee*[*OF out*, *THEN lossless-weight-spmfD*]
by(*simp add*: *measure-spmf.emeasure-eq-measure*)
also have ... ≤ ∫ + (*r*, *s′*). *expectation-gpv′′* (*c r*) ∂*measure-spmf* (*callee s out*)
by(*rule nn-integral-mono-AE*)(*auto intro*: *WT-calleeD*[*OF WT-callee - out*] *INF-lower*)
also have ... ≤ ∫ + (*r*, *s′*). ∫ + (*x*, -). *f x* ∂*measure-spmf* (*exec-gpv′* (*c r*) *s′*) ∂*measure-spmf* (*callee s out*)
by(*rule nn-integral-mono-AE*)(*auto intro*!: *step.IH intro*: *WT-gpv-ContD*[*OF step.prems that*] *WT-calleeD*[*OF WT-callee - out*])
also have ... = *?rhs* **by**(*simp add*: *measure-spmf-bind split-def nn-integral-bind*[**where** *B*=*measure-spmf* -] *o-def space-subprob-algebra*)
finally show *?thesis* **.**

qed
show *?case*
by(*simp add*: *measure-spmf-bind nn-integral-bind*[**where** *B*=*measure-spmf* -] *space-subprob-algebra*)
(*simp split*!: *generat.split add*: *measure-spmf-return-spmf nn-integral-return* ∗ *nn-integral-mono-AE*)
qed

definition *weight-gpv* :: ('*out*, '*ret*) $\mathcal{I}$ ⇒ ('*a*, '*out*, '*ret*) *gpv* ⇒ *real*
where *weight-gpv* $\mathcal{I}$ *gpv* = *enn2real* (*expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv*)

lemma *weight-gpv-Done* [*simp*]: *weight-gpv* $\mathcal{I}$ (*Done x*) = *1*
by(*simp add*: *weight-gpv-def*)

lemma *weight-gpv-Fail* [*simp*]: *weight-gpv* $\mathcal{I}$ *Fail* = *0*
by(*simp add*: *weight-gpv-def*)

lemma *weight-gpv-lift-spmf* [*simp*]: *weight-gpv* $\mathcal{I}$ (*lift-spmf p*) = *weight-spmf p*
by(*simp add*: *weight-gpv-def measure-spmf.emeasure-eq-measure*)

lemma *weight-gpv-Pause* [*simp*]:
(⋀*r*. *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* ⟹ $\mathcal{I}$ ⊢*g c r* √)
⟹ *weight-gpv* $\mathcal{I}$ (*Pause out c*) = (*if out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ *then INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *weight-gpv* $\mathcal{I}$ (*c r*) *else 0*)
apply(*clarsimp simp add*: *weight-gpv-def in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$)
apply(*erule enn2real-INF*)
apply(*clarsimp simp add*: *expectation-gpv-const-le*[*THEN le-less-trans*])
done

lemma *weight-gpv-nonneg*: *0* ≤ *weight-gpv* $\mathcal{I}$ *gpv*
by(*simp add*: *weight-gpv-def*)

lemma *weight-gpv-le-1*: $\mathcal{I}$ ⊢*g gpv* √ ⟹ *weight-gpv* $\mathcal{I}$ *gpv* ≤ *1*
using *expectation-gpv-const-le*[*of* $\mathcal{I}$ *gpv 0 1*] **by**(*simp add*: *weight-gpv-def enn2real-leI max-def*)

theorem *weight-exec-gpv*:
assumes *callee*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-spmf* (*callee s x*)
and *WT-gpv*: $\mathcal{I}$ ⊢*g gpv* √
and *WT-callee*: ⋀*s*. $\mathcal{I}$ ⊢*c callee s* √
shows *weight-gpv* $\mathcal{I}$ *gpv* ≤ *weight-spmf* (*exec-gpv callee gpv s*)
proof −
have *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* ≤ ∫ ⁺ (*x*, *s*). *1* ∂*measure-spmf* (*exec-gpv callee gpv s*)
using *assms* **by**(*rule expectation-gpv-le-exec-gpv*)
also have ... = *weight-spmf* (*exec-gpv callee gpv s*)
by(*simp add*: *split-def measure-spmf.emeasure-eq-measure*)
finally show *?thesis* **by**(*simp add*: *weight-gpv-def enn2real-leI*)
qed

lemma (**in** *callee-invariant-on*) *weight-exec-gpv*:
assumes *callee*: $\bigwedge s\ x.\ [\![\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow lossless\text{-}spmf\ (callee\ s\ x)$
and *WT-gpv*: $\mathcal{I} \vdash g\ gpv\ \surd$
and *I*: *I s*
shows $weight\text{-}gpv\ \mathcal{I}\ gpv \leq weight\text{-}spmf\ (exec\text{-}gpv\ callee\ gpv\ s)$
including *lifting-syntax*
proof −
{ assume $\exists (Rep :: 's' \Rightarrow 's)\ Abs.\ type\text{-}definition\ Rep\ Abs\ \{s.\ I\ s\}$
then obtain $Rep :: 's' \Rightarrow 's$ **and** *Abs* **where** *td*: *type-definition Rep Abs* {*s. I s*} **by** *blast*
then interpret *td*: *type-definition Rep Abs* {*s. I s*} **.**
define *cr* **where** $cr \equiv \lambda x\ y.\ x = Rep\ y$
have [*transfer-rule*]: *bi-unique cr right-total cr* **using** *td cr-def* **by**(*rule type-def-bi-unique typedef-right-total*)+
have [*transfer-domain-rule*]: *Domainp cr = I* **using** *type-definition-Domainp*[*OF td cr-def*] **by** *simp*

let $?C = eq\text{-}onp\ (\lambda x.\ x \in outs\text{-}\mathcal{I}\ \mathcal{I})$

define *callee′* **where** *callee′* ≡ (*Rep* −−−> *id* −−−> *map-spmf* (*map-prod id Abs*)) *callee*
have [*transfer-rule*]: (*cr* ===> *?C* ===> *rel-spmf* (*rel-prod* (=) *cr*)) *callee callee′*
by(*auto simp add*: *callee′-def rel-fun-def cr-def spmf-rel-map prod.rel-map td.Abs-inverse eq-onp-def intro*!: *rel-spmf-reflI intro*: *td.Rep*[*simplified*] *dest*: *callee-invariant*)
define *s′* **where** *s′* ≡ *Abs s*
have [*transfer-rule*]: *cr s s′* **using** *I* **by**(*simp add*: *cr-def s′-def td.Abs-inverse*)

have [*transfer-rule*]: $rel\text{-}\mathcal{I}\ ?C\ (=)\ \mathcal{I}\ \mathcal{I}$
by(*rule rel-ℐI*)(*auto simp add*: *rel-set-eq set-relator-eq-onp eq-onp-same-args dest*: *eq-onp-to-eq*)
note [*transfer-rule*] = *bi-unique-eq-onp bi-unique-eq*

define *gpv′* **where** $gpv' \equiv restrict\text{-}gpv\ \mathcal{I}\ gpv$
have [*transfer-rule*]: *rel-gpv* (=) *?C gpv′ gpv′*
by(*fold eq-onp-top-eq-eq*)(*auto simp add*: *gpv.rel-eq-onp eq-onp-same-args pred-gpv-def gpv′-def dest*: *in-outs′-restrict-gpvD*)

define *weight-spmf′* :: ('*c* × '*s′*) *spmf* ⇒ *real* **where** *weight-spmf′* ≡ *weight-spmf*
define *weight-spmf″* :: ('*c* × '*s*) *spmf* ⇒ *real* **where** *weight-spmf″* ≡ *weight-spmf*
have [*transfer-rule*]: (*rel-spmf* (*rel-prod* (=) *cr*) ===> (=)) *weight-spmf″ weight-spmf′*
by(*simp add*: *weight-spmf′-def weight-spmf″-def weight-spmf-parametric*)

have [*rule-format*]: $\bigwedge s.\ \forall x \in outs\text{-}\mathcal{I}\ \mathcal{I}.\ lossless\text{-}spmf\ (callee'\ s\ x)$
by(*transfer*)(*blast intro*: *callee*)
moreover have $\mathcal{I} \vdash g\ gpv'\ \surd$ **by**(*simp add*: *gpv′-def*)
moreover have $\bigwedge s.\ \mathcal{I} \vdash c\ callee'\ s\ \surd$ **by** *transfer*(*rule WT-callee*)

ultimately have ∗∗: *weight-gpv* $\mathcal{I}$ *gpv*′ ≤ *weight-spmf*′ (*exec-gpv callee*′ *gpv*′ *s*′)
unfolding *weight-spmf*′*-def* **by**(*rule weight-exec-gpv*)
have [*transfer-rule*]: ((=) ===> *?C* ===> *rel-spmf* (*rel-prod* (=) (=))) *callee callee*
by(*simp add*: *rel-fun-def eq-onp-def prod.rel-eq*)
have *weight-gpv* $\mathcal{I}$ *gpv*′ ≤ *weight-spmf*′′ (*exec-gpv callee gpv*′ *s*) **using** ∗∗ **by** *transfer*
also have *exec-gpv callee gpv*′ *s* = *exec-gpv callee gpv s*
unfolding *gpv*′*-def* **using** *WT-gpv I* **by**(*rule exec-gpv-restrict-gpv-invariant*)
also have *weight-gpv* $\mathcal{I}$ *gpv*′ = *weight-gpv* $\mathcal{I}$ *gpv* **using** *WT-gpv*
by(*simp add*: *gpv*′*-def expectation-gpv-restrict-gpv weight-gpv-def*)
finally have *?thesis* **by**(*simp add*: *weight-spmf*′′*-def*) **}**
from *this*[*cancel-type-definition*] *I* **show** *?thesis* **by** *blast*
qed

6.5 Probabilistic termination

definition *pgen-lossless-gpv* :: *ennreal* ⇒ (′*c*, ′*r*) $\mathcal{I}$ ⇒ (′*a*, ′*c*, ′*r*) *gpv* ⇒ *bool*
where *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* = (*expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1*) **for** *fail*

abbreviation *plossless-gpv* :: (′*c*, ′*r*) $\mathcal{I}$ ⇒ (′*a*, ′*c*, ′*r*) *gpv* ⇒ *bool*
where *plossless-gpv* ≡ *pgen-lossless-gpv 0*

abbreviation *pfinite-gpv* :: (′*c*, ′*r*) $\mathcal{I}$ ⇒ (′*a*, ′*c*, ′*r*) *gpv* ⇒ *bool*
where *pfinite-gpv* ≡ *pgen-lossless-gpv 1*

lemma *pgen-lossless-gpvI* [*intro?*]: *expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1* ⟹ *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* **for** *fail*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma *pgen-lossless-gpvD*: *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* ⟹ *expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1* **for** *fail*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma *lossless-imp-plossless-gpv*:
assumes *lossless-gpv* $\mathcal{I}$ *gpv* $\mathcal{I}$ ⊢*g gpv* √
shows *plossless-gpv* $\mathcal{I}$ *gpv*
proof
show *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* = *1* **using** *assms*
proof(*induction rule*: *lossless-WT-gpv-induct*)
case (*lossless-gpv p*)
have *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) (*GPV p*) = *nn-integral* (*measure-spmf p*) (*case-generat* (λ-. *1*) (λ*out c*. *INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *1*))
by(*subst expectation-gpv.simps*)(*clarsimp split*: *generat.split cong*: *INF-cong simp add*: *lossless-gpv.IH intro*!: *nn-integral-cong-AE*)
also have ... = *nn-integral* (*measure-spmf p*) (λ-. *1*)
by(*intro nn-integral-cong-AE*)(*auto split*: *generat.split dest*!: *lossless-gpv.hyps*(*2*)

simp add: *in-outs-ℐ-iff-responses-ℐ*)
finally show *?case* **by**(*simp add*: *measure-spmf.emeasure-eq-measure lossless-weight-spmfD lossless-gpv.hyps*(*1*))
qed
qed

lemma *finite-imp-pfinite-gpv*:
assumes *finite-gpv ℐ gpv ℐ ⊢g gpv √*
shows *pfinite-gpv ℐ gpv*
proof
show *expectation-gpv 1 ℐ* (λ-. *1*) *gpv* = *1* **using** *assms*
proof(*induction rule*: *finite-gpv-induct*)
case (*finite-gpv gpv*)
then have *expectation-gpv 1 ℐ* (λ-. *1*) *gpv* = *nn-integral* (*measure-spmf* (*the-gpv gpv*)) (*case-generat* (λ-. *1*) (λ*out c*. *INF r*∈*responses-ℐ ℐ out*. *1*)) + *pmf* (*the-gpv gpv*) *None*
by(*subst expectation-gpv.simps*)(*clarsimp intro*!: *nn-integral-cong-AE INF-cong*[*OF refl*] *split*!: *generat.split simp add*: *WT-gpv-ContD*)
also have ... = *nn-integral* (*measure-spmf* (*the-gpv gpv*)) (λ-. *1*) + *pmf* (*the-gpv gpv*) *None*
by(*intro arg-cong2*[**where** *f*=(+)] *nn-integral-cong-AE*)
(*auto split*: *generat.split dest*!: *WT-gpv-OutD*[*OF finite-gpv.prems*] *simp add*: *in-outs-ℐ-iff-responses-ℐ*)
finally show *?case*
by(*simp add*: *measure-spmf.emeasure-eq-measure ennreal-plus*[*symmetric*] *del*: *ennreal-plus*)
(*simp add*: *pmf-None-eq-weight-spmf*)
qed
qed

lemma *plossless-gpv-lossless-spmfD*:
assumes *lossless*: *plossless-gpv ℐ gpv*
and *WT*: *ℐ ⊢g gpv √*
shows *lossless-spmf* (*the-gpv gpv*)
proof −
have *1* = *expectation-gpv 0 ℐ* (λ-. *1*) *gpv*
using *lossless* **by**(*auto dest*: *pgen-lossless-gpvD simp add*: *weight-gpv-def*)
also have ... = ∫⁺ *generat*. (*case generat of Pure x* ⇒ *1* | *IO out c* ⇒ *INF r*∈*responses-ℐ ℐ out*. *expectation-gpv 0 ℐ* (λ-. *1*) (*c r*)) ∂*measure-spmf* (*the-gpv gpv*)
by(*subst expectation-gpv.simps*)(*auto*)
also have ... ≤ ∫⁺ *generat*. (*case generat of Pure x* ⇒ *1* | *IO out c* ⇒ *1*) ∂*measure-spmf* (*the-gpv gpv*)
apply(*rule nn-integral-mono-AE*)
apply(*clarsimp split*: *generat.split*)
apply(*frule WT-gpv-OutD*[*OF WT*])
using *expectation-gpv-const-le*[*of ℐ - 0 1*]
apply(*auto simp add*: *in-outs-ℐ-iff-responses-ℐ max-def intro*: *INF-lower2 WT-gpv-ContD*[*OF WT*] *dest*: *WT-gpv-OutD*[*OF WT*])

done
also have ... = *weight-spmf* (*the-gpv gpv*)
by(*auto simp add*: *weight-spmf-eq-nn-integral-spmf nn-integral-measure-spmf intro*!: *nn-integral-cong split*: *generat.split*)
finally show *?thesis* **using** *weight-spmf-le-1*[*of the-gpv gpv*] **by**(*simp add*: *lossless-spmf-def*)
qed

lemma
shows *plossless-gpv-ContD*:
⟦ *plossless-gpv* 𝓘 *gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*𝓘 𝓘 *out*; 𝓘 ⊢*g gpv* √ ⟧
⟹ *plossless-gpv* 𝓘 (*c input*)
and *pfinite-gpv-ContD*:
⟦ *pfinite-gpv* 𝓘 *gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*𝓘 𝓘 *out*; 𝓘 ⊢*g gpv* √ ⟧
⟹ *pfinite-gpv* 𝓘 (*c input*)
proof(*rule-tac* [!] *pgen-lossless-gpvI*, *rule-tac* [!] *antisym*[*rotated*], *rule-tac ccontr*, *rule-tac* [*3*] *ccontr*)
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *input*: *input* ∈ *responses-*𝓘 𝓘 *out*
and *WT*: 𝓘 ⊢*g gpv* √
from *WT IO input* **have** *WT'*: 𝓘 ⊢*g c input* √ **by**(*rule WT-gpv-ContD*)
from *expectation-gpv-const-le*[*OF this*, *of 0 1*] *expectation-gpv-const-le*[*OF this*, *of 1 1*]
show *expectation-gpv 0* 𝓘 (λ-. *1*) (*c input*) ≤ *1*
and *expectation-gpv 1* 𝓘 (λ-. *1*) (*c input*) ≤ *1* **by**(*simp-all add*: *max-def*)

have *less*: *expectation-gpv fail* 𝓘 (λ-. *1*) *gpv* < *weight-spmf* (*the-gpv gpv*) + *fail* ∗ *pmf* (*the-gpv gpv*) *None*
if *fail*: *fail* ≤ *1* **and** ∗: ¬ *1* ≤ *expectation-gpv fail* 𝓘 (λ-. *1*) (*c input*) **for** *fail* :: *ennreal*
proof −
have *expectation-gpv fail* 𝓘 (λ-. *1*) *gpv* = (∫⁺ *generat*. (*case generat of Pure x* ⇒ *1* | *IO out c* ⇒ *INF r*∈*responses-*𝓘 𝓘 *out*. *expectation-gpv fail* 𝓘 (λ-. *1*) (*c r*)) ∗ *spmf* (*the-gpv gpv*) *generat* ∗ *indicator* (*UNIV* − {*IO out c*}) *generat* + (*INF r*∈*responses-*𝓘 𝓘 *out*. *expectation-gpv fail* 𝓘 (λ-. *1*) (*c r*)) ∗ *spmf* (*the-gpv gpv*) (*IO out c*) ∗ *indicator* {*IO out c*} *generat* ∂*count-space UNIV*) + *fail* ∗ *pmf* (*the-gpv gpv*) *None*
by(*subst expectation-gpv.simps*)(*auto simp add*: *nn-integral-measure-spmf mult.commute intro*!: *nn-integral-cong split*: *split-indicator generat.split*)
also have ... = (∫⁺ *generat*. (*case generat of Pure x* ⇒ *1* | *IO out c* ⇒ *INF r*∈*responses-*𝓘 𝓘 *out*. *expectation-gpv fail* 𝓘 (λ-. *1*) (*c r*)) ∗ *spmf* (*the-gpv gpv*) *generat* ∗ *indicator* (*UNIV* − {*IO out c*}) *generat* ∂*count-space UNIV*) +
(*INF r*∈*responses-*𝓘 𝓘 *out*. *expectation-gpv fail* 𝓘 (λ-. *1*) (*c r*)) ∗ *spmf* (*the-gpv gpv*) (*IO out c*) + *fail* ∗ *pmf* (*the-gpv gpv*) *None* (**is** - = *?rest* + *?cr* + -)
by(*subst nn-integral-add*) *simp-all*
also from *calculation expectation-gpv-const-le*[*OF WT*, *of fail 1*] *fail* **have** *fin*: *?rest* ≠ ∞

by(*auto simp add*: *top-add top-unique max-def split*: *if-split-asm*)
have *?cr* ≤ *expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) (*c input*) ∗ *spmf* (*the-gpv gpv*) (*IO out c*)
by(*rule mult-right-mono INF-lower*[*OF input*])+ *simp*
also have *?rest* + ... < *?rest* + *1* ∗ *ennreal* (*spmf* (*the-gpv gpv*) (*IO out c*))
unfolding *ennreal-add-left-cancel-less* **using** ∗ *IO*
by(*intro conjI fin ennreal-mult-strict-right-mono*)(*simp-all add*: *not-le weight-gpv-def in-set-spmf-iff-spmf*)
also have *?rest* ≤ ∫ + *generat*. *spmf* (*the-gpv gpv*) *generat* ∗ *indicator* (*UNIV* − {*IO out c*}) *generat* ∂*count-space UNIV*
apply(*rule nn-integral-mono*)
apply(*clarsimp split*: *generat.split split-indicator*)
apply(*rule ennreal-mult-le-self2I*)
apply *simp*
subgoal premises *prems* **for** *out′ c′*
apply(*subgoal-tac IO out′ c′* ∈ *set-spmf* (*the-gpv gpv*))
apply(*frule WT-gpv-OutD*[*OF WT*])
apply(*simp add*: *in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$)
apply *safe*
apply(*erule notE*)
apply(*rule INF-lower2*, *assumption*)
apply(*rule expectation-gpv-const-le*[*THEN order-trans*])
apply(*erule* (*1*) *WT-gpv-ContD*[*OF WT*])
apply(*simp add*: *fail*)
using *prems* **by**(*simp add*: *in-set-spmf-iff-spmf*)
done
also have ... + *1* ∗ *ennreal* (*spmf* (*the-gpv gpv*) (*IO out c*)) =
(∫ + *generat*. *spmf* (*the-gpv gpv*) *generat* ∗ *indicator* (*UNIV* − {*IO out c*}) *generat* + *ennreal* (*spmf* (*the-gpv gpv*) (*IO out c*)) ∗ *indicator* {*IO out c*} *generat* ∂*count-space UNIV*)
by(*subst nn-integral-add*)(*simp-all*)
also have ... = ∫ + *generat*. *spmf* (*the-gpv gpv*) *generat* ∂*count-space UNIV*
by(*auto intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = *weight-spmf* (*the-gpv gpv*) **by**(*simp add*: *nn-integral-spmf measure-spmf.emeasure-eq-measure space-measure-spmf*)
finally show *?thesis* **using** *fail*
by(*fastforce simp add*: *top-unique add-mono ennreal-plus*[*symmetric*] *ennreal-mult-eq-top-iff*)
qed

show *False* **if** ∗: ¬ *1* ≤ *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) (*c input*) **and** *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
using *less*[*OF* - ∗] *plossless-gpv-lossless-spmfD*[*OF lossless WT*] *lossless*[*THEN pgen-lossless-gpvD*]
by(*simp add*: *lossless-spmf-def*)

show *False* **if** ∗: ¬ *1* ≤ *expectation-gpv 1* $\mathcal{I}$ (λ-. *1*) (*c input*) **and** *finite*: *pfinite-gpv* $\mathcal{I}$ *gpv*
using *less*[*OF* - ∗] *finite*[*THEN pgen-lossless-gpvD*] **by**(*simp add*: *ennreal-plus*[*symmetric*]

del: *ennreal-plus*)(*simp add*: *pmf-None-eq-weight-spmf*)
qed

lemma *plossless-iff-colossless-pfinite*:
  assumes *WT*: $\mathcal{I}$ ⊢*g gpv* √
  shows *plossless-gpv* $\mathcal{I}$ *gpv* ⟷ *colossless-gpv* $\mathcal{I}$ *gpv* ∧ *pfinite-gpv* $\mathcal{I}$ *gpv*
proof(*intro iffI conjI*; (*elim conjE*)?)
  assume ∗: *plossless-gpv* $\mathcal{I}$ *gpv*
  show *colossless-gpv* $\mathcal{I}$ *gpv* **using** ∗ *WT*
  proof(*coinduction arbitrary*: *gpv*)
    case (*colossless-gpv gpv*)
    have *?lossless-spmf* **using** *colossless-gpv* **by**(*rule plossless-gpv-lossless-spmfD*)
    moreover have *?continuation* **using** *colossless-gpv*
      by(*auto intro*: *plossless-gpv-ContD WT-gpv-ContD*)
    ultimately show *?case* **..**
  qed

  show *pfinite-gpv* $\mathcal{I}$ *gpv* **unfolding** *pgen-lossless-gpv-def*
  proof(*rule antisym*)
    from *expectation-gpv-const-le*[*OF WT*, *of 1 1*] **show** *expectation-gpv 1* $\mathcal{I}$ (λ-. *1*) *gpv* ≤ *1* **by** *simp*
    have *1* = *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* **using** ∗ **by**(*simp add*: *pgen-lossless-gpv-def*)
    also have ... ≤ *expectation-gpv 1* $\mathcal{I}$ (λ-. *1*) *gpv* **by**(*rule expectation-gpv-mono*) *simp-all*
    finally show *1* ≤ ... **.**
  qed
next
  show *plossless-gpv* $\mathcal{I}$ *gpv* **if** *colossless-gpv* $\mathcal{I}$ *gpv* **and** *pfinite-gpv* $\mathcal{I}$ *gpv* **using** *that*
    by(*simp add*: *pgen-lossless-gpv-def cong*: *expectation-gpv-cong-fail*)
qed

lemma *pgen-lossless-gpv-Done* [*simp*]: *pgen-lossless-gpv fail* $\mathcal{I}$ (*Done x*) **for** *fail*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma *pgen-lossless-gpv-Fail* [*simp*]: *pgen-lossless-gpv fail* $\mathcal{I}$ *Fail* ⟷ *fail* = *1* **for** *fail*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma *pgen-lossless-gpv-PauseI* [*simp*, *intro*!]:
  ⟦ *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; ⋀*r*. *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* ⟹ *pgen-lossless-gpv fail* $\mathcal{I}$ (*c r*) ⟧
   ⟹ *pgen-lossless-gpv fail* $\mathcal{I}$ (*Pause out c*) **for** *fail*
by(*simp add*: *pgen-lossless-gpv-def weight-gpv-def in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$)

lemma *pgen-lossless-gpv-bindI* [*simp*, *intro*!]:
  ⟦ *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv*; ⋀*x*. *x* ∈ *results-gpv* $\mathcal{I}$ *gpv* ⟹ *pgen-lossless-gpv fail* $\mathcal{I}$ (*f x*) ⟧
  ⟹ *pgen-lossless-gpv fail* $\mathcal{I}$ (*bind-gpv gpv f*) **for** *fail*
by(*simp add*: *pgen-lossless-gpv-def weight-gpv-def o-def cong*: *expectation-gpv-cong*)

lemma *pgen-lossless-gpv-lift-spmf* [*simp*]:
pgen-lossless-gpv fail $\mathcal{I}$ (*lift-spmf p*) $\longleftrightarrow$ *lossless-spmf p* $\vee$ *fail* = *1* **for** *fail*
apply(*cases fail*)
subgoal
by(*simp add*: *pgen-lossless-gpv-def lossless-spmf-def measure-spmf.emeasure-eq-measure pmf-None-eq-weight-spmf ennreal-minus ennreal-mult*[*symmetric*] *weight-spmf-le-1 ennreal-plus*[*symmetric*] *del*: *ennreal-plus*)
(*metis add-diff-cancel-left′ diff-add-cancel eq-iff-diff-eq-0 mult-cancel-right1*)
subgoal by(*simp add*: *pgen-lossless-gpv-def measure-spmf.emeasure-eq-measure ennreal-top-mult lossless-spmf-def add-top weight-spmf-conv-pmf-None*)
done

lemma *expectation-gpv-top-pfinite*:
assumes *pfinite-gpv* $\mathcal{I}$ *gpv*
shows *expectation-gpv* $\top$ $\mathcal{I}$ (λ-. $\top$) *gpv* = $\top$
proof(*rule ccontr*)
assume ∗: ¬ *?thesis*
have *1* = *expectation-gpv 1* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *assms* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have . . . ≤ *expectation-gpv* $\top$ $\mathcal{I}$ (λ-. $\top$) *gpv* **by**(*rule expectation-gpv-mono*)(*simp-all add*: *le-fun-def*)
also have . . . = *0* **using** *expectation-gpv-cmult*[*of 2* $\top$ $\mathcal{I}$ λ-. $\top$ *gpv*] ∗
by(*simp add*: *ennreal-mult-top*) (*metis ennreal-mult-cancel-left mult.commute mult-numeral-1-right not-gr-zero numeral-eq-one-iff semiring-norm*(*85*) *zero-neq-numeral*)
finally show *False* **by** *simp*
qed

lemma *pfinite-INF-le-expectation-gpv*:
fixes *fail* $\mathcal{I}$ *gpv f*
defines *c* ≡ *min* (*INF x*∈*results-gpv* $\mathcal{I}$ *gpv*. *f x*) *fail*
assumes *fin*: *pfinite-gpv* $\mathcal{I}$ *gpv*
shows *c* ≤ *expectation-gpv fail* $\mathcal{I}$ *f gpv* (**is** *?lhs* ≤ *?rhs*)
proof(*cases c* > *0*)
case *True*
have *c* = *c* ∗ *expectation-gpv 1* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *assms* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have . . . = *expectation-gpv c* $\mathcal{I}$ (λ-. *c*) *gpv* **using** *fin True*
by(*cases c* = $\top$)(*simp-all add*: *expectation-gpv-top-pfinite ennreal-top-mult expectation-gpv-cmult*, *simp add*: *pgen-lossless-gpv-def*)
also have . . . ≤ *?rhs* **by**(*rule expectation-gpv-mono-strong*)(*auto simp add*: *c-def min-def intro*: *INF-lower2*)
finally show *?thesis* **.**
qed *simp*

lemma *plossless-INF-le-expectation-gpv*:
fixes *fail*
assumes *plossless-gpv* $\mathcal{I}$ *gpv* **and** $\mathcal{I}$ ⊢*g gpv* √
shows (*INF x*∈*results-gpv* $\mathcal{I}$ *gpv*. *f x*) ≤ *expectation-gpv fail* $\mathcal{I}$ *f gpv* (**is** *?lhs* ≤ *?rhs*)

proof −
  from *assms* **have** *fin*: *pfinite-gpv* $\mathcal{I}$ *gpv* **and** *co*: *colossless-gpv* $\mathcal{I}$ *gpv*
    by(*simp-all add*: *plossless-iff-colossless-pfinite*)
  have *?lhs* ≤ *min ?lhs* ⊤ **by**(*simp add*: *min-def*)
  also have ... ≤ *expectation-gpv* ⊤ $\mathcal{I}$ *f gpv* **using** *fin* **by**(*rule pfinite-INF-le-expectation-gpv*)
  also have ... = *?rhs* **using** *co* **by**(*simp add*: *expectation-gpv-cong-fail*)
  finally show *?thesis* **.**
qed

lemma *expectation-gpv-le-inline*:
  fixes $\mathcal{I}'$
  defines *expectation-gpv2* ≡ *expectation-gpv 0* $\mathcal{I}'$
  assumes *callee*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
    and *callee'*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *results-gpv* $\mathcal{I}'$ (*callee s x*) ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* × *UNIV*
    and *WT-gpv*: $\mathcal{I}$ ⊢g *gpv* √
    and *WT-callee*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}'$ ⊢g *callee s x* √
  shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*)
  using *WT-gpv*
proof(*induction arbitrary*: *gpv s rule*: *expectation-gpv-fixp-induct*)
  case *adm* **show** *?case* **by** *simp*
  case *bottom* **show** *?case* **by** *simp*
  case (*step expectation-gpv'*)
  { fix *out c*
    assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
    with *step.prems* **have** *out*: *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*rule WT-gpv-OutD*)
    have (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) = ∫⁺ *generat*. (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) ∂*measure-spmf* (*the-gpv* (*callee s out*))
      using *WT-callee*[*OF out*, *of s*] *callee*[*OF out*, *of s*]
    by(*clarsimp simp add*: *measure-spmf.emeasure-eq-measure plossless-iff-colossless-pfinite colossless-gpv-lossless-spmfD lossless-weight-spmfD*)
    also have ... ≤ ∫⁺ *generat*. (*case generat of Pure* (*x*, *s'*) ⇒
          ∫⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
            | *Inr* (*out'*, *callee'*, *rpv*) ⇒ *INF r'*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*. *expectation-gpv 0* $\mathcal{I}'$ (λ(*r*, *s'*). *expectation-gpv 0* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s'*)) (*callee' r'*))
          ∂*measure-spmf* (*inline1 callee* (*c x*) *s'*)
        | *IO out' rpv* ⇒ *INF r'*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out'*. *expectation-gpv 0* $\mathcal{I}'$ (λ(*r'*, *s'*). *expectation-gpv 0* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*c r'*) *s'*)) (*rpv r'*))
      ∂*measure-spmf* (*the-gpv* (*callee s out*))
    proof(*rule nn-integral-mono-AE*; *simp split*!: *generat.split*)
      fix *x s'*
      assume *Pure*: *Pure* (*x*, *s'*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
      hence (*x*, *s'*) ∈ *results-gpv* $\mathcal{I}'$ (*callee s out*) **by**(*rule results-gpv.Pure*)
      with *callee'*[*OF out*, *of s*] **have** *x*: *x* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* **by** *blast*
      hence (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *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*′)
by(*rule step.IH*)(*rule WT-gpv-ContD*[*OF step.prems*(*1*) *IO x*] *step.prems*|*assumption*)+
also have ... = ∫⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out*′, *callee*′, *rpv*) ⇒ *INF r*′∈*responses-ℐ ℐ*′ *out*′. *expectation-gpv 0 ℐ*′ (λ(*r*, *s*′). *expectation-gpv 0 ℐ*′ (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s*′)) (*callee*′ *r*′))
∂*measure-spmf* (*inline1 callee* (*c x*) *s*′)
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*)
finally show (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv*′ (*c r*)) ≤ ... .
next
fix *out*′ *rpv*
assume *IO*′: *IO out*′ *rpv* ∈ *set-spmf* (*the-gpv* (*callee s out*))
have (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv*′ (*c r*)) ≤ (*INF* (*r*, *s*′)∈(⋃ *r*′∈*responses-ℐ ℐ*′ *out*′. *results-gpv ℐ*′ (*rpv r*′)). *expectation-gpv*′ (*c r*))
using *IO*′ *callee*′[*OF out*, *of s*] **by**(*intro INF-mono*)(*auto intro*: *results-gpv.IO*)
also have ... = (*INF r*′∈*responses-ℐ ℐ*′ *out*′. *INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*rpv r*′). *expectation-gpv*′ (*c r*))
by(*simp add*: *INF-UNION*)
also have ... ≤ (*INF r*′∈*responses-ℐ ℐ*′ *out*′. *expectation-gpv 0 ℐ*′ (λ(*r*′, *s*′). *expectation-gpv 0 ℐ*′ (λ(*x*, *s*). *f x*) (*inline callee* (*c r*′) *s*′)) (*rpv r*′))
proof(*rule INF-mono*, *rule bexI*)
fix *r*′
assume *r*′: *r*′ ∈ *responses-ℐ ℐ*′ *out*′
have (*INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*rpv r*′). *expectation-gpv*′ (*c r*)) ≤ (*INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*rpv r*′). *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c r*) *s*′))
using *IO IO*′ *step.prems out callee*′[*OF out*, *of s*] *r*′
by(*auto intro*!: *INF-mono rev-bexI step.IH dest*: *WT-gpv-ContD intro*: *results-gpv.IO*)
also have ... ≤ *expectation-gpv 0 ℐ*′ (λ(*r*′, *s*′). *expectation-gpv 0 ℐ*′ (λ(*x*, *s*). *f x*) (*inline callee* (*c r*′) *s*′)) (*rpv r*′)
unfolding *expectation-gpv2-def* **using** *plossless-gpv-ContD*[*OF callee*, *OF out IO*′ *r*′] *WT-callee*[*OF out*, *of s*] *IO*′ *r*′
by(*intro plossless-INF-le-expectation-gpv*)(*auto intro*: *WT-gpv-ContD*)
finally show (*INF* (*r*, *s*′)∈*results-gpv ℐ*′ (*rpv r*′). *expectation-gpv*′ (*c r*)) ≤ ... .
qed
finally show (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv*′ (*c r*)) ≤ ... .
qed
also note *calculation* **}**
then show *?case* **unfolding** *expectation-gpv2-def*
apply(*rewrite expectation-gpv.simps*)
apply(*rewrite inline-sel*)
apply(*simp add*: *o-def pmf-map-spmf-None*)
apply(*rewrite sum.case-distrib*[**where** *h*=*case-generat* - -])
apply(*simp cong del*: *sum.case-cong-weak*)
apply(*simp add*: *split-beta o-def cong del*: *sum.case-cong-weak*)
apply(*rewrite inline1.simps*)

apply(*rewrite measure-spmf-bind*)
apply(*rewrite nn-integral-bind*[**where** *B*=*measure-spmf* -])
apply *simp*
apply(*simp add*: *space-subprob-algebra*)
apply(*rule nn-integral-mono-AE*)
apply(*clarsimp split*!: *generat.split*)
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return*)
apply(*rewrite measure-spmf-bind*)
apply(*simp add*: *nn-integral-bind*[**where** *B*=*measure-spmf* -] *space-subprob-algebra*)
apply(*subst generat.case-distrib*[**where** *h*=*measure-spmf*])
apply(*subst generat.case-distrib*[**where** *h*=λ*x. nn-integral x* -])
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return split-def*)
done
qed

lemma *plossless-inline*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I}$ ⊢*g gpv* √
and *callee*: ⋀*s x. x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *callee'*: ⋀*s x. x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *results-gpv* $\mathcal{I}'$ (*callee s x*) ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* × *UNIV*
and *WT-callee*: ⋀*s x. x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}'$ ⊢*g callee s x* √
shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
unfolding *pgen-lossless-gpv-def*
proof(*rule antisym*)
have *WT'*: $\mathcal{I}'$ ⊢*g inline callee gpv s* √ **using** *callee' WT-callee WT* **by**(*rule WT-gpv-inline*)
from *expectation-gpv-const-le*[*OF WT'*, *of 0 1*]
show *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*) ≤ *1* **by**(*simp add*: *max-def*)

have *1* = *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *lossless* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have ... ≤ *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*)
by(*rule expectation-gpv-le-inline*[*unfolded split-def*]; *rule callee callee' WT WT-callee*)
finally show *1* ≤ ... **.**
qed

lemma *plossless-exec-gpv*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I}$ ⊢*g gpv* √
and *callee*: ⋀*s x. x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-spmf* (*callee s x*)
and *callee'*: ⋀*s x. x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *set-spmf* (*callee s x*) ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* × *UNIV*
shows *lossless-spmf* (*exec-gpv callee gpv s*)
proof −
have *plossless-gpv* $\mathcal{I}$*-full* (*inline* (λ*s x. lift-spmf* (*callee s x*)) *gpv s*)
using *lossless WT* **by**(*rule plossless-inline*)(*simp-all add*: *callee callee'*)
from *this*[*THEN plossless-gpv-lossless-spmfD*] **show** *?thesis*
unfolding *exec-gpv-conv-inline1* **by**(*simp add*: *inline-sel*)
qed

lemma *expectation-gpv-$\mathcal{I}$-mono*:
defines *expectation-gpv'* ≡ *expectation-gpv*
assumes *le*: $\mathcal{I} \leq \mathcal{I}'$
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *expectation-gpv fail* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv' fail* $\mathcal{I}'$ *f gpv*
using *WT*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case *step* [*unfolded expectation-gpv'-def*]: (*step expectation-gpv'*)
show *?case* **unfolding** *expectation-gpv'-def*
by(*subst expectation-gpv.simps*)
(*clarsimp intro*!: *add-mono nn-integral-mono-AE INF-mono split*: *generat.split* , *auto intro*!: *bexI step add-mono nn-integral-mono-AE INF-mono split*: *generat.split dest*: *WT-gpvD*[*OF step.prems*] *intro*!: *step dest*: *responses-$\mathcal{I}$-mono*[*OF le*])
qed

lemma *pgen-lossless-gpv-mono*:
assumes ∗: *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv*
and *le*: $\mathcal{I} \leq \mathcal{I}'$
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *fail*: *fail* ≤ *1*
shows *pgen-lossless-gpv fail* $\mathcal{I}'$ *gpv*
unfolding *pgen-lossless-gpv-def*
proof(*rule antisym*)
from *WT le* **have** $\mathcal{I}' \vdash g$ *gpv* $\surd$ **by**(*rule WT-gpv-$\mathcal{I}$-mono*)
from *expectation-gpv-const-le*[*OF this*, *of fail 1*] *fail*
show *expectation-gpv fail* $\mathcal{I}'$ (λ-. *1*) *gpv* ≤ *1* **by**(*simp add*: *max-def split*: *if-split-asm*)
from *expectation-gpv-$\mathcal{I}$-mono*[*OF le WT*, *of fail* λ-. *1*] ∗
show *expectation-gpv fail* $\mathcal{I}'$ (λ-. *1*) *gpv* ≥ *1* **by**(*simp add*: *pgen-lossless-gpv-def*)
qed

lemma *plossless-gpv-mono*:
⟦ *plossless-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ ⟹ *plossless-gpv* $\mathcal{I}'$ *gpv*
by(*erule pgen-lossless-gpv-mono*; *simp*)

lemma *pfinite-gpv-mono*:
⟦ *pfinite-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ ⟹ *pfinite-gpv* $\mathcal{I}'$ *gpv*
by(*erule pgen-lossless-gpv-mono*; *simp*)

lemma *pgen-lossless-gpv-parametric'*: **includes** *lifting-syntax* **shows**
((=) ===> *rel-$\mathcal{I}$ C R* ===> *rel-gpv'' A C R* ===> (=)) *pgen-lossless-gpv pgen-lossless-gpv*
unfolding *pgen-lossless-gpv-def* **supply** *expectation-gpv-parametric'*[*transfer-rule*]
by *transfer-prover*

lemma *pgen-lossless-gpv-parametric*: **includes** *lifting-syntax* **shows**

((=) ===> *rel-ℐ* *C* (=) ===> *rel-gpv* *A* *C* ===> (=)) *pgen-lossless-gpv* *pgen-lossless-gpv*
using *pgen-lossless-gpv-parametric′*[*of C* (=) *A*] **by**(*simp add*: *rel-gpv-conv-rel-gpv′′*)

lemma *pgen-lossless-gpv-map-gpv-id* [*simp*]:
pgen-lossless-gpv fail ℐ (*map-gpv f id gpv*) = *pgen-lossless-gpv fail ℐ gpv*
using *pgen-lossless-gpv-parametric*[*of BNF-Def.Grp UNIV id BNF-Def.Grp UNIV f*]
unfolding *gpv.rel-Grp*
by(*auto simp add*: *eq-alt*[*symmetric*] *rel-ℐ-eq rel-fun-def Grp-iff*)

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

lemma *expectation-gpv-le-inline*:
defines *expectation-gpv2* ≡ *expectation-gpv 0 ℐ′*
assumes *callee*: ⋀*s x*. ⟦ *x* ∈ *outs-ℐ ℐ*; *I s* ⟧ ⟹ *plossless-gpv ℐ′* (*callee s x*)
and *WT-gpv*: *ℐ* ⊢*g gpv* √
and *I*: *I s*
shows *expectation-gpv 0 ℐ f gpv* ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*)
using *WT-gpv I*
proof(*induction arbitrary*: *gpv s rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv′*)
{ **fix** *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
with *step.prems* (*1*) **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpv-OutD*)
have (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) = ∫ ⁺ *generat*. (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ∂*measure-spmf* (*the-gpv* (*callee s out*))
using *WT-callee*[*OF out*, *of s*] *callee*[*OF out*, *of s*] ‹*I s*›
by(*clarsimp simp add*: *measure-spmf.emeasure-eq-measure plossless-iff-colossless-pfinite colossless-gpv-lossless-spmfD lossless-weight-spmfD*)
also have ... ≤ ∫ ⁺ *generat*. (*case generat of Pure* (*x*, *s′*) ⇒
∫ ⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out′*, *callee′*, *rpv*) ⇒ *INF r′*∈*responses-ℐ ℐ′ out′*. *expectation-gpv 0 ℐ′* (λ(*r*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s′*)) (*callee′ r′*))
∂*measure-spmf* (*inline1 callee* (*c x*) *s′*)
| *IO out′ rpv* ⇒ *INF r′*∈*responses-ℐ ℐ′ out′*. *expectation-gpv 0 ℐ′* (λ(*r′*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*c r′*) *s′*)) (*rpv r′*))
∂*measure-spmf* (*the-gpv* (*callee s out*))
proof(*rule nn-integral-mono-AE*; *simp split*!: *generat.split*)
fix *x s′*
assume *Pure*: *Pure* (*x*, *s′*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
hence (*x*, *s′*) ∈ *results-gpv ℐ′* (*callee s out*) **by**(*rule results-gpv.Pure*)
with *results-callee*[*OF out*, *of s*] ‹*I s*› **have** *x*: *x* ∈ *responses-ℐ ℐ out* **and** *I s′* **by** *blast+*
from *x* **have** (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ≤ *expecta-*

tion-gpv′ (*c x*) **by**(*rule INF-lower*)
  also have ... ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c x*) *s′*)
   by(*rule step.IH*)(*rule WT-gpv-ContD*[*OF step.prems*(*1*) *IO x*] *step.prems* ‹*I s′*›|*assumption*)+
  also have ... = ∫⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
    | *Inr* (*out′*, *callee′*, *rpv*) ⇒ *INF r′*∈*responses-ℐ ℐ′ out′*. *expectation-gpv 0 ℐ′* (λ(*r*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s′*)) (*callee′ r′*))
    ∂*measure-spmf* (*inline1 callee* (*c x*) *s′*)
   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*)
  finally show (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ≤ ... .
 next
  fix *out′ rpv*
  assume *IO′*: *IO out′ rpv* ∈ *set-spmf* (*the-gpv* (*callee s out*))
 have (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ≤ (*INF* (*r*, *s′*)∈(⋃ *r′*∈*responses-ℐ ℐ′ out′*. *results-gpv ℐ′* (*rpv r′*)). *expectation-gpv′* (*c r*))
   using *IO′ results-callee*[*OF out*, *of s*] ‹*I s*› **by**(*intro INF-mono*)(*auto intro*: *results-gpv.IO*)
  also have ... = (*INF r′*∈*responses-ℐ ℐ′ out′*. *INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv′* (*c r*))
   by(*simp add*: *INF-UNION*)
  also have ... ≤ (*INF r′*∈*responses-ℐ ℐ′ out′*. *expectation-gpv 0 ℐ′* (λ(*r′*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*c r′*) *s′*)) (*rpv r′*))
  proof(*rule INF-mono*, *rule bexI*)
   fix *r′*
   assume *r′*: *r′* ∈ *responses-ℐ ℐ′ out′*
  have (*INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv′* (*c r*)) ≤ (*INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c r*) *s′*))
    using *IO IO′ step.prems out results-callee*[*OF out*, *of s*] *r′*
     by(*auto intro*!: *INF-mono rev-bexI step.IH dest*: *WT-gpv-ContD intro*: *results-gpv.IO*)
   also have ... ≤ *expectation-gpv 0 ℐ′* (λ(*r′*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*c r′*) *s′*)) (*rpv r′*)
    unfolding *expectation-gpv2-def* **using** *plossless-gpv-ContD*[*OF callee*, *OF out* ‹*I s*› *IO′ r′*] *WT-callee*[*OF out* ‹*I s*›] *IO′ r′*
    by(*intro plossless-INF-le-expectation-gpv*)(*auto intro*: *WT-gpv-ContD*)
   finally show (*INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv′* (*c r*)) ≤ ... .
  qed
  finally show (*INF r*∈*responses-ℐ ℐ out*. *expectation-gpv′* (*c r*)) ≤ ... .
 qed
 also note *calculation* **}**
 then show *?case* **unfolding** *expectation-gpv2-def*
 apply(*rewrite expectation-gpv.simps*)
 apply(*rewrite inline-sel*)
 apply(*simp add*: *o-def pmf-map-spmf-None*)
 apply(*rewrite sum.case-distrib*[**where** *h*=*case-generat* - -])

apply(*simp cong del*: *sum.case-cong-weak*)
apply(*simp add*: *split-beta o-def cong del*: *sum.case-cong-weak*)
apply(*rewrite inline1.simps*)
apply(*rewrite measure-spmf-bind*)
apply(*rewrite nn-integral-bind*[**where** *B*=*measure-spmf* -])
apply *simp*
apply(*simp add*: *space-subprob-algebra*)
apply(*rule nn-integral-mono-AE*)
apply(*clarsimp split*!: *generat.split*)
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return*)
apply(*rewrite measure-spmf-bind*)
apply(*simp add*: *nn-integral-bind*[**where** *B*=*measure-spmf* -] *space-subprob-algebra*)
apply(*subst generat.case-distrib*[**where** *h*=*measure-spmf*])
apply(*subst generat.case-distrib*[**where** $h=\lambda x.$ *nn-integral x* -])
apply(*simp add*: *measure-spmf-return-spmf nn-integral-return split-def*)
done
qed

lemma *plossless-inline*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.$ $[\![$ *I s*; $x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $]\!] \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*
shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
unfolding *pgen-lossless-gpv-def*
proof(*rule antisym*)
have *WT'*: $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$ **using** *WT I* **by**(*rule WT-gpv-inline-invar*)
from *expectation-gpv-const-le*[*OF WT'*, *of 0 1*]
show *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*) $\leq$ *1* **by**(*simp add*: *max-def*)

have *1* = *expectation-gpv 0* $\mathcal{I}$ (λ-. *1*) *gpv* **using** *lossless* **by**(*simp add*: *pgen-lossless-gpv-def*)
also have ... $\leq$ *expectation-gpv 0* $\mathcal{I}'$ (λ-. *1*) (*inline callee gpv s*)
by(*rule expectation-gpv-le-inline*[*unfolded split-def*]; *rule callee I WT*)
finally show *1* $\leq$ ... **.**
qed

end

lemma *expectation-left-gpv* [*simp*]:
expectation-gpv fail ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) *f* (*left-gpv gpv*) = *expectation-gpv fail* $\mathcal{I}$ *f gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv' expectation-gpv''*)
show *?case*
by (*auto simp add*: *pmf-map-spmf-None o-def case-map-generat image-comp*
split: *generat.split intro*!: *nn-integral-cong-AE INF-cong step.IH*)

qed

lemma *expectation-right-gpv* [*simp*]:
expectation-gpv fail ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) *f* (*right-gpv gpv*) = *expectation-gpv fail* $\mathcal{I}'$ *f gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv′ expectation-gpv″*)
show *?case*
by (*auto simp add*: *pmf-map-spmf-None o-def case-map-generat image-comp split*: *generat.split intro*!: *nn-integral-cong-AE INF-cong step.IH*)
qed

lemma *pgen-lossless-left-gpv* [*simp*]: *pgen-lossless-gpv fail* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*left-gpv gpv*) = *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma *pgen-lossless-right-gpv* [*simp*]: *pgen-lossless-gpv fail* ($\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}'$) (*right-gpv gpv*) = *pgen-lossless-gpv fail* $\mathcal{I}'$ *gpv*
by(*simp add*: *pgen-lossless-gpv-def*)

lemma (**in** *raw-converter-invariant*) *expectation-gpv-le-inline-invariant*:
defines *expectation-gpv2* ≡ *expectation-gpv 0* $\mathcal{I}'$
assumes *callee*: $\bigwedge$*s x*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *WT-gpv*: $\mathcal{I}$ ⊢*g gpv* √
and *I*: *I s*
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*)
using *WT-gpv I*
proof(*induction arbitrary*: *gpv s rule*: *expectation-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step expectation-gpv′*)
{ **fix** *out c*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
with *step.prems*(*1*) **have** *out*: *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **by**(*rule WT-gpv-OutD*)
have (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv′* (*c r*)) = ∫⁺ *generat*. (*INF r*∈*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv′* (*c r*)) ∂*measure-spmf* (*the-gpv* (*callee s out*))
using *WT-callee*[*OF out*, *of s*] *callee*[*OF out*, *of s*] *step.prems*(*2*)
by(*clarsimp simp add*: *measure-spmf.emeasure-eq-measure plossless-iff-colossless-pfinite colossless-gpv-lossless-spmfD lossless-weight-spmfD*)
also have ... ≤ ∫⁺ *generat*. (*case generat of Pure* (*x*, *s′*) ⇒
∫⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out′*, *callee′*, *rpv*) ⇒ *INF r′*∈*responses-*$\mathcal{I}$ $\mathcal{I}'$ *out′*. *expectation-gpv 0* $\mathcal{I}'$ (λ(*r*, *s′*). *expectation-gpv 0* $\mathcal{I}'$ (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s′*)) (*callee′ r′*))
∂*measure-spmf* (*inline1 callee* (*c x*) *s′*)

| *IO out′ rpv* ⇒ *INF r′*∈*responses-ℐ ℐ′ out′. expectation-gpv 0 ℐ′* (λ(*r′*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*c r′*) *s′*)) (*rpv r′*))
∂*measure-spmf* (*the-gpv* (*callee s out*))
proof(*rule nn-integral-mono-AE*; *simp split*!: *generat.split*)
fix *x s′*
assume *Pure*: *Pure* (*x*, *s′*) ∈ *set-spmf* (*the-gpv* (*callee s out*))
hence (*x*, *s′*) ∈ *results-gpv ℐ′* (*callee s out*) **by**(*rule results-gpv.Pure*)
with *results-callee*[*OF out step.prems*(*2*)] **have** *x*: *x* ∈ *responses-ℐ ℐ out* **and** *s′*: *I s′* **by** *blast*+
from *this*(*1*) **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′*)
by(*rule step.IH*)(*rule WT-gpv-ContD*[*OF step.prems*(*1*) *IO x*] *step.prems s′*|*assumption*)+
also have … = ∫ ⁺ *xx*. (*case xx of Inl* (*x*, -) ⇒ *f x*
| *Inr* (*out′*, *callee′*, *rpv*) ⇒ *INF r′*∈*responses-ℐ ℐ′ out′. expectation-gpv 0 ℐ′* (λ(*r*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*rpv r*) *s′*)) (*callee′ r′*))
∂*measure-spmf* (*inline1 callee* (*c x*) *s′*)
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*)
finally show (*INF r*∈*responses-ℐ ℐ out. expectation-gpv′* (*c r*)) ≤ … .
next
fix *out′ rpv*
assume *IO′*: *IO out′ rpv* ∈ *set-spmf* (*the-gpv* (*callee s out*))
have (*INF r*∈*responses-ℐ ℐ out. expectation-gpv′* (*c r*)) ≤ (*INF* (*r*, *s′*)∈(⋃ *r′*∈*responses-ℐ ℐ′ out′. results-gpv ℐ′* (*rpv r′*)). *expectation-gpv′* (*c r*))
using *IO′ results-callee*[*OF out step.prems*(*2*)] **by**(*intro INF-mono*)(*auto intro*: *results-gpv.IO*)
also have … = (*INF r′*∈*responses-ℐ ℐ′ out′. INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv′* (*c r*))
by(*simp add*: *INF-UNION*)
also have … ≤ (*INF r′*∈*responses-ℐ ℐ′ out′. expectation-gpv 0 ℐ′* (λ(*r′*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*c r′*) *s′*)) (*rpv r′*))
proof(*rule INF-mono*, *rule bexI*)
fix *r′*
assume *r′*: *r′* ∈ *responses-ℐ ℐ′ out′*
have (*INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv′* (*c r*)) ≤ (*INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee* (*c r*) *s′*))
using *IO IO′ step.prems out results-callee*[*OF out*, *of s*] *r′*
by(*auto intro*!: *INF-mono rev-bexI step.IH dest*: *WT-gpv-ContD intro*: *results-gpv.IO*)
also have … ≤ *expectation-gpv 0 ℐ′* (λ(*r′*, *s′*). *expectation-gpv 0 ℐ′* (λ(*x*, *s*). *f x*) (*inline callee* (*c r′*) *s′*)) (*rpv r′*)
unfolding *expectation-gpv2-def* **using** *plossless-gpv-ContD*[*OF callee*, *OF out step.prems*(*2*) *IO′ r′*] *WT-callee*[*OF out step.prems*(*2*)] *IO′ r′*
by(*intro plossless-INF-le-expectation-gpv*)(*auto intro*: *WT-gpv-ContD*)
finally show (*INF* (*r*, *s′*)∈*results-gpv ℐ′* (*rpv r′*). *expectation-gpv′* (*c r*)) ≤

```
... .
      qed
      finally show (INF r∈responses-ℐ ℐ out. expectation-gpv' (c r)) ≤ ... .
    qed
    also note calculation }
  then show ?case unfolding expectation-gpv2-def
    apply(rewrite expectation-gpv.simps)
    apply(rewrite inline-sel)
    apply(simp add: o-def pmf-map-spmf-None)
    apply(rewrite sum.case-distrib[where h=case-generat - -])
    apply(simp cong del: sum.case-cong-weak)
    apply(simp add: split-beta o-def cong del: sum.case-cong-weak)
    apply(rewrite inline1.simps)
    apply(rewrite measure-spmf-bind)
    apply(rewrite nn-integral-bind[where B=measure-spmf -])
      apply simp
     apply(simp add: space-subprob-algebra)
    apply(rule nn-integral-mono-AE)
    apply(clarsimp split!: generat.split)
     apply(simp add: measure-spmf-return-spmf nn-integral-return)
    apply(rewrite measure-spmf-bind)
   apply(simp add: nn-integral-bind[where B=measure-spmf -] space-subprob-algebra)
    apply(subst generat.case-distrib[where h=measure-spmf])
    apply(subst generat.case-distrib[where h=λx. nn-integral x -])
    apply(simp add: measure-spmf-return-spmf nn-integral-return split-def)
    done
qed

lemma (in raw-converter-invariant) plossless-inline-invariant:
  assumes lossless: plossless-gpv ℐ gpv
    and WT: ℐ ⊢g gpv √
    and callee: ⋀s x. ⟦ x ∈ outs-ℐ ℐ; I s ⟧ ⟹ plossless-gpv ℐ' (callee s x)
    and I: I s
  shows plossless-gpv ℐ' (inline callee gpv s)
  unfolding pgen-lossless-gpv-def
proof(rule antisym)
  have WT': ℐ' ⊢g inline callee gpv s √ using WT I by(rule WT-gpv-inline-invar)
  from expectation-gpv-const-le[OF WT', of 0 1]
  show expectation-gpv 0 ℐ' (λ-. 1) (inline callee gpv s) ≤ 1 by(simp add: max-def)

  have 1 = expectation-gpv 0 ℐ (λ-. 1) gpv using lossless by(simp add: pgen-lossless-gpv-def)
  also have ... ≤ expectation-gpv 0 ℐ' (λ-. 1) (inline callee gpv s)
    by(rule expectation-gpv-le-inline[unfolded split-def]; rule callee WT WT-callee
I)
  finally show 1 ≤ ... .
qed

context callee-invariant-on begin
```

lemma *raw-converter-invariant*: *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ (λs *x*. *lift-spmf* (*callee s x*)) *I*
  by(*unfold-locales*)(*auto dest*: *callee-invariant WT-callee WT-calleeD*)

lemma (**in** *callee-invariant-on*) *plossless-exec-gpv*:
  assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
    and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
    and *callee*: $\bigwedge s\ x.$ ⟦ $x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ $\Longrightarrow$ *lossless-spmf* (*callee s x*)
    and *I*: *I s*
  shows *lossless-spmf* (*exec-gpv callee gpv s*)
proof −
  interpret *raw-converter-invariant* $\mathcal{I}$ $\mathcal{I}'$ λs *x*. *lift-spmf* (*callee s x*) *I* **for** $\mathcal{I}'$
    by(*rule raw-converter-invariant*)
  have *plossless-gpv* $\mathcal{I}$*-full* (*inline* (λs *x*. *lift-spmf* (*callee s x*)) *gpv s*)
    using *lossless WT* **by**(*rule plossless-inline*)(*simp-all add*: *callee I*)
  from *this*[*THEN plossless-gpv-lossless-spmfD*] **show** *?thesis*
    unfolding *exec-gpv-conv-inline1* **by**(*simp add*: *inline-sel*)
qed

end

lemma *expectation-gpv-mk-lossless-gpv*:
  fixes $\mathcal{I}$ *y*
  defines *rhs* $\equiv$ *expectation-gpv 0* $\mathcal{I}$ (λ-. *y*)
  assumes *WT*: $\mathcal{I}' \vdash g$ *gpv* $\surd$
    and *outs*: *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$
  shows *expectation-gpv 0* $\mathcal{I}'$ (λ-. *y*) *gpv* $\leq$ *rhs* (*mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}'$) *x gpv*)
  using *WT*
proof(*induction arbitrary*: *gpv rule*: *expectation-gpv-fixp-induct*)
  case *adm* **show** *?case* **by** *simp*
  case *bottom* **show** *?case* **by** *simp*
  case *step* [*unfolded rhs-def*]: (*step expectation-gpv'*)
  show *?case* **using** *step.prems outs* **unfolding** *rhs-def*
    apply(*subst expectation-gpv.simps*)
    apply(*clarsimp intro*!: *nn-integral-mono-AE INF-mono split*!: *generat.split if-split*)
    subgoal
      by(*frule* (*1*) *WT-gpv-OutD*)(*auto simp add*: *in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$ *intro*!: *bexI step.IH*[*unfolded rhs-def*] *dest*: *WT-gpv-ContD*)
     apply(*frule* (*1*) *WT-gpv-OutD*; *clarsimp simp add*: *in-outs-*$\mathcal{I}$*-iff-responses-*$\mathcal{I}$ *ex-in-conv*[*symmetric*])
    subgoal for *out c input input'*
      using *step.hyps*[*of c input'*] *expectation-gpv-const-le*[*of* $\mathcal{I}'$ *c input' 0 y*]
      by− (*drule* (*2*) *WT-gpv-ContD*, *fastforce intro*: *rev-bexI simp add*: *max-def*)
    done
qed

lemma *plossless-gpv-mk-lossless-gpv*:
  assumes *plossless-gpv* $\mathcal{I}$ *gpv*

and $\mathcal{I} \vdash g$ *gpv* $\surd$
and *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$
shows *plossless-gpv* $\mathcal{I}'$ (*mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}$) *x gpv*)
using *assms expectation-gpv-mk-lossless-gpv*[*OF assms*(*2*), *of* $\mathcal{I}'$ *1 x*]
unfolding *pgen-lossless-gpv-def*
by −(*rule antisym*[*OF expectation-gpv-const-le*[*THEN order-trans*]]; *simp add*: *WT-gpv-mk-lossless-gpv*)

lemma (**in** *callee-invariant-on*) *exec-gpv-mk-lossless-gpv*:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I s*
shows *exec-gpv callee* (*mk-lossless-gpv* (*responses-*$\mathcal{I}$ $\mathcal{I}$) *x gpv*) *s* = *exec-gpv callee gpv s*
using *assms*
proof(*induction arbitrary*: *gpv s rule*: *exec-gpv-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv'*)
show *?case* **using** *step.prems WT-gpv-OutD*[*OF step.prems*(*1*)]
by(*clarsimp simp add*: *bind-map-spmf intro*!: *bind-spmf-cong*[*OF refl*] *split*!: *generat.split if-split*)
(*force intro*!: *step.IH dest*: *WT-callee*[*THEN WT-calleeD*] *WT-gpv-OutD callee-invariant WT-gpv-ContD*)+
qed

lemma *expectation-gpv-map-gpv'* [*simp*]:
expectation-gpv fail $\mathcal{I}$ *f* (*map-gpv' g h k gpv*) =
expectation-gpv fail (*map-*$\mathcal{I}$ *h k* $\mathcal{I}$) (*f* ∘ *g*) *gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions expectation-gpv.mono expectation-gpv.mono expectation-gpv-def expectation-gpv-def*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exp1 exp2*)
have *pmf* (*the-gpv* (*map-gpv' g h k gpv*)) *None* = *pmf* (*the-gpv gpv*) *None*
by(*simp add*: *pmf-map-spmf-None*)
then show *?case*
by *simp*
(*auto simp add*: *nn-integral-measure-spmf step.IH image-comp*
split: *generat.split intro*!: *nn-integral-cong*)
qed

lemma *plossless-gpv-map-gpv'* [*simp*]:
pgen-lossless-gpv b $\mathcal{I}$ (*map-gpv' f g h gpv*) ⟷ *pgen-lossless-gpv b* (*map-*$\mathcal{I}$ *g h* $\mathcal{I}$) *gpv*
unfolding *pgen-lossless-gpv-def* **by**(*simp add*: *o-def*)

end

```
theory GPV-Bisim imports
  GPV-Expectation
begin
```

6.6 Bisimulation for oracles

Bisimulation is a consequence of parametricity

```
lemma exec-gpv-oracle-bisim':
  assumes *: X s1 s2
  and bisim: ⋀s1 s2 x. X s1 s2 ⟹ rel-spmf (λ(a, s1') (b, s2'). a = b ∧ X s1'
s2') (oracle1 s1 x) (oracle2 s2 x)
  shows rel-spmf (λ(a, s1') (b, s2'). a = b ∧ X s1' s2') (exec-gpv oracle1 gpv s1)
(exec-gpv oracle2 gpv s2)
by(rule exec-gpv-parametric[of X (=) (=), unfolded gpv.rel-eq rel-prod-conv, THEN
rel-funD, THEN rel-funD, THEN rel-funD, OF rel-funI refl, OF rel-funI *])(simp
add: bisim)

lemma exec-gpv-oracle-bisim:
  assumes *: X s1 s2
  and bisim: ⋀s1 s2 x. X s1 s2 ⟹ rel-spmf (λ(a, s1') (b, s2'). a = b ∧ X s1'
s2') (oracle1 s1 x) (oracle2 s2 x)
  and R: ⋀x s1' s2'. ⟦ X s1' s2'; (x, s1') ∈ set-spmf (exec-gpv oracle1 gpv s1);
(x, s2') ∈ set-spmf (exec-gpv oracle2 gpv s2) ⟧ ⟹ R (x, s1') (x, s2')
  shows rel-spmf R (exec-gpv oracle1 gpv s1) (exec-gpv oracle2 gpv s2)
apply(rule spmf-rel-mono-strong)
apply(rule exec-gpv-oracle-bisim'[OF * bisim])
apply(auto dest: R)
done

lemma run-gpv-oracle-bisim:
  assumes X s1 s2
  and ⋀s1 s2 x. X s1 s2 ⟹ rel-spmf (λ(a, s1') (b, s2'). a = b ∧ X s1' s2')
(oracle1 s1 x) (oracle2 s2 x)
  shows run-gpv oracle1 gpv s1 = run-gpv oracle2 gpv s2
using exec-gpv-oracle-bisim'[OF assms]
by(fold spmf-rel-eq)(fastforce simp add: spmf-rel-map intro: rel-spmf-mono)

context
  fixes joint-oracle :: ('s1 × 's2) ⇒ 'a ⇒ (('b × 's1) × ('b × 's2)) spmf
  and oracle1 :: 's1 ⇒ 'a ⇒ ('b × 's1) spmf
  and bad1 :: 's1 ⇒ bool
  and oracle2 :: 's2 ⇒ 'a ⇒ ('b × 's2) spmf
  and bad2 :: 's2 ⇒ bool
begin

partial-function (spmf) exec-until-bad :: ('x, 'a, 'b) gpv ⇒ 's1 ⇒ 's2 ⇒ (('x ×
's1) × ('x × 's2)) spmf
```

where
exec-until-bad gpv s1 s2 =
(*if bad1 s1* ∨ *bad2 s2 then pair-spmf* (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
else bind-spmf (*the-gpv gpv*) (λ*generat*.
case generat of Pure x ⇒ *return-spmf* ((*x*, *s1*), (*x*, *s2*))
| *IO out f* ⇒ *bind-spmf* (*joint-oracle* (*s1*, *s2*) *out*) (λ((*x*, *s1'*), (*y*, *s2'*)).
if bad1 s1' ∨ *bad2 s2' then pair-spmf* (*exec-gpv oracle1* (*f x*) *s1'*) (*exec-gpv oracle2* (*f y*) *s2'*)
else exec-until-bad (*f x*) *s1' s2'*)))

lemma *exec-until-bad-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*f*. *P* (λ*gpv s1 s2*. *f* ((*gpv*, *s1*), *s2*)))
and *P* (λ- - -. *return-pmf None*)
and ⋀*exec-until-bad'*. *P exec-until-bad'* ⟹
P (λ*gpv s1 s2*. *if bad1 s1* ∨ *bad2 s2 then pair-spmf* (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
else bind-spmf (*the-gpv gpv*) (λ*generat*.
case generat of Pure x ⇒ *return-spmf* ((*x*, *s1*), (*x*, *s2*))
| *IO out f* ⇒ *bind-spmf* (*joint-oracle* (*s1*, *s2*) *out*) (λ((*x*, *s1'*), (*y*, *s2'*)).
if bad1 s1' ∨ *bad2 s2' then pair-spmf* (*exec-gpv oracle1* (*f x*) *s1'*) (*exec-gpv oracle2* (*f y*) *s2'*)
else exec-until-bad' (*f x*) *s1' s2'*)))
shows *P exec-until-bad*
using *assms* **by**(*rule exec-until-bad.fixp-induct*[*unfolded curry-conv*[*abs-def*]])

end

lemma *exec-gpv-oracle-bisim-bad-plossless*:
fixes *s1* :: *'s1* **and** *s2* :: *'s2* **and** *X* :: *'s1* ⇒ *'s2* ⇒ *bool*
and *oracle1* :: *'s1* ⇒ *'a* ⇒ (*'b* × *'s1*) *spmf*
and *oracle2* :: *'s2* ⇒ *'a* ⇒ (*'b* × *'s2*) *spmf*
assumes ∗: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and *bad*: *bad1 s1* = *bad2 s2*
and *bisim*: ⋀*s1 s2 x*. ⟦ *X s1 s2*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1'* = *bad2 s2'* ∧ (*if bad2 s2' then X-bad s1' s2' else a* = *b* ∧ *X s1' s2'*)) (*oracle1 s1 x*) (*oracle2 s2 x*)
and *bad-sticky1*: ⋀*s2*. *bad2 s2* ⟹ *callee-invariant-on oracle1* (λ*s1*. *bad1 s1* ∧ *X-bad s1 s2*) *ℐ*
and *bad-sticky2*: ⋀*s1*. *bad1 s1* ⟹ *callee-invariant-on oracle2* (λ*s2*. *bad2 s2* ∧ *X-bad s1 s2*) *ℐ*
and *lossless1*: ⋀*s1 x*. ⟦ *bad1 s1*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *lossless-spmf* (*oracle1 s1 x*)
and *lossless2*: ⋀*s2 x*. ⟦ *bad2 s2*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *lossless-spmf* (*oracle2 s2 x*)
and *lossless*: *plossless-gpv ℐ gpv*
and *WT-oracle1*: ⋀*s1*. *ℐ* ⊢c *oracle1 s1* √
and *WT-oracle2*: ⋀*s2*. *ℐ* ⊢c *oracle2 s2* √
and *WT-gpv*: *ℐ* ⊢g *gpv* √
shows *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1'* = *bad2 s2'* ∧ (*if bad2 s2' then X-bad*

s1′ s2′ else a = b ∧ X s1′ s2′)) (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
(**is** *rel-spmf ?R ?p ?q*)
proof −
let *?R′ = λ(a, s1′) (b, s2′). bad1 s1′ = bad2 s2′ ∧ (if bad2 s2′ then X-bad s1′ s2′ else a = b ∧ X s1′ s2′)*
from *bisim* **have** *∀ s1 s2. ∀ x ∈ outs-ℐ ℐ. X s1 s2 ⟶ rel-spmf ?R′ (oracle1 s1 x) (oracle2 s2 x)* **by** *blast*
then obtain *joint-oracle*
where *oracle1* [*symmetric*]: *⋀s1 s2 x. ⟦ X s1 s2; x ∈ outs-ℐ ℐ ⟧ ⟹ map-spmf fst (joint-oracle s1 s2 x) = oracle1 s1 x*
and *oracle2* [*symmetric*]: *⋀s1 s2 x. ⟦ X s1 s2; x ∈ outs-ℐ ℐ ⟧ ⟹ map-spmf snd (joint-oracle s1 s2 x) = oracle2 s2 x*
and *3* [*rotated 2*]: *⋀s1 s2 x y y′ s1′ s2′. ⟦ X s1 s2; x ∈ outs-ℐ ℐ; ((y, s1′), (y′, s2′)) ∈ set-spmf (joint-oracle s1 s2 x) ⟧*
⟹ bad1 s1′ = bad2 s2′ ∧ (if bad2 s2′ then X-bad s1′ s2′ else y = y′ ∧ X s1′ s2′)
apply *atomize-elim*
apply(*unfold rel-spmf-simps all-conj-distrib*[*symmetric*] *all-simps*(*6*) *imp-conjR*[*symmetric*])
apply(*subst choice-iff*[*symmetric*] *ex-simps*(*6*))+
apply *fastforce*
done
let *?joint-oracle = λ(s1, s2). joint-oracle s1 s2*
let *?pq = exec-until-bad ?joint-oracle oracle1 bad1 oracle2 bad2 gpv s1 s2*

have *setD*: *⋀s1 s2 x y y′ s1′ s2′. ⟦ X s1 s2; x ∈ outs-ℐ ℐ; ((y, s1′), (y′, s2′)) ∈ set-spmf (joint-oracle s1 s2 x) ⟧*
⟹ (y, s1′) ∈ set-spmf (oracle1 s1 x) ∧ (y′, s2′) ∈ set-spmf (oracle2 s2 x)
unfolding *oracle1 oracle2* **by**(*auto intro*: *rev-image-eqI*)
show *?thesis*
proof
show *map-spmf fst ?pq = exec-gpv oracle1 gpv s1*
proof(*rule spmf.leq-antisym*)
show *ord-spmf* (=) (*map-spmf fst ?pq*) (*exec-gpv oracle1 gpv s1*) **using** ∗ *bad WT-gpv lossless*
proof(*induction arbitrary*: *s1 s2 gpv rule*: *exec-until-bad-fixp-induct*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-until-bad′*)
show *?case*
proof(*cases bad2 s2*)
case *True*
then have *weight-spmf (exec-gpv oracle2 gpv s2) = 1*
using *callee-invariant-on.weight-exec-gpv*[*OF bad-sticky2 lossless2, of s1 gpv s2*]
step.prems weight-spmf-le-1[*of exec-gpv oracle2 gpv s2*]
by(*simp add*: *pgen-lossless-gpv-def weight-gpv-def*)
then show *?thesis* **using** *True* **by** *simp*
next

case *False*
hence ¬ *bad1 s1* **using** *step.prems*(*2*) **by** *simp*
moreover {
fix *out c r1 s1′ r2 s2′*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *joint*: ((*r1*, *s1′*), (*r2*, *s2′*)) ∈ *set-spmf* (*joint-oracle s1 s2 out*)
from *step.prems*(*3*) *IO* **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpvD*)
from *setD*[*OF - out joint*] *step.prems*(*1*) *False*
have *1*: (*r1*, *s1′*) ∈ *set-spmf* (*oracle1 s1 out*)
and *2*: (*r2*, *s2′*) ∈ *set-spmf* (*oracle2 s2 out*) **by** *simp-all*
hence *r1*: *r1* ∈ *responses-ℐ ℐ out* **and** *r2*: *r2* ∈ *responses-ℐ ℐ out*
using *WT-oracle1 WT-oracle2 out* **by**(*blast dest*: *WT-calleeD*)+
have ∗: *plossless-gpv ℐ* (*c r2*) **using** *step.prems*(*4*) *IO r2 step.prems*(*3*)
by(*rule plossless-gpv-ContD*)
then have *bad2 s2′* ⟹ *weight-spmf* (*exec-gpv oracle2* (*c r2*) *s2′*) = *1*
and ¬ *bad2 s2′* ⟹ *ord-spmf* (=) (*map-spmf fst* (*exec-until-bad′* (*c r2*) *s1′ s2′*)) (*exec-gpv oracle1* (*c r2*) *s1′*)
using *callee-invariant-on.weight-exec-gpv*[*OF bad-sticky2 lossless2*, *of s1′ c r2 s2′*]
weight-spmf-le-1[*of exec-gpv oracle2* (*c r2*) *s2′*] *WT-gpv-ContD*[*OF step.prems*(*3*) *IO r2*]
3[*OF joint - out*] *step.prems*(*1*) *False*
by(*simp-all add*: *pgen-lossless-gpv-def weight-gpv-def step.IH*) }
ultimately show *?thesis* **using** *False step.prems*(*1*)
by(*rewrite* **in** *ord-spmf - -* ⌑ *exec-gpv.simps*)
(*fastforce simp add*: *split-def bind-map-spmf map-spmf-bind-spmf oracle1 WT-gpv-OutD*[*OF step.prems*(*3*)] *intro*!: *ord-spmf-bind-reflI split*!: *generat.split dest*: *3*)
qed
qed
show *ord-spmf* (=) (*exec-gpv oracle1 gpv s1*) (*map-spmf fst ?pq*) **using** ∗ *bad WT-gpv lossless*
proof(*induction arbitrary*: *gpv s1 s2 rule*: *exec-gpv-fixp-induct-strong*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv′*)
then show *?case*
proof(*cases bad2 s2*)
case *True*
then have *weight-spmf* (*exec-gpv oracle2 gpv s2*) = *1*
using *callee-invariant-on.weight-exec-gpv*[*OF bad-sticky2 lossless2*, *of s1 gpv s2*]
step.prems weight-spmf-le-1[*of exec-gpv oracle2 gpv s2*]
by(*simp add*: *pgen-lossless-gpv-def weight-gpv-def*)
then show *?thesis* **using** *True*
by(*rewrite exec-until-bad.simps*; *rewrite exec-gpv.simps*)
(*clarsimp intro*!: *ord-spmf-bind-reflI split*!: *generat.split simp add*: *step.hyps*)
next

```
        case False
        hence ¬ bad1 s1 using step.prems(2) by simp
        moreover {
          fix out c r1 s1' r2 s2'
          assume IO: IO out c ∈ set-spmf (the-gpv gpv)
            and joint: ((r1, s1'), (r2, s2')) ∈ set-spmf (joint-oracle s1 s2 out)
          from step.prems(3) IO have out: out ∈ outs-ℐ ℐ by(rule WT-gpvD)
          from setD[OF - out joint] step.prems(1) False
          have 1: (r1, s1') ∈ set-spmf (oracle1 s1 out)
            and 2: (r2, s2') ∈ set-spmf (oracle2 s2 out) by simp-all
          hence r1: r1 ∈ responses-ℐ ℐ out and r2: r2 ∈ responses-ℐ ℐ out
            using WT-oracle1 WT-oracle2 out by(blast dest: WT-calleeD)+
          have *: plossless-gpv ℐ (c r2) using step.prems(4) IO r2 step.prems(3)
            by(rule plossless-gpv-ContD)
          then have bad2 s2' ⟹ weight-spmf (exec-gpv oracle2 (c r2) s2') = 1
              and ¬ bad2 s2' ⟹ ord-spmf (=) (exec-gpv' (c r2) s1') (map-spmf
fst (exec-until-bad (λ(x, y). joint-oracle x y) oracle1 bad1 oracle2 bad2 (c r2) s1'
s2'))
             using callee-invariant-on.weight-exec-gpv[OF bad-sticky2 lossless2, of
s1' c r2 s2']
               weight-spmf-le-1[of exec-gpv oracle2 (c r2) s2'] WT-gpv-ContD[OF
step.prems(3) IO r2]
             3[OF joint - out] step.prems(1) False
            by(simp-all add: pgen-lossless-gpv-def weight-gpv-def step.IH) }
        ultimately show ?thesis using False step.prems(1)
          by(rewrite exec-until-bad.simps)
         (fastforce simp add: map-spmf-bind-spmf WT-gpv-OutD[OF step.prems(3)]
oracle1 bind-map-spmf step.hyps intro!: ord-spmf-bind-reflI split!: generat.split dest:
3)
      qed
    qed
  qed

  show map-spmf snd ?pq = exec-gpv oracle2 gpv s2
  proof(rule spmf.leq-antisym)
     show ord-spmf (=) (map-spmf snd ?pq) (exec-gpv oracle2 gpv s2) using *
bad WT-gpv lossless
    proof(induction arbitrary: s1 s2 gpv rule: exec-until-bad-fixp-induct)
      case adm show ?case by simp
      case bottom show ?case by simp
      case (step exec-until-bad')
      show ?case
      proof(cases bad2 s2)
        case True
        then have weight-spmf (exec-gpv oracle1 gpv s1) = 1
          using callee-invariant-on.weight-exec-gpv[OF bad-sticky1 lossless1, of s2
gpv s1]
            step.prems weight-spmf-le-1[of exec-gpv oracle1 gpv s1]
          by(simp add: pgen-lossless-gpv-def weight-gpv-def)
```

then show *?thesis* **using** *True* **by** *simp*
next
case *False*
hence ¬ *bad1 s1* **using** *step.prems*(*2*) **by** *simp*
moreover {
fix *out c r1 s1′ r2 s2′*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *joint*: ((*r1*, *s1′*), (*r2*, *s2′*)) ∈ *set-spmf* (*joint-oracle s1 s2 out*)
from *step.prems*(*3*) *IO* **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpvD*)
from *setD*[*OF - out joint*] *step.prems*(*1*) *False*
have *1*: (*r1*, *s1′*) ∈ *set-spmf* (*oracle1 s1 out*)
and *2*: (*r2*, *s2′*) ∈ *set-spmf* (*oracle2 s2 out*) **by** *simp-all*
hence *r1*: *r1* ∈ *responses-ℐ ℐ out* **and** *r2*: *r2* ∈ *responses-ℐ ℐ out*
using *WT-oracle1 WT-oracle2 out* **by**(*blast dest*: *WT-calleeD*)+
have ∗: *plossless-gpv ℐ* (*c r1*) **using** *step.prems*(*4*) *IO r1 step.prems*(*3*)
by(*rule plossless-gpv-ContD*)
then have *bad2 s2′* ⟹ *weight-spmf* (*exec-gpv oracle1* (*c r1*) *s1′*) = *1*
and ¬ *bad2 s2′* ⟹ *ord-spmf* (=) (*map-spmf snd* (*exec-until-bad′* (*c r2*) *s1′ s2′*)) (*exec-gpv oracle2* (*c r2*) *s2′*)
using *callee-invariant-on.weight-exec-gpv*[*OF bad-sticky1 lossless1*, *of s2′ c r1 s1′*]
weight-spmf-le-1[*of exec-gpv oracle1* (*c r1*) *s1′*] *WT-gpv-ContD*[*OF step.prems*(*3*) *IO r1*]
3[*OF joint - out*] *step.prems*(*1*) *False*
by(*simp-all add*: *pgen-lossless-gpv-def weight-gpv-def step.IH*) **}**
ultimately show *?thesis* **using** *False step.prems*(*1*)
by(*rewrite* **in** *ord-spmf - -* ⌑ *exec-gpv.simps*)
(*fastforce simp add*: *split-def bind-map-spmf map-spmf-bind-spmf oracle2 WT-gpv-OutD*[*OF step.prems*(*3*)] *intro*!: *ord-spmf-bind-reflI split*!: *generat.split dest*: *3*)
qed
qed
show *ord-spmf* (=) (*exec-gpv oracle2 gpv s2*) (*map-spmf snd ?pq*) **using** ∗ *bad WT-gpv lossless*
proof(*induction arbitrary*: *gpv s1 s2 rule*: *exec-gpv-fixp-induct-strong*)
case *adm* **show** *?case* **by** *simp*
case *bottom* **show** *?case* **by** *simp*
case (*step exec-gpv′*)
then show *?case*
proof(*cases bad2 s2*)
case *True*
then have *weight-spmf* (*exec-gpv oracle1 gpv s1*) = *1*
using *callee-invariant-on.weight-exec-gpv*[*OF bad-sticky1 lossless1*, *of s2 gpv s1*]
step.prems weight-spmf-le-1[*of exec-gpv oracle1 gpv s1*]
by(*simp add*: *pgen-lossless-gpv-def weight-gpv-def*)
then show *?thesis* **using** *True*
by(*rewrite exec-until-bad.simps*; *subst* (*2*) *exec-gpv.simps*)
(*clarsimp intro*!: *ord-spmf-bind-reflI split*!: *generat.split simp add*:

step.hyps)
next
case *False*
hence ¬ *bad1 s1* **using** *step.prems*(*2*) **by** *simp*
moreover {
fix *out c r1 s1′ r2 s2′*
assume *IO*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *joint*: ((*r1*, *s1′*), (*r2*, *s2′*)) ∈ *set-spmf* (*joint-oracle s1 s2 out*)
from *step.prems*(*3*) *IO* **have** *out*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpvD*)
from *setD*[*OF - out joint*] *step.prems*(*1*) *False*
have *1*: (*r1*, *s1′*) ∈ *set-spmf* (*oracle1 s1 out*)
and *2*: (*r2*, *s2′*) ∈ *set-spmf* (*oracle2 s2 out*) **by** *simp-all*
hence *r1*: *r1* ∈ *responses-ℐ ℐ out* **and** *r2*: *r2* ∈ *responses-ℐ ℐ out*
using *WT-oracle1 WT-oracle2 out* **by**(*blast dest*: *WT-calleeD*)+
have ∗: *plossless-gpv ℐ* (*c r1*) **using** *step.prems*(*4*) *IO r1 step.prems*(*3*)
by(*rule plossless-gpv-ContD*)
then have *bad2 s2′* ⟹ *weight-spmf* (*exec-gpv oracle1* (*c r1*) *s1′*) = *1*
and ¬ *bad2 s2′* ⟹ *ord-spmf* (=) (*exec-gpv′* (*c r2*) *s2′*) (*map-spmf snd* (*exec-until-bad* (λ(*x*, *y*). *joint-oracle x y*) *oracle1 bad1 oracle2 bad2* (*c r2*) *s1′ s2′*))
using *callee-invariant-on.weight-exec-gpv*[*OF bad-sticky1 lossless1*, *of s2′ c r1 s1′*]
weight-spmf-le-1[*of exec-gpv oracle1* (*c r1*) *s1′*] *WT-gpv-ContD*[*OF step.prems*(*3*) *IO r1*]
3[*OF joint - out*] *step.prems*(*1*) *False*
by(*simp-all add*: *pgen-lossless-gpv-def step.IH weight-gpv-def*) }
ultimately show *?thesis* **using** *False step.prems*(*1*)
by(*rewrite exec-until-bad.simps*)
(*fastforce simp add*: *map-spmf-bind-spmf WT-gpv-OutD*[*OF step.prems*(*3*)] *oracle2 bind-map-spmf step.hyps intro*!: *ord-spmf-bind-reflI split*!: *generat.split dest*: *3*)
qed
qed
qed

have *set-spmf ?pq* ⊆ {(*as1*, *bs2*). *?R′ as1 bs2*} **using** ∗ *bad WT-gpv*
proof(*induction arbitrary*: *gpv s1 s2 rule*: *exec-until-bad-fixp-induct*)
case *adm* **show** *?case* **by**(*intro cont-intro ccpo-class.admissible-leI*)
case *bottom* **show** *?case* **by** *simp*
case *step*
have *switch*: *set-spmf* (*exec-gpv oracle1* (*c r1*) *s1′*) × *set-spmf* (*exec-gpv oracle2* (*c r2*) *s2′*)
⊆ {((*a*, *s1′*), *b*, *s2′*). *bad1 s1′* = *bad2 s2′* ∧ (*if bad2 s2′ then X-bad s1′ s2′ else a* = *b* ∧ *X s1′ s2′*)}
if ¬ *bad1 s1 ℐ* ⊢*g gpv* √ ¬ *bad2 s2* **and** *X*: *X s1 s2* **and** *out*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*)
and *joint*: ((*r1*, *s1′*), (*r2*, *s2′*)) ∈ *set-spmf* (*joint-oracle s1 s2 out*)
and *bad2*: *bad2 s2′*
for *out c r1 s1′ r2 s2′*

proof(*clarify*; *rule conjI*)
from *step.prems*(*3*) *out* **have** *outs*: *out* ∈ *outs-ℐ ℐ* **by**(*rule WT-gpv-OutD*)
from *bad2 3*[*OF joint X this*] **have** *bad1*: *bad1 s1′* ∧ *X-bad s1′ s2′* **by** *simp-all*

have *s1′*: (*r1*, *s1′*) ∈ *set-spmf* (*oracle1 s1 out*) **and** *s2′*: (*r2*, *s2′*) ∈ *set-spmf* (*oracle2 s2 out*)
using *setD*[*OF X outs joint*] **by** *simp-all*
have *resp*: *r1* ∈ *responses-ℐ ℐ out* **using** *WT-oracle1 s1′ outs* **by**(*rule WT-calleeD*)
with *step.prems*(*3*) *out* **have** *WT1*: *ℐ ⊢g c r1* √ **by**(*rule WT-gpv-ContD*)
have *resp*: *r2* ∈ *responses-ℐ ℐ out* **using** *WT-oracle2 s2′ outs* **by**(*rule WT-calleeD*)
with *step.prems*(*3*) *out* **have** *WT2*: *ℐ ⊢g c r2* √ **by**(*rule WT-gpv-ContD*)

fix *r1′ s1″ r2′ s2″*
assume *s1″*: (*r1′*, *s1″*) ∈ *set-spmf* (*exec-gpv oracle1* (*c r1*) *s1′*)
and *s2″*: (*r2′*, *s2″*) ∈ *set-spmf* (*exec-gpv oracle2* (*c r2*) *s2′*)
have ∗: *bad1 s1″* ∧ *X-bad s1″ s2′* **using** *bad2 s1″ bad1 WT1*
by(*rule callee-invariant-on.exec-gpv-invariant*[*OF bad-sticky1*])
have *bad2 s2″* ∧ *X-bad s1″ s2″* **using** - *s2″* - *WT2*
by(*rule callee-invariant-on.exec-gpv-invariant*[*OF bad-sticky2*])(*simp-all add*: *bad2* ∗)
then show *bad1 s1″* = *bad2 s2″ if bad2 s2″ then X-bad s1″ s2″ else r1′* = *r2′* ∧ *X s1″ s2″*
using ∗ **by**(*simp-all*)
qed
show *?case* **using** *step.prems*
apply(*clarsimp simp add*: *bind-UNION step.IH 3 WT-gpv-OutD WT-gpv-ContD del*: *subsetI intro*!: *UN-least split*: *generat.split if-split-asm*)
subgoal by(*auto 4 3 dest*: *callee-invariant-on.exec-gpv-invariant*[*OF bad-sticky1*, *rotated*] *callee-invariant-on.exec-gpv-invariant*[*OF bad-sticky2*, *rotated*] *3*)
apply(*intro strip conjI*)
subgoal by(*drule* (*6*) *switch*) *auto*
subgoal by(*auto 4 3 intro*!: *step.IH*[*THEN order.trans*] *del*: *subsetI dest*: *3 setD*[*rotated 2*] *simp add*: *WT-gpv-OutD WT-gpv-ContD intro*: *WT-gpv-ContD intro*!: *WT-calleeD*[*OF WT-oracle1*])
done
qed
then show ⋀*x y*. (*x*, *y*) ∈ *set-spmf ?pq* ⟹ *?R x y* **by** *auto*
qed
qed

lemma *exec-gpv-oracle-bisim-bad′*:
fixes *s1* :: *′s1* **and** *s2* :: *′s2* **and** *X* :: *′s1* ⇒ *′s2* ⇒ *bool*
and *oracle1* :: *′s1* ⇒ *′a* ⇒ (*′b* × *′s1*) *spmf*
and *oracle2* :: *′s2* ⇒ *′a* ⇒ (*′b* × *′s2*) *spmf*
assumes ∗: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and *bad*: *bad1 s1* = *bad2 s2*

and *bisim*: $\bigwedge s1\ s2\ x.\ [\![\ X\ s1\ s2;\ x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow rel\text{-}spmf\ (\lambda(a,\ s1')\ (b,\ s2').$ $bad1\ s1' = bad2\ s2' \wedge (if\ bad2\ s2'\ then\ X\text{-}bad\ s1'\ s2'\ else\ a = b \wedge X\ s1'\ s2'))$ $(oracle1\ s1\ x)\ (oracle2\ s2\ x)$
and *bad-sticky1*: $\bigwedge s2.\ bad2\ s2 \Longrightarrow callee\text{-}invariant\text{-}on\ oracle1\ (\lambda s1.\ bad1\ s1 \wedge X\text{-}bad\ s1\ s2)\ \mathcal{I}$
and *bad-sticky2*: $\bigwedge s1.\ bad1\ s1 \Longrightarrow callee\text{-}invariant\text{-}on\ oracle2\ (\lambda s2.\ bad2\ s2 \wedge X\text{-}bad\ s1\ s2)\ \mathcal{I}$
and *lossless1*: $\bigwedge s1\ x.\ [\![\ bad1\ s1;\ x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow lossless\text{-}spmf\ (oracle1\ s1\ x)$
and *lossless2*: $\bigwedge s2\ x.\ [\![\ bad2\ s2;\ x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow lossless\text{-}spmf\ (oracle2\ s2\ x)$
and *lossless*: *lossless-gpv* $\mathcal{I}$ *gpv*
and *WT-oracle1*: $\bigwedge s1.\ \mathcal{I} \vdash c\ oracle1\ s1\ \surd$
and *WT-oracle2*: $\bigwedge s2.\ \mathcal{I} \vdash c\ oracle2\ s2\ \surd$
and *WT-gpv*: $\mathcal{I} \vdash g\ gpv\ \surd$
shows $rel\text{-}spmf\ (\lambda(a,\ s1')\ (b,\ s2').\ bad1\ s1' = bad2\ s2' \wedge (if\ bad2\ s2'\ then\ X\text{-}bad\ s1'\ s2'\ else\ a = b \wedge X\ s1'\ s2'))\ (exec\text{-}gpv\ oracle1\ gpv\ s1)\ (exec\text{-}gpv\ oracle2\ gpv\ s2)$
using *assms*$(1-7)$ *lossless-imp-plossless-gpv*[*OF lossless WT-gpv*] *assms*$(9-)$
by(*rule exec-gpv-oracle-bisim-bad-plossless*)

lemma *exec-gpv-oracle-bisim-bad-invariant*:
fixes $s1 :: 's1$ **and** $s2 :: 's2$ **and** $X :: 's1 \Rightarrow 's2 \Rightarrow bool$ **and** $I1 :: 's1 \Rightarrow bool$ **and** $I2 :: 's2 \Rightarrow bool$
and $oracle1 :: 's1 \Rightarrow 'a \Rightarrow ('b \times 's1)\ spmf$
and $oracle2 :: 's2 \Rightarrow 'a \Rightarrow ('b \times 's2)\ spmf$
assumes $*$: $if\ bad2\ s2\ then\ X\text{-}bad\ s1\ s2\ else\ X\ s1\ s2$
and *bad*: $bad1\ s1 = bad2\ s2$
and *bisim*: $\bigwedge s1\ s2\ x.\ [\![\ X\ s1\ s2;\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I1\ s1;\ I2\ s2\ ]\!] \Longrightarrow rel\text{-}spmf\ (\lambda(a,\ s1')\ (b,\ s2').\ bad1\ s1' = bad2\ s2' \wedge (if\ bad2\ s2'\ then\ X\text{-}bad\ s1'\ s2'\ else\ a = b \wedge X\ s1'\ s2'))\ (oracle1\ s1\ x)\ (oracle2\ s2\ x)$
and *bad-sticky1*: $\bigwedge s2.\ [\![\ bad2\ s2;\ I2\ s2\ ]\!] \Longrightarrow callee\text{-}invariant\text{-}on\ oracle1\ (\lambda s1.\ bad1\ s1 \wedge X\text{-}bad\ s1\ s2)\ \mathcal{I}$
and *bad-sticky2*: $\bigwedge s1.\ [\![\ bad1\ s1;\ I1\ s1\ ]\!] \Longrightarrow callee\text{-}invariant\text{-}on\ oracle2\ (\lambda s2.\ bad2\ s2 \wedge X\text{-}bad\ s1\ s2)\ \mathcal{I}$
and *lossless1*: $\bigwedge s1\ x.\ [\![\ bad1\ s1;\ I1\ s1;\ x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow lossless\text{-}spmf\ (oracle1\ s1\ x)$
and *lossless2*: $\bigwedge s2\ x.\ [\![\ bad2\ s2;\ I2\ s2;\ x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow lossless\text{-}spmf\ (oracle2\ s2\ x)$
and *lossless*: *lossless-gpv* $\mathcal{I}$ *gpv*
and *WT-gpv*: $\mathcal{I} \vdash g\ gpv\ \surd$
and *I1*: *callee-invariant-on oracle1 I1* $\mathcal{I}$
and *I2*: *callee-invariant-on oracle2 I2* $\mathcal{I}$
and *s1*: *I1 s1*
and *s2*: *I2 s2*
shows $rel\text{-}spmf\ (\lambda(a,\ s1')\ (b,\ s2').\ bad1\ s1' = bad2\ s2' \wedge (if\ bad2\ s2'\ then\ X\text{-}bad\ s1'\ s2'\ else\ a = b \wedge X\ s1'\ s2'))\ (exec\text{-}gpv\ oracle1\ gpv\ s1)\ (exec\text{-}gpv\ oracle2\ gpv\ s2)$
including *lifting-syntax*
proof −
interpret *I1*: *callee-invariant-on oracle1 I1* $\mathcal{I}$ **by**(*fact I1*)

interpret *I2*: *callee-invariant-on oracle2 I2 $\mathcal{I}$* **by**(*fact I2*)
from *s1* **have** *nonempty1*: {*s. I1 s*} ≠ {} **by** *auto*
{ assume ∃(*Rep1* :: *'s1' ⇒ 's1*) *Abs1*. *type-definition Rep1 Abs1* {*s. I1 s*}
and ∃(*Rep2* :: *'s2' ⇒ 's2*) *Abs2*. *type-definition Rep2 Abs2* {*s. I2 s*}
then obtain *Rep1* :: *'s1' ⇒ 's1* **and** *Abs1* **and** *Rep2* :: *'s2' ⇒ 's2* **and** *Abs2*
where *td1*: *type-definition Rep1 Abs1* {*s. I1 s*} **and** *td2*: *type-definition Rep2 Abs2* {*s. I2 s*}
by *blast*
interpret *td1*: *type-definition Rep1 Abs1* {*s. I1 s*} **by**(*rule td1*)
interpret *td2*: *type-definition Rep2 Abs2* {*s. I2 s*} **by**(*rule td2*)
define *cr1* **where** *cr1* ≡ *λx y. x = Rep1 y*
have [*transfer-rule*]: *bi-unique cr1 right-total cr1* **using** *td1 cr1-def* **by**(*rule typedef-bi-unique typedef-right-total*)+
have [*transfer-domain-rule*]: *Domainp cr1 = I1* **using** *type-definition-Domainp*[*OF td1 cr1-def*] **by** *simp*
define *cr2* **where** *cr2* ≡ *λx y. x = Rep2 y*
have [*transfer-rule*]: *bi-unique cr2 right-total cr2* **using** *td2 cr2-def* **by**(*rule typedef-bi-unique typedef-right-total*)+
have [*transfer-domain-rule*]: *Domainp cr2 = I2* **using** *type-definition-Domainp*[*OF td2 cr2-def*] **by** *simp*

let *?C = eq-onp* (*λout. out ∈ outs-$\mathcal{I}$ $\mathcal{I}$*)

define *oracle1'* **where** *oracle1'* ≡ (*Rep1 −−−> id −−−> map-spmf* (*map-prod id Abs1*)) *oracle1*
have [*transfer-rule*]: (*cr1 ===> ?C ===> rel-spmf* (*rel-prod* (=) *cr1*)) *oracle1 oracle1'*
by(*auto simp add*: *oracle1'-def rel-fun-def cr1-def spmf-rel-map prod.rel-map td1.Abs-inverse eq-onp-def intro*!: *rel-spmf-reflI intro*: *td1.Rep*[*simplified*] *dest*: *I1.callee-invariant*)
define *oracle2'* **where** *oracle2'* ≡ (*Rep2 −−−> id −−−> map-spmf* (*map-prod id Abs2*)) *oracle2*
have [*transfer-rule*]: (*cr2 ===> ?C ===> rel-spmf* (*rel-prod* (=) *cr2*)) *oracle2 oracle2'*
by(*auto simp add*: *oracle2'-def rel-fun-def cr2-def spmf-rel-map prod.rel-map td2.Abs-inverse eq-onp-def intro*!: *rel-spmf-reflI intro*: *td2.Rep*[*simplified*] *dest*: *I2.callee-invariant*)

define *s1'* **where** *s1'* ≡ *Abs1 s1*
have [*transfer-rule*]: *cr1 s1 s1'* **using** *s1* **by**(*simp add*: *cr1-def s1'-def td1.Abs-inverse*)
define *s2'* **where** *s2'* ≡ *Abs2 s2*
have [*transfer-rule*]: *cr2 s2 s2'* **using** *s2* **by**(*simp add*: *cr2-def s2'-def td2.Abs-inverse*)

define *bad1'* **where** *bad1'* ≡ (*Rep1 −−−> id*) *bad1*
have [*transfer-rule*]: (*cr1 ===>* (=)) *bad1 bad1'* **by**(*simp add*: *rel-fun-def bad1'-def cr1-def*)
define *bad2'* **where** *bad2'* ≡ (*Rep2 −−−> id*) *bad2*
have [*transfer-rule*]: (*cr2 ===>* (=)) *bad2 bad2'* **by**(*simp add*: *rel-fun-def bad2'-def cr2-def*)

define *X'* **where** *X'* ≡ (*Rep1 −−−> Rep2 −−−> id*) *X*

have [*transfer-rule*]: (*cr1* ===> *cr2* ===> (=)) *X X'* **by**(*simp add*: *rel-fun-def X'-def cr1-def cr2-def*)
define *X-bad'* **where** *X-bad'* ≡ (*Rep1* −−−> *Rep2* −−−> *id*) *X-bad*
have [*transfer-rule*]: (*cr1* ===> *cr2* ===> (=)) *X-bad X-bad'* **by**(*simp add*: *rel-fun-def X-bad'-def cr1-def cr2-def*)

define *gpv'* **where** *gpv'* ≡ *restrict-gpv* $\mathcal{I}$ *gpv*
have [*transfer-rule*]: *rel-gpv* (=) *?C gpv' gpv'*
by(*fold eq-onp-top-eq-eq*)(*auto simp add*: *gpv.rel-eq-onp eq-onp-same-args pred-gpv-def gpv'-def dest*: *in-outs'-restrict-gpvD*)

have *if bad2' s2' then X-bad' s1' s2' else X' s1' s2'* **using** ∗ **by** *transfer*
moreover have *bad1' s1'* ⟷ *bad2' s2'* **using** *bad* **by** *transfer*
moreover have *x*: *?C x x* **if** *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **for** *x* **using** *that* **by**(*simp add*: *eq-onp-def*)
have *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). (*bad1' s1'* ⟷ *bad2' s2'*) ∧ (*if bad2' s2' then X-bad' s1' s2' else a* = *b* ∧ *X' s1' s2'*)) (*oracle1' s1 x*) (*oracle2' s2 x*)
if *X' s1 s2* **and** *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **for** *s1 s2 x* **using** *that*(*1*) **supply** *that*(*2*)[*THEN x*, *transfer-rule*]
by(*transfer*)(*rule bisim*[*OF - that*(*2*)])
moreover have [*transfer-rule*]: *rel-*$\mathcal{I}$ *?C* (=) $\mathcal{I}$ $\mathcal{I}$ **by**(*rule rel-*$\mathcal{I}$*I*)(*auto simp add*: *set-relator-eq-onp eq-onp-same-args rel-set-eq dest*: *eq-onp-to-eq*)
have *callee-invariant-on oracle1'* (λ*s1*. *bad1' s1* ∧ *X-bad' s1 s2*) $\mathcal{I}$ **if** *bad2' s2* **for** *s2*
using *that* **unfolding** *callee-invariant-on-alt-def* **apply**(*transfer*)
using *bad-sticky1*[*unfolded callee-invariant-on-alt-def*] **by** *blast*
moreover have *callee-invariant-on oracle2'* (λ*s2*. *bad2' s2* ∧ *X-bad' s1 s2*) $\mathcal{I}$ **if** *bad1' s1* **for** *s1*
using *that* **unfolding** *callee-invariant-on-alt-def* **apply**(*transfer*)
using *bad-sticky2*[*unfolded callee-invariant-on-alt-def*] **by** *blast*
moreover have *lossless-spmf* (*oracle1' s1 x*) **if** *bad1' s1 x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **for** *s1 x*
using *that* **supply** *that*(*2*)[*THEN x*, *transfer-rule*] **by** *transfer*(*rule lossless1*)
moreover have *lossless-spmf* (*oracle2' s2 x*) **if** *bad2' s2 x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ **for** *s2 x*
using *that* **supply** *that*(*2*)[*THEN x*, *transfer-rule*] **by** *transfer*(*rule lossless2*)
moreover have *lossless-gpv* $\mathcal{I}$ *gpv'* **using** *WT-gpv lossless* **by**(*simp add*: *gpv'-def lossless-restrict-gpvI*)
moreover have $\mathcal{I}$ ⊢*c oracle1' s1* √ **for** *s1* **using** *I1.WT-callee* **by** *transfer*
moreover have $\mathcal{I}$ ⊢*c oracle2' s2* √ **for** *s2* **using** *I2.WT-callee* **by** *transfer*
moreover have $\mathcal{I}$ ⊢*g gpv'* √ **by**(*simp add*: *gpv'-def*)
ultimately have ∗∗: *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1' s1'* = *bad2' s2'* ∧ (*if bad2' s2' then X-bad' s1' s2' else a* = *b* ∧ *X' s1' s2'*)) (*exec-gpv oracle1' gpv' s1'*) (*exec-gpv oracle2' gpv' s2'*)
by(*rule exec-gpv-oracle-bisim-bad'*)
have [*transfer-rule*]: ((=) ===> *?C* ===> *rel-spmf* (*rel-prod* (=) (=))) *oracle2 oracle2*
((=) ===> *?C* ===> *rel-spmf* (*rel-prod* (=) (=))) *oracle1 oracle1*
by(*simp-all add*: *rel-fun-def eq-onp-def prod.rel-eq*)
note [*transfer-rule*] = *bi-unique-eq-onp bi-unique-eq*
from ∗∗ **have** *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1'* = *bad2 s2'* ∧ (*if bad2 s2'*

then X-bad s1′ s2′ else a = b ∧ X s1′ s2′)) (*exec-gpv oracle1 gpv′ s1*) (*exec-gpv oracle2 gpv′ s2*)
by(*transfer*)
also have *exec-gpv oracle1 gpv′ s1 = exec-gpv oracle1 gpv s1*
unfolding *gpv′-def* **using** *WT-gpv s1* **by**(*rule I1.exec-gpv-restrict-gpv-invariant*)
also have *exec-gpv oracle2 gpv′ s2 = exec-gpv oracle2 gpv s2*
unfolding *gpv′-def* **using** *WT-gpv s2* **by**(*rule I2.exec-gpv-restrict-gpv-invariant*)
finally have *?thesis* **. }**
from *this*[*cancel-type-definition*, *OF nonempty1*, *cancel-type-definition*] *s2* **show** *?thesis* **by** *blast*
qed

lemma *exec-gpv-oracle-bisim-bad*:
assumes ∗: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and *bad*: *bad1 s1 = bad2 s2*
and *bisim*: ⋀*s1 s2 x. X s1 s2* ⟹ *rel-spmf* (λ(*a*, *s1′*) (*b*, *s2′*). *bad1 s1′ = bad2 s2′* ∧ (*if bad2 s2′ then X-bad s1′ s2′ else a = b* ∧ *X s1′ s2′*)) (*oracle1 s1 x*) (*oracle2 s2 x*)
and *bad-sticky1*: ⋀*s2. bad2 s2* ⟹ *callee-invariant-on oracle1* (λ*s1. bad1 s1* ∧ *X-bad s1 s2*) ℐ
and *bad-sticky2*: ⋀*s1. bad1 s1* ⟹ *callee-invariant-on oracle2* (λ*s2. bad2 s2* ∧ *X-bad s1 s2*) ℐ
and *lossless1*: ⋀*s1 x. bad1 s1* ⟹ *lossless-spmf* (*oracle1 s1 x*)
and *lossless2*: ⋀*s2 x. bad2 s2* ⟹ *lossless-spmf* (*oracle2 s2 x*)
and *lossless*: *lossless-gpv* ℐ *gpv*
and *WT-oracle1*: ⋀*s1*. ℐ ⊢*c oracle1 s1* √
and *WT-oracle2*: ⋀*s2*. ℐ ⊢*c oracle2 s2* √
and *WT-gpv*: ℐ ⊢*g gpv* √
and *R*: ⋀*a s1 b s2*. ⟦ *bad1 s1 = bad2 s2*; ¬ *bad2 s2* ⟹ *a = b* ∧ *X s1 s2*; *bad2 s2* ⟹ *X-bad s1 s2* ⟧ ⟹ *R* (*a*, *s1*) (*b*, *s2*)
shows *rel-spmf R* (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
using *exec-gpv-oracle-bisim-bad′*[*OF* ∗ *bad bisim bad-sticky1 bad-sticky2 lossless1 lossless2 lossless WT-oracle1 WT-oracle2 WT-gpv*]
by(*rule rel-spmf-mono*)(*auto intro*: *R*)

lemma *exec-gpv-oracle-bisim-bad-full*:
assumes *X s1 s2*
and *bad1 s1 = bad2 s2*
and ⋀*s1 s2 x. X s1 s2* ⟹ *rel-spmf* (λ(*a*, *s1′*) (*b*, *s2′*). *bad1 s1′ = bad2 s2′* ∧ (¬ *bad2 s2′* ⟶ *a = b* ∧ *X s1′ s2′*)) (*oracle1 s1 x*) (*oracle2 s2 x*)
and *callee-invariant oracle1 bad1*
and *callee-invariant oracle2 bad2*
and ⋀*s1 x. bad1 s1* ⟹ *lossless-spmf* (*oracle1 s1 x*)
and ⋀*s2 x. bad2 s2* ⟹ *lossless-spmf* (*oracle2 s2 x*)
and *lossless-gpv* ℐ*-full gpv*
and *R*: ⋀*a s1 b s2*. ⟦ *bad1 s1 = bad2 s2*; ¬ *bad2 s2* ⟹ *a = b* ∧ *X s1 s2* ⟧ ⟹ *R* (*a*, *s1*) (*b*, *s2*)
shows *rel-spmf R* (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
using *assms*

by(*intro exec-gpv-oracle-bisim-bad*[*of bad2 s2* λ- -. *True s1 X bad1 oracle1 oracle2* $\mathcal{I}$*-full gpv R*])(*auto intro*: *rel-spmf-mono*)

lemma *max-enn2ereal*: *max* (*enn2ereal x*) (*enn2ereal y*) = *enn2ereal* (*max x y*)
including *ennreal.lifting* **unfolding** *max-def* **by** *transfer simp*

lemma *identical-until-bad*:
assumes *bad-eq*: *map-spmf bad p* = *map-spmf bad q*
and *not-bad*: *measure* (*measure-spmf* (*map-spmf* (λ*x*. (*f x*, *bad x*)) *p*)) (*A* × {*False*}) = *measure* (*measure-spmf* (*map-spmf* (λ*x*. (*f x*, *bad x*)) *q*)) (*A* × {*False*})
shows |*measure* (*measure-spmf* (*map-spmf f p*)) *A* − *measure* (*measure-spmf* (*map-spmf f q*)) *A*| ≤ *spmf* (*map-spmf bad p*) *True*
proof −
have |*enn2ereal* (*measure* (*measure-spmf* (*map-spmf f p*)) *A*) − *enn2ereal* (*measure* (*measure-spmf* (*map-spmf f q*)) *A*)| =
|*enn2ereal* (∫ $^+$ *x*. *indicator A* (*f x*) ∂*measure-spmf p*) − *enn2ereal* (∫ $^+$ *x*. *indicator A* (*f x*) ∂*measure-spmf q*)|
unfolding *measure-spmf.emeasure-eq-measure*[*symmetric*]
by(*simp add*: *nn-integral-indicator*[*symmetric*] *indicator-vimage*[*abs-def*] *o-def*)
also have . . . =
|*enn2ereal* (∫ $^+$ *x*. *indicator* (*A* × {*False*}) (*f x*, *bad x*) + *indicator* (*A* × {*True*}) (*f x*, *bad x*) ∂*measure-spmf p*) −
enn2ereal (∫ $^+$ *x*. *indicator* (*A* × {*False*}) (*f x*, *bad x*) + *indicator* (*A* × {*True*}) (*f x*, *bad x*) ∂*measure-spmf q*)|
by(*intro arg-cong*[**where** *f*=*abs*] *arg-cong2*[**where** *f*=(−)] *arg-cong*[**where** *f*=*enn2ereal*] *nn-integral-cong*)(*simp-all split*: *split-indicator*)
also have . . . =
|*enn2ereal* (*emeasure* (*measure-spmf* (*map-spmf* (λ*x*. (*f x*, *bad x*)) *p*)) (*A* × {*False*}) + (∫ $^+$ *x*. *indicator* (*A* × {*True*}) (*f x*, *bad x*) ∂*measure-spmf p*)) −
enn2ereal (*emeasure* (*measure-spmf* (*map-spmf* (λ*x*. (*f x*, *bad x*)) *q*)) (*A* × {*False*}) + (∫ $^+$ *x*. *indicator* (*A* × {*True*}) (*f x*, *bad x*) ∂*measure-spmf q*))|
by(*subst* (*1 2*) *nn-integral-add*)(*simp-all add*: *indicator-vimage*[*abs-def*] *o-def* *nn-integral-indicator*[*symmetric*])
also have . . . = |*enn2ereal* (∫ $^+$ *x*. *indicator* (*A* × {*True*}) (*f x*, *bad x*) ∂*measure-spmf p*) − *enn2ereal* (∫ $^+$ *x*. *indicator* (*A* × {*True*}) (*f x*, *bad x*) ∂*measure-spmf q*)|
(**is** - = |*?x* − *?y*|)
by(*simp add*: *measure-spmf.emeasure-eq-measure not-bad plus-ennreal.rep-eq ereal-diff-add-eq-diff-diff-swap ereal-diff-add-assoc2 ereal-add-uminus-conv-diff*)
also have . . . ≤ *max ?x ?y*
proof(*rule ereal-abs-leI*)
have *?x* − *?y* ≤ *?x* − *0* **by**(*rule ereal-minus-mono*)(*simp-all*)
also have . . . ≤ *max ?x ?y* **by** *simp*
finally show *?x* − *?y* ≤ . . . **.**

have − (*?x* − *?y*) = *?y* − *?x*
by(*rule ereal-minus-diff-eq*)(*simp-all add*: *measure-spmf.nn-integral-indicator-neq-top*)
also have . . . ≤ *?y* − *0* **by**(*rule ereal-minus-mono*)(*simp-all*)
also have . . . ≤ *max ?x ?y* **by** *simp*
finally show − (*?x* − *?y*) ≤ . . . **.**

qed
also have ... ≤ *enn2ereal* (*max* (∫ + *x*. *indicator* {*True*} (*bad x*) ∂*measure-spmf p*) (∫ + *x*. *indicator* {*True*} (*bad x*) ∂*measure-spmf q*))
unfolding *max-enn2ereal less-eq-ennreal.rep-eq*[*symmetric*]
by(*intro max.mono nn-integral-mono*)(*simp-all split*: *split-indicator*)
also have ... = *enn2ereal* (*spmf* (*map-spmf bad p*) *True*)
using *arg-cong2*[**where** *f*=*spmf*, *OF bad-eq refl*, *of True*, *THEN arg-cong*[**where** *f*=*ennreal*]]
unfolding *ennreal-spmf-map-conv-nn-integral indicator-vimage*[*abs-def*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma (**in** *callee-invariant-on*) *exec-gpv-bind-materialize*:
fixes *f* :: ′*s* ⇒ ′*r spmf*
and *g* :: ′*x* × ′*s* ⇒ ′*r* ⇒ ′*y spmf*
and *s* :: ′*s*
defines *exec-gpv2* ≡ *exec-gpv*
assumes *cond*: ⋀*s x y s*′. ⟦ (*y*, *s*′) ∈ *set-spmf* (*callee s x*); *I s* ⟧ ⟹ *f s* = *f s*′
and ℐ: ℐ = ℐ*-full*
shows *bind-spmf* (*exec-gpv callee gpv s*) (λ*as*. *bind-spmf* (*f* (*snd as*)) (*g as*)) =
exec-gpv2 (λ(*r*, *s*) *x*. *bind-spmf* (*callee s x*) (λ(*y*, *s*′). *if I s*′ ∧ *r* = *None then map-spmf* (λ*r*. (*y*, (*Some r*, *s*′))) (*f s*′) *else return-spmf* (*y*, (*r*, *s*′)))) *gpv* (*None*, *s*)
≫= (λ(*a*, *r*, *s*). *case r of None* ⇒ *bind-spmf* (*f s*) (*g* (*a*, *s*)) | *Some r*′ ⇒ *g* (*a*, *s*) *r*′)
(**is** *?lhs* = *?rhs* **is** - = *bind-spmf* (*exec-gpv2 ?callee2* - -) -)
proof −
define *exec-gpv1* :: (′*a*, ′*b*, ′*s option* × ′*s*) *callee* ⇒ (′*x*, ′*a*, ′*b*) *gpv* ⇒ -
where [*simp*]: *exec-gpv1* = *exec-gpv*
let *?X* = λ*s* (*ss*, *s*′). *s* = *s*′
let *?callee* = λ(*ss*, *s*) *x*. *map-spmf* (λ(*y*, *s*′). (*y*, *if I s*′ ∧ *ss* = *None then Some s*′ *else ss*, *s*′)) (*callee s x*)
let *?track* = *exec-gpv1 ?callee gpv* (*None*, *s*)
have *rel-spmf* (*rel-prod* (=) *?X*) (*exec-gpv callee gpv s*) *?track* **unfolding** *exec-gpv1-def*
by(*rule exec-gpv-oracle-bisim*[**where** *X*=*?X*])(*auto simp add*: *spmf-rel-map intro*!: *rel-spmf-reflI*)
hence *exec-gpv callee gpv s* = *map-spmf* (λ(*a*, *ss*, *s*). (*a*, *s*)) *?track*
by(*auto simp add*: *spmf-rel-eq*[*symmetric*] *spmf-rel-map elim*: *rel-spmf-mono*)
hence *?lhs* = *bind-spmf ?track* (λ(*a*, *s*′′, *s*′). *bind-spmf* (*f s*′) (*g* (*a*, *s*′)))
by(*simp add*: *bind-map-spmf o-def split-def*)
also let *?inv* = λ(*ss*, *s*). *case ss of None* ⇒ *True* | *Some s*′ ⇒ *f s* = *f s*′ ∧ *I s*′ ∧ *I s*
interpret *inv*: *callee-invariant-on ?callee ?inv* ℐ
by *unfold-locales*(*auto 4 4 split*: *option.split if-split-asm dest*: *cond callee-invariant simp add*: ℐ)
have *bind-spmf ?track* (λ(*a*, *s*′′, *s*′). *bind-spmf* (*f s*′) (*g* (*a*, *s*′))) =
bind-spmf ?track (λ(*a*, *ss*′, *s*′). *bind-spmf* (*f* (*case ss*′ *of None* ⇒ *s*′ | *Some s*′′ ⇒ *s*′′)) (*g* (*a*, *s*′)))
(**is** - = *?rhs*′)

by(*rule bind-spmf-cong*[*OF refl*])(*auto dest*!: *inv.exec-gpv-invariant split*: *option.split-asm simp add*: $\mathcal{I}$)
also
have *track-Some*: *exec-gpv ?callee gpv* (*Some ss*, *s*) = *map-spmf* (λ(*a*, *s*). (*a*, *Some ss*, *s*)) (*exec-gpv callee gpv s*)
for *s ss* :: ′*s* **and** *gpv* :: (′*x*, ′*a*, ′*b*) *gpv*
proof −
let *?X* = λ(*ss*′, *s*′) *s*. *s* = *s*′ ∧ *ss*′ = *Some ss*
have *rel-spmf* (*rel-prod* (=) *?X*) (*exec-gpv ?callee gpv* (*Some ss*, *s*)) (*exec-gpv callee gpv s*)
by(*rule exec-gpv-oracle-bisim*[**where** *X*=*?X*])(*auto simp add*: *spmf-rel-map intro*!: *rel-spmf-reflI*)
thus *?thesis* **by**(*auto simp add*: *spmf-rel-eq*[*symmetric*] *spmf-rel-map elim*: *rel-spmf-mono*)
qed
have *sample-Some*: *exec-gpv ?callee2 gpv* (*Some r*, *s*) = *map-spmf* (λ(*a*, *s*). (*a*, *Some r*, *s*)) (*exec-gpv callee gpv s*)
for *s* :: ′*s* **and** *r* :: ′*r* **and** *gpv* :: (′*x*, ′*a*, ′*b*) *gpv*
proof −
let *?X* = λ(*r*′, *s*′) *s*. *s*′ = *s* ∧ *r*′ = *Some r*
have *rel-spmf* (*rel-prod* (=) *?X*) (*exec-gpv ?callee2 gpv* (*Some r*, *s*)) (*exec-gpv callee gpv s*)
by(*rule exec-gpv-oracle-bisim*[**where** *X*=*?X*])(*auto simp add*: *spmf-rel-map map-spmf-conv-bind-spmf*[*symmetric*] *split-def intro*!: *rel-spmf-reflI*)
then show *?thesis* **by**(*auto simp add*: *spmf-rel-eq*[*symmetric*] *spmf-rel-map elim*: *rel-spmf-mono*)
qed
have *?rhs*′ = *?rhs*
— Actually, parallel fixpoint induction should be used here, but then we cannot use the facts *track-Some* and *sample-Some* because fixpoint induction replaces *exec-gpv* with approximations. So we do two separate fixpoint inductions instead and jump from the approximation to the fixpoint when the state has been found.
proof(*rule spmf.leq-antisym*)
show *ord-spmf* (=) *?rhs*′ *?rhs* **unfolding** *exec-gpv1-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** *exec-gpv2-def*
apply(*rewrite* **in** *ord-spmf* - - ⌑ *exec-gpv.simps*)
apply(*clarsimp split*: *generat.split simp add*: *bind-map-spmf intro*!: *ord-spmf-bind-reflI split del*: *if-split*)
subgoal for *out rpv ret s*′
apply(*cases I s*′)
subgoal
apply *simp*
apply(*rule spmf.leq-trans*)
apply(*rule ord-spmf-bindI*[*OF step.hyps*])
apply *hypsubst*

```
            apply(rule spmf.leq-refl)
          apply(simp add: track-Some sample-Some bind-map-spmf o-def)
          apply(subst bind-commute-spmf)
          apply(simp add: split-def)
          done
        subgoal
          apply simp
          apply(rule step.IH[THEN spmf.leq-trans])
          apply(simp add: split-def exec-gpv2-def)
          done
        done
      done
  qed
  show ord-spmf (=) ?rhs ?rhs' unfolding exec-gpv2-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 exec-gpv1-def
      apply(rewrite in ord-spmf - - ⌑ exec-gpv.simps)
     apply(clarsimp split: generat.split simp add: bind-map-spmf intro!: ord-spmf-bind-reflI
split del: if-split)
      subgoal for out rpv ret s'
        apply(cases I s')
        subgoal
          apply(simp add: bind-map-spmf o-def)
          apply(rule spmf.leq-trans)
            apply(rule ord-spmf-bind-reflI)
            apply(rule ord-spmf-bindI)
             apply(rule step.hyps)
            apply hypsubst
            apply(rule spmf.leq-refl)
          apply(simp add: track-Some sample-Some bind-map-spmf o-def)
          apply(subst bind-commute-spmf)
          apply(simp add: split-def)
          done
        subgoal
          apply simp
          apply(rule step.IH[THEN spmf.leq-trans])
          apply(simp add: split-def exec-gpv2-def)
          done
        done
      done
  qed
 qed
 finally show ?thesis .
qed

primcorec gpv-stop :: ('a, 'c, 'r) gpv ⇒ ('a option, 'c, 'r option) gpv
```

where
the-gpv (*gpv-stop gpv*) =
map-spmf (*map-generat Some id* (*λrpv input. case input of None* ⇒ *Done None* | *Some input′* ⇒ *gpv-stop* (*rpv input′*)))
(*the-gpv gpv*)

lemma *gpv-stop-Done* [*simp*]: *gpv-stop* (*Done x*) = *Done* (*Some x*)
by(*rule gpv.expand*) *simp*

lemma *gpv-stop-Fail* [*simp*]: *gpv-stop Fail* = *Fail*
by(*rule gpv.expand*) *simp*

lemma *gpv-stop-Pause* [*simp*]: *gpv-stop* (*Pause out rpv*) = *Pause out* (*λinput. case input of None* ⇒ *Done None* | *Some input′* ⇒ *gpv-stop* (*rpv input′*))
by(*rule gpv.expand*) *simp*

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

lemma *gpv-stop-bind* [*simp*]:
gpv-stop (*bind-gpv gpv f*) = *bind-gpv* (*gpv-stop gpv*) (*λx. case x of None* ⇒ *Done None* | *Some x′* ⇒ *gpv-stop* (*f x′*))
apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
apply(*auto 4 3 simp add*: *spmf-rel-map map-spmf-bind-spmf o-def bind-map-spmf bind-gpv.sel generat.rel-map simp del*: *bind-gpv-sel′ intro*!: *rel-spmf-bind-reflI generat.rel-refl-strong rel-spmf-reflI rel-funI split*!: *generat.split option.split*)
done

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

lemma *gpv-stop-parametric′*:
notes [*transfer-rule*] = *the-gpv-parametric′ the-gpv-parametric′ Done-parametric′ corec-gpv-parametric′*
shows (*rel-gpv′′ A C R* ===> *rel-gpv′′* (*rel-option A*) *C* (*rel-option R*)) *gpv-stop gpv-stop*
unfolding *gpv-stop-def* **by** *transfer-prover*

lemma *gpv-stop-parametric* [*transfer-rule*]:
shows (*rel-gpv A C* ===> *rel-gpv* (*rel-option A*) *C*) *gpv-stop gpv-stop*
unfolding *gpv-stop-def* **by** *transfer-prover*

lemma *gpv-stop-transfer*:
(*rel-gpv′′ A B C* ===> *rel-gpv′′* (*pcr-Some A*) *B* (*pcr-Some C*)) (*λx. x*) *gpv-stop*
apply(*rule rel-funI*)
subgoal for *gpv gpv′*
apply(*coinduction arbitrary*: *gpv gpv′*)
apply(*drule rel-gpv′′D*)
apply(*auto simp add*: *spmf-rel-map generat.rel-map rel-fun-def elim*!: *pcr-SomeE*

generat.rel-mono-strong rel-spmf-mono)
  done
done

end

lemma *gpv-stop-map′* [*simp*]:
  gpv-stop (*map-gpv′ f g h gpv*) = *map-gpv′* (*map-option f*) *g* (*map-option h*) (*gpv-stop gpv*)
apply(*coinduction arbitrary*: *gpv rule*: *gpv.coinduct-strong*)
apply(*auto 4 3 simp add*: *spmf-rel-map generat.rel-map intro*!: *rel-spmf-reflI generat.rel-refl-strong split*!: *option.split*)
done

lemma *interaction-bound-gpv-stop* [*simp*]:
  interaction-bound consider (*gpv-stop gpv*) = *interaction-bound consider gpv*
proof(*induction arbitrary*: *gpv rule*: *parallel-fixp-induct-strong-1-1*[*OF complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions interaction-bound.mono interaction-bound.mono interaction-bound-def interaction-bound-def*, *case-names adm bottom step*])
  case *adm* **show** *?case* **by** *simp*
  case *bottom* **show** *?case* **by** *simp*
next
  case (*step interaction-bound′ interaction-bound″*)
  have (*SUP x. interaction-bound′* (*case x of None ⇒ Done None* | *Some input ⇒ gpv-stop* (*c input*))) =
    (*SUP input. interaction-bound″* (*c input*)) (**is** *?lhs* = *?rhs* **is** (*SUP x. ?f x*) = -)
    if *IO out c* ∈ *set-spmf* (*the-gpv gpv*) **for** *out c*
  proof −
    have *?lhs* = *sup* (*interaction-bound′* (*Done None*)) (⨆*x. ?f* (*Some x*))
      by (*simp add*: *UNIV-option-conv image-comp*)
    also have *interaction-bound′* (*Done None*) = *0* **using** *step.hyps*(*1*)[*of Done None*] **by** *simp*
    also have (⨆*x. ?f* (*Some x*)) = *?rhs* **by** (*simp add*: *step.IH*)
    finally show *?thesis* **by** (*simp add*: *bot-enat-def* [*symmetric*])
  qed
  then show *?case*
   by (*auto simp add*: *case-map-generat o-def image-comp cong del*: *generat.case-cong-weak if-weak-cong intro*!: *SUP-cong split*: *generat.split*)
qed

abbreviation *exec-gpv-stop* :: (′*s* ⇒ ′*c* ⇒ (′*r option* × ′*s*) *spmf*) ⇒ (′*a*, ′*c*, ′*r*) *gpv* ⇒ ′*s* ⇒ (′*a option* × ′*s*) *spmf*
where *exec-gpv-stop callee gpv* ≡ *exec-gpv callee* (*gpv-stop gpv*)

abbreviation *inline-stop* :: (′*s* ⇒ ′*c* ⇒ (′*r option* × ′*s*, ′*c*′, ′*r*′) *gpv*) ⇒ (′*a*, ′*c*, ′*r*) *gpv* ⇒ ′*s* ⇒ (′*a option* × ′*s*, ′*c*′, ′*r*′) *gpv*
where *inline-stop callee gpv* ≡ *inline callee* (*gpv-stop gpv*)

context
fixes *joint-oracle* :: *'s1* ⇒ *'s2* ⇒ *'c* ⇒ ((*'r option* × *'s1*) *option* × (*'r option* × *'s2*) *option*) *pmf*
and *callee1* :: *'s1* ⇒ *'c* ⇒ (*'r option* × *'s1*) *spmf*
notes [[*function-internals*]]
begin

partial-function (*spmf*) *exec-until-stop* :: (*'a option*, *'c*, *'r*) *gpv* ⇒ *'s1* ⇒ *'s2* ⇒ *bool* ⇒ (*'a option* × *'s1* × *'s2*) *spmf*
where
exec-until-stop gpv s1 s2 b =
(*if b then*
bind-spmf (*the-gpv gpv*) (λ*generat. case generat of*
Pure x ⇒ *return-spmf* (*x*, *s1*, *s2*)
| *IO out rpv* ⇒ *bind-pmf* (*joint-oracle s1 s2 out*) (λ(*a*, *b*).
case a of None ⇒ *return-pmf None*
| *Some* (*r1*, *s1'*) ⇒ (*case b of None* ⇒ *undefined* | *Some* (*r2*, *s2'*) ⇒
(*case* (*r1*, *r2*) *of* (*None*, *None*) ⇒ *exec-until-stop* (*Done None*) *s1' s2' True*
| (*Some r1'*, *Some r2'*) ⇒ *exec-until-stop* (*rpv r1'*) *s1' s2' True*
| (*None*, *Some r2'*) ⇒ *exec-until-stop* (*Done None*) *s1' s2' True*
| (*Some r1'*, *None*) ⇒ *exec-until-stop* (*rpv r1'*) *s1' s2' False*))))
else
bind-spmf (*the-gpv gpv*) (λ*generat. case generat of*
Pure x ⇒ *return-spmf* (*None*, *s1*, *s2*)
| *IO out rpv* ⇒ *bind-spmf* (*callee1 s1 out*) (λ(*r1*, *s1'*).
case r1 of None ⇒ *exec-until-stop* (*Done None*) *s1' s2 False*
| *Some r1'* ⇒ *exec-until-stop* (*rpv r1'*) *s1' s2 False*)))

end

lemma *ord-spmf-exec-gpv-stop*:
fixes *callee1* :: (*'c*, *'r option*, *'s*) *callee*
and *callee2* :: (*'c*, *'r option*, *'s*) *callee*
and *S* :: *'s* ⇒ *'s* ⇒ *bool*
and *gpv* :: (*'a*, *'c*, *'r*) *gpv*
assumes *bisim*:
⋀*s1 s2 x*. ⟦ *S s1 s2*; ¬ *stop s2* ⟧ ⟹
ord-spmf (λ(*r1*, *s1'*) (*r2*, *s2'*). *le-option r2 r1* ∧ *S s1' s2'* ∧ (*r2* = *None* ∧ *r1* ≠ *None* ⟷ *stop s2'*))
(*callee1 s1 x*) (*callee2 s2 x*)
and *init*: *S s1 s2*
and *go*: ¬ *stop s2*
and *sticking*: ⋀*s1 s2 x y s1'*. ⟦ (*y*, *s1'*) ∈ *set-spmf* (*callee1 s1 x*); *S s1 s2*; *stop s2* ⟧ ⟹ *S s1' s2*
shows *ord-spmf* (*rel-prod* (*ord-option* ⊤)$^{-1-1}$ *S*) (*exec-gpv-stop callee1 gpv s1*) (*exec-gpv-stop callee2 gpv s2*)
proof −
let *?R* = λ(*r1*, *s1'*) (*r2*, *s2'*). *le-option r2 r1* ∧ *S s1' s2'* ∧ (*r2* = *None* ∧ *r1* ≠

None ⟷ *stop s2'*)
obtain *joint* :: *'s* ⇒ *'s* ⇒ *'c* ⇒ ((*'r option* × *'s*) *option* × (*'r option* × *'s*) *option*) *pmf*
where *j1*: *map-pmf fst* (*joint s1 s2 x*) = *callee1 s1 x*
and *j2*: *map-pmf snd* (*joint s1 s2 x*) = *callee2 s2 x*
and *rel* [*rule-format*, *rotated −1*]: ∀ (*a*, *b*) ∈ *set-pmf* (*joint s1 s2 x*). *ord-option ?R a b*
if *S s1 s2* ¬ *stop s2* **for** *x s1 s2* **using** *bisim*
apply *atomize-elim*
apply(*subst* (*asm*) *rel-pmf.simps*)
apply(*unfold rel-spmf-simps all-conj-distrib*[*symmetric*] *all-simps*(*6*) *imp-conjR*[*symmetric*])
apply(*subst all-comm*)
apply(*subst* (*2*) *all-comm*)
apply(*subst choice-iff* [*symmetric*] *ex-simps*(*6*))+
apply *fastforce*
done
note [*simp del*] = *top-apply conversep-iff id-apply*
have ¬ *stop s2* ⟹ *rel-spmf* (*rel-prod* (*ord-option* ⊤)$^{-1-1}$ *S*) (*exec-gpv-stop callee1 gpv s1*) (*map-spmf* (λ(*x*, *s1*, *s2*). (*x*, *s2*)) (*exec-until-stop joint callee1* (*map-gpv Some id gpv*) *s1 s2 True*))
and *rel-spmf* (*rel-prod* (*ord-option* ⊤)$^{-1-1}$ *S*) (*exec-gpv callee1* (*Done None* :: (*'a option*, *'c*, *'r option*) *gpv*) *s1*) (*map-spmf* (λ(*x*, *s1*, *s2*). (*x*, *s2*)) (*exec-until-stop joint callee1* (*Done None* :: (*'a option*, *'c*, *'r*) *gpv*) *s1 s2 b*))
and *stop s2* ⟹ *rel-spmf* (*rel-prod* (*ord-option* ⊤)$^{-1-1}$ *S*) (*exec-gpv-stop callee1 gpv s1*) (*map-spmf* (λ(*x*, *s1*, *y*). (*x*, *y*)) (*exec-until-stop joint callee1* (*map-gpv Some id gpv*) *s1 s2 False*))
for *b* **using** *init*
proof(*induction arbitrary*: *gpv s1 s2 b rule*: *parallel-fixp-induct-2-4* [*OF partial-function-definitions-spmf partial-function-definitions-spmf exec-gpv.mono exec-until-stop.mono exec-gpv-def exec-until-stop-def*, *unfolded lub-spmf-empty*, *case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
{ case *bottom* **case** *1* **show** *?case* **by** *simp* **}**
{ case *bottom* **case** *2* **show** *?case* **by** *simp* **}**
{ case *bottom* **case** *3* **show** *?case* **by** *simp* **}**
next
case (*step exec-gpv' exec-until-stop'*) **case** *step*: *1*
show *?case* **using** *step.prems*
apply(*rewrite gpv-stop.sel*)
apply(*simp add*: *map-spmf-bind-spmf bind-map-spmf gpv.map-sel*)
apply(*rule rel-spmf-bind-reflI*)
apply(*clarsimp split*!: *generat.split*)
apply(*rewrite j1* [*symmetric*], *assumption*+)
apply(*rewrite bind-spmf-def*)
apply(*auto 4 3 split*!: *option.split dest*: *rel intro*: *step.IH intro*!: *rel-pmf-bind-reflI simp add*: *map-bind-pmf bind-map-pmf*)
done
next
case *step* **case** *2*

then show *?case* **by**(*simp add*: *conversep-iff*)
next
case (*step exec-gpv′ exec-until-stop′*) **case** *step*: *3*
show *?case* **using** *step.prems*
apply(*simp add*: *map-spmf-bind-spmf bind-map-spmf gpv.map-sel*)
apply(*rule rel-spmf-bind-reflI*)
apply(*clarsimp simp add*: *map-spmf-bind-spmf split*!: *generat.split*)
apply(*rule rel-spmf-bind-reflI*)
apply *clarsimp*
apply(*drule* (*2*) *sticking*)
apply(*auto split*!: *option.split intro*: *step.IH*)
done
qed
note *this*(*1*)[*OF go*]
also
have ¬ *stop s2* ⟹ *ord-spmf* (=) (*map-spmf* (λ(*x*, *s1*, *s2*). (*x*, *s2*)) (*exec-until-stop joint callee1* (*map-gpv Some id gpv*) *s1 s2 True*)) (*exec-gpv-stop callee2 gpv s2*)
and *ord-spmf* (=) (*map-spmf* (λ(*x*, *s1*, *y*). (*x*, *y*)) (*exec-until-stop joint callee1* (*Done None* :: (′*a option*, ′*c*, ′*r*) *gpv*) *s1 s2 b*)) (*return-spmf* (*None*, *s2*))
and *stop s2* ⟹ *ord-spmf* (=) (*map-spmf* (λ(*x*, *s1*, *s2*). (*x*, *s2*)) (*exec-until-stop joint callee1* (*map-gpv Some id gpv*) *s1 s2 False*)) (*return-spmf* (*None*, *s2*))
for *b* **using** *init*
proof(*induction arbitrary*: *gpv s1 s2 b rule*: *exec-until-stop.fixp-induct*[*case-names adm bottom step*])
case *adm* **show** *?case* **by** *simp*
{ case *bottom* **case** *1* **show** *?case* **by** *simp* **}**
{ case *bottom* **case** *2* **show** *?case* **by** *simp* **}**
{ case *bottom* **case** *3* **show** *?case* **by** *simp* **}**
next
case (*step exec-until-stop′*) **case** *step*: *1*
show *?case* **using** *step.prems*
using [[*show-variants*]]
apply(*rewrite exec-gpv.simps*)
apply(*simp add*: *map-spmf-bind-spmf bind-map-spmf gpv.map-sel*)
apply(*rule ord-spmf-bind-reflI*)
apply(*clarsimp split*!: *generat.split simp add*: *map-bind-pmf bind-spmf-def*)
apply(*rewrite j2*[*symmetric*], *assumption*+)
apply(*auto 4 3 split*!: *option.split dest*: *rel intro*: *step.IH intro*!: *rel-pmf-bind-reflI simp add*: *bind-map-pmf*)
done
next
case *step* **case** *2* **thus** *?case* **by** *simp*
next
case (*step exec-until-stop′*) **case** *3*
thus *?case*
apply(*simp add*: *map-spmf-bind-spmf o-def*)
apply(*rule ord-spmf-bind-spmfI1*)
apply(*clarsimp split*!: *generat.split simp add*: *map-spmf-bind-spmf o-def gpv.map-sel*)

apply(*rule ord-spmf-bind-spmfI1*)
apply *clarsimp*
apply(*drule* (*2*) *sticking*)
apply(*clarsimp split*!: *option.split simp add*: *step.IH*)
done
qed
note *this*(*1*)[*OF go*]
finally show *?thesis* **by**(*rule pmf.rel-mono-strong*)(*auto elim*!: *option.rel-cases*)
qed

end
theory *GPV-Applicative* **imports**
Generative-Probabilistic-Value
SPMF-Applicative
begin

6.7 Applicative instance for (-, *'out*, *'in*) *gpv*

definition *ap-gpv* :: (*'a* ⇒ *'b*, *'out*, *'in*) *gpv* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ (*'b*, *'out*, *'in*) *gpv*
where *ap-gpv f x* = *bind-gpv f* (λ*f'*. *bind-gpv x* (λ*x'*. *Done* (*f' x'*)))

adhoc-overloading *Applicative.ap* ⇌ *ap-gpv*

abbreviation (*input*) *pure-gpv* :: *'a* ⇒ (*'a*, *'out*, *'in*) *gpv*
where *pure-gpv* ≡ *Done*

context includes *applicative-syntax* **begin**

lemma *ap-gpv-id*: *pure-gpv* (λ*x*. *x*) ⋄ *x* = *x*
by(*simp add*: *ap-gpv-def*)

lemma *ap-gpv-comp*: *pure-gpv* (∘) ⋄ *u* ⋄ *v* ⋄ *w* = *u* ⋄ (*v* ⋄ *w*)
by(*simp add*: *ap-gpv-def bind-gpv-assoc*)

lemma *ap-gpv-homo*: *pure-gpv f* ⋄ *pure-gpv x* = *pure-gpv* (*f x*)
by(*simp add*: *ap-gpv-def*)

lemma *ap-gpv-interchange*: *u* ⋄ *pure-gpv x* = *pure-gpv* (λ*f*. *f x*) ⋄ *u*
by(*simp add*: *ap-gpv-def*)

applicative *gpv*
for
pure: *pure-gpv*
ap: *ap-gpv*
by(*rule ap-gpv-id ap-gpv-comp*[*unfolded o-def*[*abs-def*]] *ap-gpv-homo ap-gpv-interchange*)+

lemma *map-conv-ap-gpv*: *map-gpv f* (λ*x*. *x*) *gpv* = *pure-gpv f* ⋄ *gpv*
by(*simp add*: *ap-gpv-def map-gpv-conv-bind*)

lemma *exec-gpv-ap*:
exec-gpv callee (*f* ⋄ *x*) *σ* =
exec-gpv callee f σ ⤜ (λ(*f*′, *σ*′). *pure-spmf* (λ(*x*′, *σ*′′). (*f*′ *x*′, *σ*′′)) ⋄ *exec-gpv callee x σ*′)
by(*simp add*: *ap-gpv-def exec-gpv-bind ap-spmf-conv-bind split-def*)

lemma *exec-gpv-ap-pure* [*simp*]:
exec-gpv callee (*pure-gpv f* ⋄ *x*) *σ* = *pure-spmf* (*apfst f*) ⋄ *exec-gpv callee x σ*
by(*simp add*: *exec-gpv-ap apfst-def map-prod-def*)

end

end

7 Cyclic groups

theory *Cyclic-Group* **imports**
HOL−Algebra.Coset
begin

record ′*a cyclic-group* = ′*a monoid* +
generator :: ′*a* (‹**g**ı›)

locale *cyclic-group* = *group G*
for *G* :: (′*a*, ′*b*) *cyclic-group-scheme* (**structure**)
+
assumes *generator-closed* [*intro*, *simp*]: *generator G* ∈ *carrier G*
and *generator*: *carrier G* ⊆ *range* (λ*n* :: *nat*. *generator G* [^]$_G$ *n*)
begin

lemma *generatorE* [*elim?*]:
assumes *x* ∈ *carrier G*
obtains *n* :: *nat* **where** *x* = *generator G* [^] *n*
using *generator assms* **by** *auto*

lemma *inj-on-generator*: *inj-on* (([^]) **g**) {..<*order G*}
proof(*rule inj-onI*)
fix *n m*
assume *n* ∈ {..<*order G*} *m* ∈ {..<*order G*}
hence *n*: *n* < *order G* **and** *m*: *m* < *order G* **by** *simp-all*
moreover
assume **g** [^] *n* = **g** [^] *m*
ultimately show *n* = *m*
proof(*induction n m rule*: *linorder-wlog*)
case *sym* **thus** *?case* **by** *simp*
next
case (*le n m*)
let *?d* = *m* − *n*

```
    have g [^] (int m − int n) = g [^] int m ⊗ inv (g [^] int n)
      by(simp add: int-pow-diff)
    also have g [^] int m = g [^] int n by(simp add: le.prems int-pow-int)
    also have . . . ⊗ inv (g [^] (int n)) = 1 by simp
    finally have g [^] ?d = 1
      using le.prems(3) pow-eq-div2 by force
    { assume n < m
      have carrier G ⊆ (λn. g [^] n) ‘ {..<?d}
      proof
        fix x
        assume x ∈ carrier G
        then obtain k :: nat where x = g [^] k ..
        also have . . . = (g [^] ?d) [^] (k div ?d) ⊗ g [^] (k mod ?d)
          by(simp add: nat-pow-pow nat-pow-mult div-mult-mod-eq)
        also have . . . = g [^] (k mod ?d)
          using ‹g [^] ?d = 1› by simp
        finally show x ∈ (λn. g [^] n) ‘ {..<?d} using ‹n < m› by auto
      qed
      hence order G ≤ card ((λn. g [^] n) ‘ {..<?d})
        by(simp add: order-def card-mono)
      also have . . . ≤ card {..<?d} by(rule card-image-le) simp
      also have . . . < order G using ‹m < order G› by simp
      finally have False by simp }
    with ‹n ≤ m› show n = m by(auto simp add: order.order-iff-strict)
  qed
qed

lemma finite-carrier: finite (carrier G)
proof −
  from generator obtain n :: nat where g [^] n = inv g
    by(metis generatorE generator-closed inv-closed)
  then have g1: g [^] (Suc n) = 1
    by auto
  have mod: g [^] m = g [^] (m mod Suc n) for m
  proof −
    obtain k where m mod Suc n + Suc n ∗ k = m
      using mod-mult-div-eq by blast
    then have g [^] m = g [^] (m mod Suc n + Suc n ∗ k) by simp
    also have . . . = g [^] (m mod Suc n)
     unfolding nat-pow-mult[symmetric, OF generator-closed] nat-pow-pow[symmetric,
OF generator-closed] g1
      by simp
    finally show ?thesis .
  qed
  have g [^] x ∈ ([^]) g ‘ {..<Suc n} for x :: nat by (subst mod) auto
  then have range (([^]) g :: nat ⇒ -) ⊆ (([^]) g) ‘ {..<Suc n} by auto
  then have finite (range (([^]) g :: nat ⇒ -)) by(rule finite-surj[rotated]) simp
  with generator show ?thesis by(rule finite-subset)
qed
```

lemma *carrier-conv-generator*: *carrier G* = ($\lambda n.$ **g** [^] *n*) ' {..<*order G*}
proof −
  have ($\lambda n.$ **g** [^] *n*) ' {..<*order G*} ⊆ *carrier G* **by** *auto*
  moreover have *card* (($\lambda n.$ **g** [^] *n*) ' {..<*order G*}) ≥ *order G*
    using *inj-on-generator* **by**(*simp add*: *card-image*)
  ultimately show *?thesis* **using** *finite-carrier*
    unfolding *order-def* **by**(*rule card-seteq*[*symmetric*, *rotated*])
qed

lemma *bij-betw-generator-carrier*:
  bij-betw (λn :: *nat*. **g** [^] *n*) {..<*order G*} (*carrier G*)
  by (*simp add*: *carrier-conv-generator inj-on-generator inj-on-imp-bij-betw*)

lemma *order-gt-0*: *order G* > *0*
  using *order-gt-0-iff-finite* **by**(*simp add*: *finite-carrier*)

end

lemma (**in** *monoid*) *order-in-range-Suc*: *order G* ∈ *range Suc* ⟷ *finite* (*carrier G*)
  by(*cases order G*)(*auto simp add*: *order-def carrier-not-empty intro*: *card-ge-0-finite*)

end

theory *Cyclic-Group-SPMF* **imports**
  Cyclic-Group
  HOL−Probability.SPMF
begin

definition *sample-uniform* :: *nat* ⇒ *nat spmf*
where *sample-uniform n* = *spmf-of-set* {..<*n*}

lemma *spmf-sample-uniform*: *spmf* (*sample-uniform n*) *x* = *indicator* {..<*n*} *x* / *n*
by(*simp add*: *sample-uniform-def spmf-of-set*)

lemma *weight-sample-uniform*: *weight-spmf* (*sample-uniform n*) = *indicator* (*range Suc*) *n*
by(*auto simp add*: *sample-uniform-def weight-spmf-of-set split*: *split-indicator elim*: *lessE*)

lemma *weight-sample-uniform-0* [*simp*]: *weight-spmf* (*sample-uniform 0*) = *0*
by(*auto simp add*: *weight-sample-uniform indicator-def*)

lemma *weight-sample-uniform-gt-0* [*simp*]: *0* < *n* ⟹ *weight-spmf* (*sample-uniform n*) = *1*
by(*auto simp add*: *weight-sample-uniform indicator-def gr0-conv-Suc*)

```
lemma lossless-sample-uniform [simp]: lossless-spmf (sample-uniform n) ⟷ 0 <
n
by(auto simp add: lossless-spmf-def intro: ccontr)

lemma set-spmf-sample-uniform [simp]: 0 < n ⟹ set-spmf (sample-uniform n)
= {..<n}
by(simp add: sample-uniform-def)

lemma (in cyclic-group) sample-uniform-one-time-pad:
  assumes [simp]: c ∈ carrier G
  shows
  map-spmf (λx. g [^] x ⊗ c) (sample-uniform (order G)) =
   map-spmf (λx. g [^] x) (sample-uniform (order G))
   (is ?lhs = ?rhs)
proof(cases finite (carrier G))
  case False
  thus ?thesis by(simp add: order-def sample-uniform-def)
next
  case True
  have ?lhs = map-spmf (λx. x ⊗ c) ?rhs
    by(simp add: pmf.map-comp o-def option.map-comp)
  also have rhs: ?rhs = spmf-of-set (carrier G)
   using True by(simp add: carrier-conv-generator inj-on-generator sample-uniform-def)
  also have map-spmf (λx. x ⊗ c) ... = spmf-of-set ((λx. x ⊗ c) ` carrier G)
    by(simp add: inj-on-multc)
  also have (λx. x ⊗ c) ` carrier G = carrier G
    using True by(rule endo-inj-surj)(auto simp add: inj-on-multc)
  finally show ?thesis using rhs by simp
qed

end
theory CryptHOL imports
  GPV-Bisim
  GPV-Applicative
  Computational-Model
  Negligible
  Cyclic-Group-SPMF
  List-Bits
  Environment-Functor
begin

end
```

References

[1] A. Lochbihler. Probabilistic functions and cryptographic oracles in higher order logic. In P. Thiemann, editor, *Programming Languages and*

Systems (ESOP 2016), volume 9632 of *LNCS*, pages 503–531. Springer, 2016.