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*: $(PROP\ P \Longrightarrow PROP\ P) \equiv Trueprop\ True$
⟨*proof*⟩

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

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

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

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

lemma *imp-OO-imp* [*simp*]: $(\longrightarrow)\ OO\ (\longrightarrow) = (\longrightarrow)$
⟨*proof*⟩

lemma *inj-on-fun-updD*: $[\![\ inj\text{-}on\ (f(x := y))\ A;\ x \notin A\ ]\!] \Longrightarrow inj\text{-}on\ f\ A$
⟨*proof*⟩

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

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 $Least\ P \le Least\ Q$
⟨*proof*⟩

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

⟨*proof*⟩

1.2 Relations

inductive *Imagep* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b \Rightarrow bool$
for *R P*
where *ImagepI*: $[\![P\ x;\ R\ x\ y]\!] \Longrightarrow Imagep\ R\ P\ y$

lemma *r-r-into-tranclp*: $[\![r\ x\ y;\ r\ y\ z]\!] \Longrightarrow r\hat{}{++}\ x\ z$
⟨*proof*⟩

lemma *transp-tranclp-id*:
assumes *transp R*
shows *tranclp* $R = R$
⟨*proof*⟩

lemma *transp-inv-image*: *transp r* $\Longrightarrow$ *transp* $(\lambda x\ y.\ r\ (f\ x)\ (f\ y))$
⟨*proof*⟩

lemma *Domainp-conversep*: *Domainp* R^{-1-1} = *Rangep R*
⟨*proof*⟩

lemma *bi-unique-rel-set-bij-betw*:
assumes *unique*: *bi-unique R*
and *rel*: *rel-set R A B*
shows $\exists f.$ *bij-betw f A B* $\wedge$ $(\forall x{\in}A.\ R\ x\ (f\ x))$
⟨*proof*⟩

definition *restrict-relp* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
(‹- ↿ (- ⊗ -)› [*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*]: $(R \upharpoonleft P \otimes Q)\ x\ y \longleftrightarrow R\ x\ y \wedge P\ x \wedge Q\ y$
⟨*proof*⟩

lemma *restrict-relpI* [*intro?*]: $[\![R\ x\ y;\ P\ x;\ Q\ y]\!] \Longrightarrow (R \upharpoonleft P \otimes Q)\ x\ y$
⟨*proof*⟩

lemma *restrict-relpE* [*elim?*, *cases pred*]:
assumes $(R \upharpoonleft P \otimes Q)\ x\ y$
obtains (*restrict-relp*) *R x y P x Q y*
⟨*proof*⟩

lemma *conversep-restrict-relp* [*simp*]: $(R \upharpoonleft P \otimes Q)^{-1-1} = R^{-1-1} \upharpoonleft Q \otimes P$
⟨*proof*⟩

lemma *restrict-relp-restrict-relp* [*simp*]: $R \upharpoonleft P \otimes Q \upharpoonleft P' \otimes Q' = R \upharpoonleft$ *inf* $P\ P' \otimes$ *inf* $Q\ Q'$

⟨proof⟩

lemma *restrict-relp-cong*:
$[\![P = P'; Q = Q'; \bigwedge x\ y.\ [\![P\ x;\ Q\ y]\!] \Longrightarrow R\ x\ y = R'\ x\ y]\!] \Longrightarrow R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$
⟨proof⟩

lemma *restrict-relp-cong-simp*:
$[\![P = P'; Q = Q'; \bigwedge x\ y.\ P\ x =simp=> Q\ y =simp=> R\ x\ y = R'\ x\ y]\!] \Longrightarrow R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$
⟨proof⟩

lemma *restrict-relp-parametric* [*transfer-rule*]:
includes *lifting-syntax* **shows**
((*A* ===> *B* ===> (=)) ===> (*A* ===> (=)) ===> (*B* ===> (=)) ===> *A* ===> *B* ===> (=)) *restrict-relp restrict-relp*
⟨proof⟩

lemma *restrict-relp-mono*: $[\![R \leq R'; P \leq P'; Q \leq Q']\!] \Longrightarrow R \upharpoonright P \otimes Q \leq R' \upharpoonright P' \otimes Q'$
⟨proof⟩

lemma *restrict-relp-mono'*:
$[\![(R \upharpoonright P \otimes Q)\ x\ y; [\![R\ x\ y;\ P\ x;\ Q\ y]\!] \Longrightarrow R'\ x\ y\ \&\&\&\ P'\ x\ \&\&\&\ Q'\ y]\!]$
$\Longrightarrow (R' \upharpoonright P' \otimes Q')\ x\ y$
⟨proof⟩

lemma *restrict-relp-DomainpD*: *Domainp* $(R \upharpoonright P \otimes Q)\ x \Longrightarrow$ *Domainp* $R\ x \wedge P\ x$
⟨proof⟩

lemma *restrict-relp-True*: $R \upharpoonright (\lambda\text{-}.\ True) \otimes (\lambda\text{-}.\ True) = R$
⟨proof⟩

lemma *restrict-relp-False1*: $R \upharpoonright (\lambda\text{-}.\ False) \otimes Q = bot$
⟨proof⟩

lemma *restrict-relp-False2*: $R \upharpoonright P \otimes (\lambda\text{-}.\ False) = bot$
⟨proof⟩

definition *rel-prod2* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow ('c \times 'b) \Rightarrow bool$
where *rel-prod2* $R\ a = (\lambda(c, b).\ R\ a\ b)$

lemma *rel-prod2-simps* [*simp*]: *rel-prod2* $R\ a\ (c, b) \longleftrightarrow R\ a\ b$
⟨proof⟩

lemma *restrict-rel-prod*:
rel-prod $(R \upharpoonright I1 \otimes I2)\ (S \upharpoonright I1' \otimes I2')$ = *rel-prod* $R\ S \upharpoonright$ *pred-prod* $I1\ I1' \otimes$ *pred-prod* $I2\ I2'$

⟨*proof*⟩

lemma *restrict-rel-prod1*:
rel-prod $(R \upharpoonright I1 \otimes I2)$ S = *rel-prod* R S $\upharpoonright$ *pred-prod* $I1$ $(\lambda$-. *True*$)$ $\otimes$ *pred-prod* $I2$ $(\lambda$-. *True*$)$
⟨*proof*⟩

lemma *restrict-rel-prod2*:
rel-prod R $(S \upharpoonright I1 \otimes I2)$ = *rel-prod* R S $\upharpoonright$ *pred-prod* $(\lambda$-. *True*$)$ $I1$ $\otimes$ *pred-prod* $(\lambda$-. *True*$)$ $I2$
⟨*proof*⟩

consts *relcompp-witness* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'a \times 'c \Rightarrow 'b$
specification (*relcompp-witness*)
relcompp-witness1: $(A\ OO\ B)$ (*fst xy*) (*snd xy*) $\Longrightarrow$ A (*fst xy*) (*relcompp-witness* A B *xy*)
relcompp-witness2: $(A\ OO\ B)$ (*fst xy*) (*snd xy*) $\Longrightarrow$ B (*relcompp-witness* A B *xy*) (*snd xy*)
⟨*proof*⟩

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

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

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

1.3 Pairs

lemma *split-apfst* [*simp*]: *case-prod* h (*apfst* f *xy*) = *case-prod* $(h \circ f)$ *xy*
⟨*proof*⟩

definition *corec-prod* :: $('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow 'b) \Rightarrow 's \Rightarrow 'a \times 'b$
where *corec-prod* f g = $(\lambda s.\ (f\ s,\ g\ s))$

lemma *corec-prod-apply*: *corec-prod* f g s = $(f\ s,\ g\ s)$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *apfst-corec-prod* [*simp*]: *apfst* h (*corec-prod* f g s) = *corec-prod* $(h \circ f)$ g s
⟨*proof*⟩

lemma *apsnd-corec-prod* [*simp*]: *apsnd* h (*corec-prod* f g s) = *corec-prod* f $(h \circ g)$ s

⟨*proof*⟩

lemma *map-corec-prod* [*simp*]: *map-prod f g* (*corec-prod h k s*) = *corec-prod* (*f* ∘ *h*) (*g* ∘ *k*) *s*
⟨*proof*⟩

lemma *split-corec-prod* [*simp*]: *case-prod h* (*corec-prod f g s*) = *h* (*f s*) (*g s*)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

1.4 Sums

lemma *islE*:

assumes *isl x*
obtains *l* **where** *x = Inl l*
⟨*proof*⟩

lemma *Inl-in-Plus* [*simp*]: *Inl x* ∈ *A* <+> *B* ⟷ *x* ∈ *A*
⟨*proof*⟩

lemma *Inr-in-Plus* [*simp*]: *Inr x* ∈ *A* <+> *B* ⟷ *x* ∈ *B*
⟨*proof*⟩

lemma *Inl-eq-map-sum-iff*: *Inl x = map-sum f g y* ⟷ (∃ *z*. *y = Inl z* ∧ *x = f z*)
⟨*proof*⟩

lemma *Inr-eq-map-sum-iff*: *Inr x = map-sum f g y* ⟷ (∃ *z*. *y = Inr z* ∧ *x = g z*)
⟨*proof*⟩

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

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

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

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

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

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

1.5 Option

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

lemma *case-option-collapse*: *case-option x* (λ-. *x*) *y* = *x*
⟨*proof*⟩

lemma *indicator-single-Some*: $indicator\ \{Some\ x\}\ (Some\ y) = indicator\ \{x\}\ y$
⟨*proof*⟩

1.5.1 Predicator and relator

lemma *option-pred-mono-strong*:
$[\![$ *pred-option* $P\ x$; $\bigwedge a.\ [\![\ a \in$ *set-option* $x;\ P\ a\ ]\!] \Longrightarrow P'\ a\ ]\!] \Longrightarrow$ *pred-option* $P'\ x$
⟨*proof*⟩

lemma *option-pred-map* [*simp*]: *pred-option* P (*map-option* $f\ x$) = *pred-option* ($P \circ f$) x
⟨*proof*⟩

lemma *option-pred-o-map* [*simp*]: *pred-option* $P \circ$ *map-option* f = *pred-option* ($P \circ f$)
⟨*proof*⟩

lemma *option-pred-bind* [*simp*]: *pred-option* P (*Option.bind* $x\ f$) = *pred-option* (*pred-option* $P \circ f$) x
⟨*proof*⟩

lemma *pred-option-conj* [*simp*]:
pred-option ($\lambda x.\ P\ x \wedge Q\ x$) = ($\lambda x.$ *pred-option* $P\ x \wedge$ *pred-option* $Q\ x$)
⟨*proof*⟩

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

lemma *rel-option-restrict-relpI* [*intro?*]:
$[\![$ *rel-option* $R\ x\ y$; *pred-option* $P\ x$; *pred-option* $Q\ y\ ]\!] \Longrightarrow$ *rel-option* ($R \upharpoonright P \otimes Q$) $x\ y$
⟨*proof*⟩

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*⟩

lemma *rel-option-restrict-relp-iff*:
rel-option ($R \upharpoonright P \otimes Q$) $x\ y \longleftrightarrow$ *rel-option* $R\ x\ y \wedge$ *pred-option* $P\ x \wedge$ *pred-option* $Q\ y$
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

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

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

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*)
⟨*proof*⟩

lemma *le-option-refl* [*simp*]: *le-option x x*
⟨*proof*⟩

lemma *le-option-conv-option-ord*: *le-option* = *option-ord*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pcr-SomeE* [*cases pred*]:
  assumes *pcr-Some R x y*
  obtains (*pcr-Some*) *z* **where** $y = Some\ z\ R\ x\ z$
⟨*proof*⟩

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*}
⟨*proof*⟩

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

lemma *is-none-filter-option* [*simp*]: *Option.is-none* (*filter-option P x*) ⟷ *Option.is-none x* ∨ ¬ *P* (*the x*)
⟨*proof*⟩

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

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

lemma *filter-conv-bind-option*: *filter-option P x* = *Option.bind x* (λ*y*. *if P y then Some y else None*)
⟨*proof*⟩

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* {})
⟨*proof*⟩

lemma *in-set-assert-option* [*simp*]: *x* ∈ *set-option* (*assert-option b*) ⟷ *b*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *in-set-join-option*: *x* ∈ *set-option* (*join-option* (*Some* (*Some x*)))
⟨*proof*⟩

lemma *map-join-option*: *map-option f* (*join-option x*) = *join-option* (*map-option* (*map-option f*) *x*)
⟨*proof*⟩

lemma *bind-conv-join-option*: *Option.bind x f* = *join-option* (*map-option f x*)
⟨*proof*⟩

lemma *join-conv-bind-option*: *join-option x* = *Option.bind x id*
⟨*proof*⟩

lemma *join-option-parametric* [*transfer-rule*]:
includes *lifting-syntax* **shows**
(*rel-option* (*rel-option R*) ===> *rel-option R*) *join-option join-option*
⟨*proof*⟩

lemma *join-option-eq-Some* [*simp*]: *join-option x* = *Some y* ⟷ *x* = *Some* (*Some y*)
⟨*proof*⟩

lemma *Some-eq-join-option* [*simp*]: *Some y* = *join-option x* ⟷ *x* = *Some* (*Some y*)
⟨*proof*⟩

lemma *join-option-eq-None*: *join-option x* = *None* ⟷ *x* = *None* ∨ *x* = *Some None*
⟨*proof*⟩

lemma *None-eq-join-option*: *None* = *join-option x* ⟷ *x* = *None* ∨ *x* = *Some None*
⟨*proof*⟩

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*
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *zip-option-eq-Some-iff* [*iff*]:
  zip-option x y = *Some* (*a*, *b*) ⟷ *x* = *Some a* ∧ *y* = *Some b*
⟨*proof*⟩

lemma *set-zip-option* [*simp*]:
  set-option (*zip-option x y*) = *set-option x* × *set-option y*
⟨*proof*⟩

lemma *zip-map-option1*: *zip-option* (*map-option f x*) *y* = *map-option* (*apfst f*) (*zip-option x y*)
⟨*proof*⟩

lemma *zip-map-option2*: *zip-option x* (*map-option g y*) = *map-option* (*apsnd g*) (*zip-option x y*)
⟨*proof*⟩

lemma *map-zip-option*:
  map-option (*map-prod f g*) (*zip-option x y*) = *zip-option* (*map-option f x*) (*map-option g y*)
⟨*proof*⟩

lemma *zip-conv-bind-option*:
  zip-option x y = *Option.bind x* (λ*x*. *Option.bind y* (λ*y*. *Some* (*x*, *y*)))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-option-eqI* [*simp*]: *rel-option* (=) *x x*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sup-option-assoc*: *sup-option* (*sup-option x y*) *z* = *sup-option x* (*sup-option y z*)
⟨*proof*⟩

lemma *sup-option-left-idem*: *sup-option x* (*sup-option x y*) = *sup-option x y*
⟨*proof*⟩

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

lemma *sup-option-None* [*simp*]: *sup-option None y* = *y*
⟨*proof*⟩

1.5.8 Restriction on *'a option*

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5.9 Maps

lemma *map-add-apply*: (*m1* ++ *m2*) *x* = *sup-option* (*m1 x*) (*m2 x*)
⟨*proof*⟩

lemma *map-le-map-upd2*: ⟦ $f \subseteq_m g$; ⋀*y'*. *f x* = *Some y'* ⟹ *y'* = *y* ⟧ ⟹ $f \subseteq_m g(x \mapsto y)$
⟨*proof*⟩

lemma *eq-None-iff-not-dom*: *f x* = *None* ⟷ *x* ∉ *dom f*
⟨*proof*⟩

lemma *card-ran-le-dom*: *finite* (*dom m*) ⟹ *card* (*ran m*) ≤ *card* (*dom m*)
⟨*proof*⟩

lemma *dom-subset-ran-iff*:
assumes *finite* (*ran m*)
shows *dom m* ⊆ *ran m* ⟷ *dom m* = *ran m*
⟨*proof*⟩

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-empty* ≡ *Map.empty*

lemma *map-le-Some1D*: ⟦ $m \subseteq_m m'$; *m x* = *Some y* ⟧ ⟹ *m' x* = *Some y*
⟨*proof*⟩

lemma *map-le-fun-upd2*: ⟦ $f \subseteq_m g$; *x* ∉ *dom f* ⟧ ⟹ $f \subseteq_m g(x := y)$
⟨*proof*⟩

lemma *map-eqI*: ∀ *x*∈*dom m* ∪ *dom m'*. *m x* = *m' x* ⟹ *m* = *m'*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

1.7 Extended naturals

lemma *idiff-enat-eq-enat-iff*: $x -$ *enat* $n =$ *enat* $m \longleftrightarrow (\exists k.\ x =$ *enat* $k \wedge k - n = m)$
⟨*proof*⟩

lemma *eSuc-SUP*: $A \neq \{\} \Longrightarrow$ *eSuc* $(\bigsqcup\ (f\ `\ A)) = (\bigsqcup x \in A.$ *eSuc* $(f\ x))$
⟨*proof*⟩

lemma *ereal-of-enat-1*: *ereal-of-enat 1* = *ereal 1*
⟨*proof*⟩

lemma *ennreal-real-conv-ennreal-of-enat*: *ennreal* (*real n*) = *ennreal-of-enat n*
⟨*proof*⟩

lemma *enat-add-sub-same2*: $b \neq \infty \Longrightarrow a + b - b = (a :: enat)$
⟨*proof*⟩

lemma *enat-sub-add*: $y \leq x \Longrightarrow x - y + z = x + z - (y :: enat)$
⟨*proof*⟩

lemma *SUP-enat-eq-0-iff* [*simp*]: $\bigsqcup\ (f\ `\ A) = (0 :: enat) \longleftrightarrow (\forall x \in A.\ f\ x = 0)$
⟨*proof*⟩

lemma *SUP-enat-add-left*:
assumes $I \neq \{\}$
shows $(SUP\ i \in I.\ f\ i + c :: enat) = (SUP\ i \in I.\ f\ i) + c$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *SUP-enat-add-right*:
assumes $I \neq \{\}$
shows $(SUP\ i \in I.\ c + f\ i :: enat) = c + (SUP\ i \in I.\ f\ i)$
⟨*proof*⟩

lemma *iadd-SUP-le-iff*: $n + (SUP\ x \in A.\ f\ x :: enat) \leq y \longleftrightarrow$ (*if* $A = \{\}$ *then* $n \leq y$ *else* $\forall x \in A.\ n + f\ x \leq y$)
⟨*proof*⟩

lemma *SUP-iadd-le-iff*: $(SUP\ x{\in}A.\ f\ x :: enat) + n \leq y \longleftrightarrow (if\ A = \{\}\ then\ n \leq y\ else\ \forall x{\in}A.\ f\ x + n \leq y)$
⟨*proof*⟩

1.8 Extended non-negative reals

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

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

lemma *nn-integral-indicator-map*:
assumes [*measurable*]: $f \in measurable\ M\ N\ \{x{\in}space\ N.\ P\ x\} \in sets\ N$
shows $(\int^+ x.\ indicator\ \{x{\in}space\ N.\ P\ x\}\ (f\ x)\ \partial M) = emeasure\ M\ \{x{\in}space\ M.\ P\ (f\ x)\}$
⟨*proof*⟩

1.9 BNF material

lemma *transp-rel-fun*: $[\![\ is\text{-}equality\ Q;\ transp\ R\ ]\!] \Longrightarrow transp\ (rel\text{-}fun\ Q\ R)$
⟨*proof*⟩

lemma *rel-fun-inf*: $inf\ (rel\text{-}fun\ Q\ R)\ (rel\text{-}fun\ Q\ R') = rel\text{-}fun\ Q\ (inf\ R\ R')$
⟨*proof*⟩

lemma *reflp-fun1*: **includes** *lifting-syntax* **shows** $[\![\ is\text{-}equality\ A;\ reflp\ B\ ]\!] \Longrightarrow reflp\ (A ===> B)$
⟨*proof*⟩

lemma *type-copy-id'*: $type\text{-}definition\ (\lambda x.\ x)\ (\lambda x.\ x)\ UNIV$
⟨*proof*⟩

lemma *type-copy-id*: *type-definition id id UNIV*
⟨*proof*⟩

lemma *GrpE* [*cases pred*]:
assumes *BNF-Def.Grp A f x y*
obtains $(Grp)\ y = f\ x\ x \in A$
⟨*proof*⟩

lemma *rel-fun-Grp-copy-Abs*:
includes *lifting-syntax*
assumes *type-definition Rep Abs A*
shows $rel\text{-}fun\ (BNF\text{-}Def.Grp\ A\ Abs)\ (BNF\text{-}Def.Grp\ B\ g) = BNF\text{-}Def.Grp\ \{f.\ f\ `\ A \subseteq B\}\ (Rep\ ---> g)$
⟨*proof*⟩

lemma *rel-set-Grp*:
rel-set (*BNF-Def.Grp* *A* *f*) = *BNF-Def.Grp* {*B*. *B* ⊆ *A*} (*image f*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *pred-prod-conj* [*simp*]:
shows *pred-prod-conj1*: ⋀*P Q R*. *pred-prod* (λ*x*. *P x* ∧ *Q x*) *R* = (λ*x*. *pred-prod P R x* ∧ *pred-prod Q R x*)
and *pred-prod-conj2*: ⋀*P Q R*. *pred-prod P* (λ*x*. *Q x* ∧ *R x*) = (λ*x*. *pred-prod P Q x* ∧ *pred-prod P R x*)
⟨*proof*⟩

lemma *pred-sum-conj* [*simp*]:
shows *pred-sum-conj1*: ⋀*P Q R*. *pred-sum* (λ*x*. *P x* ∧ *Q x*) *R* = (λ*x*. *pred-sum P R x* ∧ *pred-sum Q R x*)
and *pred-sum-conj2*: ⋀*P Q R*. *pred-sum P* (λ*x*. *Q x* ∧ *R x*) = (λ*x*. *pred-sum P Q x* ∧ *pred-sum P R x*)
⟨*proof*⟩

lemma *pred-list-conj* [*simp*]: *list-all* (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *list-all P x* ∧ *list-all Q x*)
⟨*proof*⟩

lemma *pred-prod-top* [*simp*]:
pred-prod (λ-. *True*) (λ-. *True*) = (λ-. *True*)
⟨*proof*⟩

lemma *rel-fun-conversep*: **includes** *lifting-syntax* **shows**
(*A*^−−*1* ===> *B*^−−*1*) = (*A* ===> *B*)^−−*1*
⟨*proof*⟩

lemma *left-unique-Grp* [*iff*]:
left-unique (*BNF-Def.Grp A f*) ⟷ *inj-on f A*
⟨*proof*⟩

lemma *right-unique-Grp* [*simp*, *intro*!]: *right-unique* (*BNF-Def.Grp A f*)
⟨*proof*⟩

lemma *bi-unique-Grp* [*iff*]:
bi-unique (*BNF-Def.Grp A f*) ⟷ *inj-on f A*
⟨*proof*⟩

lemma *left-total-Grp* [*iff*]:
left-total (*BNF-Def.Grp A f*) $\longleftrightarrow$ *A* = *UNIV*
⟨*proof*⟩

lemma *right-total-Grp* [*iff*]:
right-total (*BNF-Def.Grp A f*) $\longleftrightarrow$ *f* ' *A* = *UNIV*
⟨*proof*⟩

lemma *bi-total-Grp* [*iff*]:
bi-total (*BNF-Def.Grp A f*) $\longleftrightarrow$ *A* = *UNIV* ∧ *surj f*
⟨*proof*⟩

lemma *left-unique-vimage2p* [*simp*]:
⟦ *left-unique P*; *inj f* ⟧ $\Longrightarrow$ *left-unique* (*BNF-Def.vimage2p f g P*)
⟨*proof*⟩

lemma *right-unique-vimage2p* [*simp*]:
⟦ *right-unique P*; *inj g* ⟧ $\Longrightarrow$ *right-unique* (*BNF-Def.vimage2p f g P*)
⟨*proof*⟩

lemma *bi-unique-vimage2p* [*simp*]:
⟦ *bi-unique P*; *inj f*; *inj g* ⟧ $\Longrightarrow$ *bi-unique* (*BNF-Def.vimage2p f g P*)
⟨*proof*⟩

lemma *left-total-vimage2p* [*simp*]:
⟦ *left-total P*; *surj g* ⟧ $\Longrightarrow$ *left-total* (*BNF-Def.vimage2p f g P*)
⟨*proof*⟩

lemma *right-total-vimage2p* [*simp*]:
⟦ *right-total P*; *surj f* ⟧ $\Longrightarrow$ *right-total* (*BNF-Def.vimage2p f g P*)
⟨*proof*⟩

lemma *bi-total-vimage2p* [*simp*]:
⟦ *bi-total P*; *surj f*; *surj g* ⟧ $\Longrightarrow$ *bi-total* (*BNF-Def.vimage2p f g P*)
⟨*proof*⟩

lemma *vimage2p-eq* [*simp*]:
inj f $\Longrightarrow$ *BNF-Def.vimage2p f f* (=) = (=)
⟨*proof*⟩

lemma *vimage2p-conversep*: *BNF-Def.vimage2p f g R*^−−*1* = (*BNF-Def.vimage2p g f R*)^−−*1*
⟨*proof*⟩

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

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

A' x}; B x y ⟧ ⟹ B' x y ⟧ ⟹ *rel-fun* A' B' f g
⟨*proof*⟩

lemma *rel-fun-refl-strong*:
assumes $A \leq (=)$ $\bigwedge x.\ x \in f$ ‘ $\{x.$ *Domainp* $A\ x\} \Longrightarrow B\ x\ x$
shows *rel-fun* A B f f
⟨*proof*⟩

lemma *Grp-iff*: *BNF-Def.Grp* B g x $y \longleftrightarrow y = g\ x \wedge x \in B$ ⟨*proof*⟩

lemma *Rangep-Grp*: *Rangep* (*BNF-Def.Grp* A f) $= (\lambda x.\ x \in f$ ‘ $A)$
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *fun-ord-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total* C
shows ((A ===> B ===> (=)) ===> (C ===> A) ===> (C ===> B) ===> (=)) *fun-ord fun-ord*
⟨*proof*⟩

lemma *Plus-parametric* [*transfer-rule*]:
(*rel-set* A ===> *rel-set* B ===> *rel-set* (*rel-sum* A B)) (<+>) (<+>)
⟨*proof*⟩

lemma *pred-fun-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total* A
shows ((A ===> (=)) ===> (B ===> (=)) ===> (A ===> B) ===> (=)) *pred-fun pred-fun*
⟨*proof*⟩

lemma *rel-fun-eq-OO*: ((=) ===> A) *OO* ((=) ===> B) = ((=) ===> A *OO* B)
⟨*proof*⟩

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*⟩

lemma *Domainp-eq*: *Domainp* (=) = (λ-. *True*)
⟨*proof*⟩

lemma *rel-fun-eq-onpI*: *eq-onp* (*pred-fun P Q*) *f g* ⟹ *rel-fun* (*eq-onp P*) (*eq-onp Q*) *f g*
⟨*proof*⟩

lemma *bi-unique-eq-onp*: *bi-unique* (*eq-onp P*)
⟨*proof*⟩

lemma *rel-fun-eq-conversep*: **includes** *lifting-syntax* **shows** (A^{-1-1} ===> (=)) = (*A* ===> (=))$^{-1-1}$
⟨*proof*⟩

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*
  ⟨*proof*⟩

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

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

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

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

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

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

A A′ (*f*, *g*))
and *rel-witness-fun2*: *rel-fun A′* (λ(*x*, *y′*) *y*. *y* = *y′* ∧ *B x y′*) (*rel-witness-fun A A′* (*f*, *g*)) *g*
⟨*proof*⟩

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

1.11 Arithmetic

lemma *abs-diff-triangle-ineq2*: |*a* − *b* :: - :: *ordered-ab-group-add-abs*| ≤ |*a* − *c*| + |*c* − *b*|
⟨*proof*⟩

lemma (**in** *ordered-ab-semigroup-add*) *add-left-mono-trans*:
⟦ *x* ≤ *a* + *b*; *b* ≤ *c* ⟧ ⟹ *x* ≤ *a* + *c*
⟨*proof*⟩

lemma *of-nat-le-one-cancel-iff* [*simp*]:
fixes *n* :: *nat* **shows** *real n* ≤ *1* ⟷ *n* ≤ *1*
⟨*proof*⟩

lemma (**in** *linordered-semidom*) *mult-right-le*: *c* ≤ *1* ⟹ *0* ≤ *a* ⟹ *c* ∗ *a* ≤ *a*
⟨*proof*⟩

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

lemma *fun-ordD*: *fun-ord ord f g* ⟹ *ord* (*f x*) (*g x*)
⟨*proof*⟩

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*) (λ*x*. *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* ⟧ ⟹ *P* (*f x*) (*g y*)
shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
⟨*proof*⟩

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* (λ*f*. *U1* (*F* (*C1 f*)) *x*)
and *G*: ⋀*y*. *monotone* (*fun-ord ordb*) *ordb* (λ*g*. *U2* (*G* (*C2 g*)) *y*)
and *eq1*: *f* ≡ *C1* (*ccpo.fixp* (*fun-lub luba*) (*fun-ord orda*) (λ*f*. *U1* (*F* (*C1 f*))))
and *eq2*: *g* ≡ *C2* (*ccpo.fixp* (*fun-lub lubb*) (*fun-ord ordb*) (λ*g*. *U2* (*G* (*C2 g*))))

and *inverse*: $\bigwedge f.\ U1\ (C1\ f) = f$
and *inverse2*: $\bigwedge g.\ U2\ (C2\ g) = g$
and *adm*: *ccpo.admissible* (*prod-lub* (*fun-lub luba*) (*fun-lub lubb*)) (*rel-prod* (*fun-ord orda*) (*fun-ord ordb*)) ($\lambda x.\ P\ (fst\ x)\ (snd\ x)$)
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 $P\ (U1\ f)\ (U2\ g)$
⟨*proof*⟩

lemmas *parallel-fixp-induct-strong-1-1* = *parallel-fixp-induct-strong-uc*[
of - - - - $\lambda x.\ x$ - $\lambda x.\ x$ $\lambda x.\ x$ - $\lambda x.\ x$,
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 $F :: 'c \Rightarrow 'c$ **and**
$U :: 'c \Rightarrow 'b \Rightarrow 'a\ option$ **and**
$C :: ('b \Rightarrow 'a\ option) \Rightarrow 'c$ **and**
$P :: 'b \Rightarrow 'a \Rightarrow bool$
assumes *mono*: $\bigwedge x.\ mono\text{-}option\ (\lambda f.\ U\ (F\ (C\ f))\ x)$
assumes *eq*: $f \equiv C$ (*ccpo.fixp* (*fun-lub* (*flat-lub None*)) (*fun-ord option-ord*) ($\lambda f.\ U\ (F\ (C\ f))$)))
assumes *inverse2*: $\bigwedge f.\ U\ (C\ f) = f$
assumes *step*: $\bigwedge g\ x\ y.\ [\![\ \bigwedge x\ y.\ U\ g\ x = Some\ y \Longrightarrow P\ x\ y;\ U\ (F\ g)\ x = Some\ y;\ \bigwedge x.\ option\text{-}ord\ (U\ g\ x)\ (U\ f\ x)\ ]\!] \Longrightarrow P\ x\ y$
assumes *defined*: $U\ f\ x = Some\ y$
shows $P\ x\ y$
⟨*proof*⟩

⟨*ML*⟩

lemma *bot-fun-least* [*simp*]: (λ-. $bot :: 'a :: order\text{-}bot$) $\leq x$
⟨*proof*⟩

lemma *fun-ord-conv-rel-fun*: *fun-ord* = *rel-fun* (=)
⟨*proof*⟩

inductive *finite-chains* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
for *ord*
where *finite-chainsI*: ($\bigwedge Y.\ Complete\text{-}Partial\text{-}Order.chain\ ord\ Y \Longrightarrow finite\ Y$) $\Longrightarrow$ *finite-chains ord*

lemma *finite-chainsD*: ⟦ *finite-chains ord*; *Complete-Partial-Order.chain ord Y* ⟧ ⟹ *finite Y*
⟨*proof*⟩

lemma *finite-chains-flat-ord* [*simp*, *intro*!]: *finite-chains* (*flat-ord x*)
⟨*proof*⟩

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*⟩

lemma *rel-fun-curry*: **includes** *lifting-syntax* **shows**
  (*A* ===> *B* ===> *C*) *f g* ⟷ (*rel-prod A B* ===> *C*) (*case-prod f*) (*case-prod g*)
⟨*proof*⟩

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*⟩

lemma (**in** *ccpo*) *admissible-le-mono*:
  assumes *monotone* (≤) (≤) *f*
  shows *ccpo.admissible Sup* (≤) (λ*x*. *x* ≤ *f x*)
⟨*proof*⟩

lemma (**in** *ccpo*) *fixp-induct-strong2*:
  assumes *adm*: *ccpo.admissible Sup* (≤) *P*
  and *mono*: *monotone* (≤) (≤) *f*
  and *bot*: *P* (⨆{})
  and *step*: ⋀*x*. ⟦ *x* ≤ *ccpo-class.fixp f*; *x* ≤ *f x*; *P x* ⟧ ⟹ *P* (*f x*)
  shows *P* (*ccpo-class.fixp f*)
⟨*proof*⟩

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

lemma *fixp-induct-strong2-uc*:
  fixes *F* :: ′*c* ⇒ ′*c*
    and *U* :: ′*c* ⇒ ′*b* ⇒ ′*a*
    and *C* :: (′*b* ⇒ ′*a*) ⇒ ′*c*
    and *P* :: (′*b* ⇒ ′*a*) ⇒ *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 $P\ (U\ f)$
⟨*proof*⟩

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* $(\leq)$ $(\leq)$ f
and *ge*: $\bigwedge y.\ f\ y \leq y \Longrightarrow x \leq y$
shows $x \leq$ *ccpo.fixp Sup* $(\leq)$ f
⟨*proof*⟩

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* $(\lambda x.\ g\ (f\ x)) = g$ (*ccpo.fixp lub2 leq2* $(\lambda x.\ f\ (g\ x)))$
⟨*proof*⟩

lemma *fixp-lfp-parametric-eq*:
includes *lifting-syntax*
assumes f: $\bigwedge x.$ *lfp.mono-body* $(\lambda f.\ F\ f\ x)$
and g: $\bigwedge x.$ *lfp.mono-body* $(\lambda f.\ G\ f\ x)$
and *param*: $((A$ ===> $(=))$ ===> A ===> $(=))$ $F\ G$
shows $(A$ ===> $(=))$ (*lfp.fixp-fun* F) (*lfp.fixp-fun* G)
⟨*proof*⟩

lemma *mono2mono-map-option*[*THEN option.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-map-option*: *monotone option-ord option-ord* (*map-option f*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *mono2mono-set-option* [*THEN lfp.mono2mono*]:
shows *monotone-set-option*: *monotone option-ord* (⊆) *set-option*
⟨*proof*⟩

lemma *mcont2mcont-set-option* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-set-option*: *mcont* (*flat-lub None*) *option-ord Union* (⊆) *set-option*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *map-option-mono* [*partial-function-mono*]:
mono-option B ⟹ *mono-option* (λ*f*. *map-option g* (*B f*))
⟨*proof*⟩

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*⟩

lemma (**in** *comp-fun-commute*) *fold-invariant-insert* [*consumes 1*, *case-names start step*]:
assumes *fin*: *finite A*
and *start*: *I* {} *s*
and *step*: ⋀*x s A′*. $[\![$ *I A′ s*; *x* ∉ *A′*; *x* ∈ *A*; *A′* ⊆ *A* $]\!]$ ⟹ *I* (*insert x A′*) (*f x s*)
shows *I A* (*Finite-Set.fold f s A*)
⟨*proof*⟩

lemma (**in** *comp-fun-idem*) *fold-set-union*:
assumes *finite A finite B*
shows *Finite-Set.fold f z* (*A* ∪ *B*) = *Finite-Set.fold f* (*Finite-Set.fold f z A*) *B*
⟨*proof*⟩

1.14 Parametrisation of transfer rules

⟨*ML*⟩

1.15 Lists

lemma *nth-eq-tlI*: *xs* ! *n* = *z* ⟹ (*x* # *xs*) ! *Suc n* = *z*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *disjointpD'*:
⟦ *disjointp xs*; *P x*; *Q x*; *xs* ! *n* = *P*; *xs* ! *m* = *Q*; *n* < *length xs*; *m* < *length xs* ⟧ ⟹ *n* = *m*
⟨*proof*⟩

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

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

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*}
⟨*proof*⟩

lemma *nlists-empty*: *nlists* {} *n* = (*if n* = *0 then* {[]} *else* {})
⟨*proof*⟩

lemma *nlists-empty-gt0* [*simp*]: *n* > *0* ⟹ *nlists* {} *n* = {}
⟨*proof*⟩

lemma *nlists-0* [*simp*]: *nlists A 0* = {[]}
⟨*proof*⟩

lemma *Cons-in-nlists-Suc* [*simp*]: *x* # *xs* ∈ *nlists A* (*Suc n*) ⟷ *x* ∈ *A* ∧ *xs* ∈ *nlists A n*
⟨*proof*⟩

lemma *Nil-in-nlists* [*simp*]: [] ∈ *nlists A n* ⟷ *n* = *0*
⟨*proof*⟩

lemma *Cons-in-nlists-iff*: $x \# xs \in nlists\ A\ n \longleftrightarrow (\exists n'.\ n = Suc\ n' \land x \in A \land xs \in nlists\ A\ n')$
⟨*proof*⟩

lemma *in-nlists-Suc-iff*: $xs \in nlists\ A\ (Suc\ n) \longleftrightarrow (\exists x\ xs'.\ xs = x \# xs' \land x \in A \land xs' \in nlists\ A\ n)$
⟨*proof*⟩

lemma *nlists-Suc*: $nlists\ A\ (Suc\ n) = (\bigcup x{\in}A.\ (\#)\ x\ `\ nlists\ A\ n)$
⟨*proof*⟩

lemma *replicate-in-nlists* [*simp*, *intro*]: $x \in A \Longrightarrow replicate\ n\ x \in nlists\ A\ n$
⟨*proof*⟩

lemma *nlists-eq-empty-iff* [*simp*]: $nlists\ A\ n = \{\} \longleftrightarrow n > 0 \land A = \{\}$
⟨*proof*⟩

lemma *finite-nlists* [*simp*]: $finite\ A \Longrightarrow finite\ (nlists\ A\ n)$
⟨*proof*⟩

lemma *finite-nlistsD*:
  assumes $finite\ (nlists\ A\ n)$
  shows $finite\ A \lor n = 0$
⟨*proof*⟩

lemma *finite-nlists-iff*: $finite\ (nlists\ A\ n) \longleftrightarrow finite\ A \lor n = 0$
⟨*proof*⟩

lemma *card-nlists*: $card\ (nlists\ A\ n) = card\ A\ \widehat{}\ n$
⟨*proof*⟩

lemma *in-nlists-UNIV*: $xs \in nlists\ UNIV\ n \longleftrightarrow length\ xs = n$
⟨*proof*⟩

1.15.2 The type of lists of a given length

typedef (**overloaded**) $('a, 'b :: len0)\ nlist = nlists\ (UNIV :: 'a\ set)\ (LENGTH('b))$
⟨*proof*⟩

setup-lifting *type-definition-nlist*

1.16 Streams and infinite lists

primrec $sprefix :: 'a\ list \Rightarrow 'a\ stream \Rightarrow bool$ **where**
  sprefix-Nil: $sprefix\ []\ ys = True$
| *sprefix-Cons*: $sprefix\ (x \# xs)\ ys \longleftrightarrow x = shd\ ys \land sprefix\ xs\ (stl\ ys)$

lemma *sprefix-append*: $sprefix\ (xs\ @\ ys)\ zs \longleftrightarrow sprefix\ xs\ zs \land sprefix\ ys\ (sdrop\ (length\ xs)\ zs)$

⟨*proof*⟩

lemma *sprefix-stake-same* [*simp*]: *sprefix* (*stake n xs*) *xs*
⟨*proof*⟩

lemma *sprefix-same-imp-eq*:
assumes *sprefix xs ys sprefix xs′ ys*
and *length xs* = *length xs′*
shows *xs* = *xs′*
⟨*proof*⟩

lemma *sprefix-shift-same* [*simp*]:
sprefix xs (*xs* @− *ys*)
⟨*proof*⟩

lemma *sprefix-shift* [*simp*]:
length xs ≤ *length ys* ⟹ *sprefix xs* (*ys* @− *zs*) ⟷ *prefix xs ys*
⟨*proof*⟩

lemma *prefixeq-stake2* [*simp*]: *prefix xs* (*stake n ys*) ⟷ *length xs* ≤ *n* ∧ *sprefix xs ys*
⟨*proof*⟩

lemma *tlength-eq-infinity-iff*: *tlength xs* = ∞ ⟷ ¬ *tfinite xs*
including *tllist.lifting* ⟨*proof*⟩

1.17 Monomorphic monads

context includes *lifting-syntax* **begin**
⟨*ML*⟩

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*
⟨*proof*⟩

lemma *bind-option-K*:
⋀*monad*. (*x* = *None* ⟹ *m* = *fail*) ⟹ *bind-option fail x* (λ-. *m*) = *m*
⟨*proof*⟩

end

lemma *bind-option-option* [*simp*]: *monad.bind-option None* = *Option.bind*
⟨*proof*⟩

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

lemma *hom-bind-option*: h (*monad.bind-option fail1 x f*) = *monad.bind-option fail2 x* ($h \circ f$)
⟨*proof*⟩

end

lemma *bind-option-set* [*simp*]: *monad.bind-option fail-set* = ($\lambda x\, f.\ \bigcup$ (f ‘ *set-option x*))
⟨*proof*⟩

lemma *run-bind-option-stateT* [*simp*]:
$\bigwedge$*more. run-state* (*monad.bind-option* (*fail-state fail*) *x f*) *s* =
monad.bind-option fail x (λy. *run-state* (*f y*) *s*)
⟨*proof*⟩

lemma *run-bind-option-envT* [*simp*]:
$\bigwedge$*more. run-env* (*monad.bind-option* (*fail-env fail*) *x f*) *s* =
monad.bind-option fail x (λy. *run-env* (*f y*) *s*)
⟨*proof*⟩

1.18 Measures

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

lemma (**in** *sigma-algebra*) *sets-Collect-countable-Ex1*:
($\bigwedge i$ :: *'i* :: *countable*. $\{x \in \Omega.\ P\ i\ x\} \in M$) $\Longrightarrow$ $\{x \in \Omega.\ \exists!i.\ P\ i\ x\} \in M$
⟨*proof*⟩

lemma *pred-countable-Ex1* [*measurable*]:
($\bigwedge i$ :: - :: *countable*. *Measurable.pred M* ($\lambda x.\ P\ i\ x$))
$\Longrightarrow$ *Measurable.pred M* ($\lambda x.\ \exists!i.\ P\ i\ x$)
⟨*proof*⟩

lemma *measurable-snd-count-space* [*measurable*]:
$A \subseteq B \Longrightarrow$ *snd* $\in$ *measurable* (*M1* $\bigotimes_M$ *count-space A*) (*count-space B*)
⟨*proof*⟩

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

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

1.19 Sequence space

lemma (**in** *sequence-space*) *nn-integral-split*:
  assumes $f[measurable]$: $f \in$ *borel-measurable* S
  shows $(\int^+\omega.\ f\ \omega\ \partial S) = (\int^+\omega.\ (\int^+\omega'.\ f\ (comb\text{-}seq\ i\ \omega\ \omega')\ \partial S)\ \partial S)$
⟨*proof*⟩

lemma (**in** *sequence-space*) *prob-Collect-split*:
  assumes $f[measurable]$: $\{x{\in}space\ S.\ P\ x\} \in sets\ S$
  shows $\mathcal{P}(x\ in\ S.\ P\ x) = (\int^+x.\ \mathcal{P}(x'\ in\ S.\ P\ (comb\text{-}seq\ i\ x\ x'))\ \partial S)$
⟨*proof*⟩

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*
⟨*proof*⟩

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
  ⟨*proof*⟩
end

lemma *pred-rel-pmf*:
  ⟦ *pred-pmf P p*; *rel-pmf R p q* ⟧ ⟹ *pred-pmf* (*Imagep R P*) *q*
⟨*proof*⟩

lemma *pmf-rel-mono'*: ⟦ *rel-pmf P x y*; $P \le Q$ ⟧ ⟹ *rel-pmf Q x y*
⟨*proof*⟩

lemma *rel-pmf-eqI* [*simp*]: *rel-pmf* (=) *x x*
⟨*proof*⟩

lemma *rel-pmf-bind-reflI*:
  (⋀*x*. $x \in$ *set-pmf p* ⟹ *rel-pmf R* (*f x*) (*g x*))
  ⟹ *rel-pmf R* (*bind-pmf p f*) (*bind-pmf p g*)
⟨*proof*⟩

lemma *pmf-pred-mono-strong*:
  ⟦ *pred-pmf P p*; ⋀*a*. ⟦ $a \in$ *set-pmf p*; *P a* ⟧ ⟹ *P' a* ⟧ ⟹ *pred-pmf P' p*
⟨*proof*⟩

lemma *rel-pmf-restrict-relpI* [*intro?*]:
  ⟦ *rel-pmf R x y*; *pred-pmf P x*; *pred-pmf Q y* ⟧ ⟹ *rel-pmf* ($R \upharpoonright P \otimes Q$) *x y*
⟨*proof*⟩

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*⟩

lemma *rel-pmf-restrict-relp-iff*:
rel-pmf (*R* ↾ *P* ⊗ *Q*) *x y* ⟷ *rel-pmf R x y* ∧ *pred-pmf P x* ∧ *pred-pmf Q y*
⟨*proof*⟩

lemma *rel-pmf-OO-trans* [*trans*]:
⟦ *rel-pmf R p q*; *rel-pmf S q r* ⟧ ⟹ *rel-pmf* (*R OO S*) *p r*
⟨*proof*⟩

lemma *pmf-pred-map* [*simp*]: *pred-pmf P* (*map-pmf f p*) = *pred-pmf* (*P* ∘ *f*) *p*
⟨*proof*⟩

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

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

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

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

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*
⟨*proof*⟩

lemma *pred-pmf-pair* [*simp*]:
pred-pmf P (*pair-pmf p q*) = *pred-pmf* (λ*x*. *pred-pmf* (*P* ∘ *Pair x*) *q*) *p*
⟨*proof*⟩

lemma *pred-pmf-join* [*simp*]: *pred-pmf P* (*join-pmf p*) = *pred-pmf* (*pred-pmf P*) *p*
⟨*proof*⟩

lemma *pred-pmf-bernoulli* [*simp*]: ⟦ *0* < *p*; *p* < *1* ⟧ ⟹ *pred-pmf P* (*bernoulli-pmf p*) = *All P*
⟨*proof*⟩

lemma *pred-pmf-geometric* [*simp*]: ⟦ *0* < *p*; *p* < *1* ⟧ ⟹ *pred-pmf P* (*geometric-pmf p*) = *All P*
⟨*proof*⟩

lemma *pred-pmf-poisson* [*simp*]: *0* < *rate* ⟹ *pred-pmf P* (*poisson-pmf rate*) = *All P*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pred-pmf-conj* [*simp*]: *pred-pmf* (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *pred-pmf P x* ∧ *pred-pmf Q x*)
⟨*proof*⟩

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

lemma *rel-pmf-of-setI*:
assumes *A*: *A* ≠ {} *finite A*
and *B*: *B* ≠ {} *finite B*
and *card*: ⋀*X*. *X* ⊆ *A* ⟹ *card B* ∗ *card X* ≤ *card A* ∗ *card* {*y*∈*B*. ∃ *x*∈*X*. *R x y*}
shows *rel-pmf R* (*pmf-of-set A*) (*pmf-of-set B*)
⟨*proof*⟩

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

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

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

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

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*⟩

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* $(A \times B)$
⟨*proof*⟩

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*⟩

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* $\cap A \neq \{\}$
⟨*proof*⟩

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

1.21 Subprobability mass functions

lemma *ord-spmf-return-spmf1*: *ord-spmf R (return-spmf x) p* $\longleftrightarrow$ *lossless-spmf p* $\wedge$ ($\forall$ *y*$\in$*set-spmf p. R x y*)
⟨*proof*⟩

lemma *ord-spmf-conv*:
ord-spmf R = rel-spmf R OO ord-spmf (=)
⟨*proof*⟩

lemma *ord-spmf-expand*:
NO-MATCH (=) *R* $\Longrightarrow$ *ord-spmf R = rel-spmf R OO ord-spmf* (=)
⟨*proof*⟩

lemma *ord-spmf-eqD-measure*: *ord-spmf* (=) *p q* $\Longrightarrow$ *measure (measure-spmf p)*

$A \le$ *measure* (*measure-spmf* q) A
⟨*proof*⟩

lemma *ord-spmf-measureD*:
  assumes *ord-spmf* R p q
  shows *measure* (*measure-spmf* p) $A \le$ *measure* (*measure-spmf* q) $\{y.\ \exists x{\in}A.\ R\ x\ y\}$
    (**is** *?lhs* $\le$ *?rhs*)
⟨*proof*⟩

lemma *ord-spmf-bind-pmfI1*:
  $(\bigwedge x.\ x \in$ *set-pmf* $p \Longrightarrow$ *ord-spmf* R $(f\ x)$ $q) \Longrightarrow$ *ord-spmf* R (*bind-pmf* p f) q
  ⟨*proof*⟩

lemma *ord-spmf-bind-spmfI1*:
  $(\bigwedge x.\ x \in$ *set-spmf* $p \Longrightarrow$ *ord-spmf* R $(f\ x)$ $q) \Longrightarrow$ *ord-spmf* R (*bind-spmf* p f) q
⟨*proof*⟩

lemma *spmf-of-set-empty*: *spmf-of-set* $\{\}$ = *return-pmf None*
⟨*proof*⟩

lemma *rel-spmf-of-setI*:
  assumes *card*: $\bigwedge X.\ X \subseteq A \Longrightarrow$ *card* $B *$ *card* $X \le$ *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)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nn-integral-spmf-neq-infinity*: $(\int^+ x.$ *spmf* p x ∂*count-space UNIV*$) \neq \infty$
⟨*proof*⟩

lemma *return-pmf-bind-option*:
  return-pmf (*Option.bind* x f) = *bind-spmf* (*return-pmf* x) (*return-pmf* $\circ f$)
⟨*proof*⟩

lemma *rel-spmf-pos-distr*: *rel-spmf* A *OO* *rel-spmf* $B \le$ *rel-spmf* (A *OO* B)
⟨*proof*⟩

lemma *rel-spmf-OO-trans* [*trans*]:
  ⟦ *rel-spmf* R p q; *rel-spmf* S q r ⟧ $\Longrightarrow$ *rel-spmf* (R *OO* S) p r
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-spmf-eq-empty*: *set-spmf p* = {} ⟷ *p* = *return-pmf None*
⟨*proof*⟩

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*⟩

lemma *lossless-spmfD-set-spmf-nonempty*: *lossless-spmf p* ⟹ *set-spmf p* ≠ {}
⟨*proof*⟩

lemma *set-spmf-return-pmf*: *set-spmf* (*return-pmf x*) = *set-option x*
⟨*proof*⟩

lemma *bind-spmf-pmf-assoc*: *bind-spmf* (*bind-pmf p f*) *g* = *bind-pmf p* (λ*x*. *bind-spmf* (*f x*) *g*)
⟨*proof*⟩

lemma *bind-spmf-of-set*: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *bind-spmf* (*spmf-of-set A*) *f* = *bind-pmf* (*pmf-of-set A*) *f*
⟨*proof*⟩

lemma *bind-spmf-map-pmf*:
bind-spmf (*map-pmf f p*) *g* = *bind-pmf p* (λ*x*. *bind-spmf* (*return-pmf* (*f x*)) *g*)
⟨*proof*⟩

lemma *rel-spmf-eqI* [*simp*]: *rel-spmf* (=) *x x*
⟨*proof*⟩

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

lemma *ord-spmf-return-spmf* [*simp*]: *ord-spmf* (=) (*return-spmf x*) *p* ⟷ *p* =

return-spmf x
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *return-pmf-map-option-conv-bind*:
  return-pmf (*map-option f x*) = *bind-spmf* (*return-pmf x*) (*return-spmf* ∘ *f*)
⟨*proof*⟩

lemma *lossless-return-pmf-iff* [*simp*]: *lossless-spmf* (*return-pmf x*) ⟷ *x* ≠ *None*
⟨*proof*⟩

lemma *lossless-map-pmf*: *lossless-spmf* (*map-pmf f p*) ⟷ (∀ *x* ∈ *set-pmf p*. *f x* ≠ *None*)
⟨*proof*⟩

lemma *bind-pmf-spmf-assoc*:
  g None = *return-pmf None*
  ⟹ *bind-pmf* (*bind-spmf p f*) *g* = *bind-spmf p* (λ*x*. *bind-pmf* (*f x*) *g*)
⟨*proof*⟩

abbreviation *pred-spmf* :: (′*a* ⇒ *bool*) ⇒ ′*a spmf* ⇒ *bool*
where *pred-spmf P* ≡ *pred-pmf* (*pred-option P*)

lemma *pred-spmf-def*: *pred-spmf P p* ⟷ (∀ *x*∈*set-spmf p*. *P x*)
⟨*proof*⟩

lemma *spmf-pred-mono-strong*:
  ⟦ *pred-spmf P p*; ⋀*a*. ⟦ *a* ∈ *set-spmf p*; *P a* ⟧ ⟹ *P*′ *a* ⟧ ⟹ *pred-spmf P*′ *p*
⟨*proof*⟩

lemma *spmf-Domainp-rel*: *Domainp* (*rel-spmf R*) = *pred-spmf* (*Domainp R*)
⟨*proof*⟩

lemma *rel-spmf-restrict-relpI* [*intro?*]:
  ⟦ *rel-spmf R p q*; *pred-spmf P p*; *pred-spmf Q q* ⟧ ⟹ *rel-spmf* (*R* ↾ *P* ⊗ *Q*) *p q*
⟨*proof*⟩

lemma *rel-spmf-restrict-relpE* [*elim?*]:
  assumes *rel-spmf* (*R* ↾ *P* ⊗ *Q*) *x y*
  obtains *rel-spmf R x y pred-spmf P x pred-spmf Q y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *spmf-pred-map*: *pred-spmf P* (*map-spmf f p*) = *pred-spmf* ($P \circ f$) *p*
⟨*proof*⟩

lemma *pred-spmf-bind* [*simp*]: *pred-spmf P* (*bind-spmf p f*) = *pred-spmf* (*pred-spmf* $P \circ f$) *p*
⟨*proof*⟩

lemma *pred-spmf-return*: *pred-spmf P* (*return-spmf x*) = *P x*
⟨*proof*⟩

lemma *pred-spmf-return-pmf-None*: *pred-spmf P* (*return-pmf None*)
⟨*proof*⟩

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

lemma *pred-spmf-of-set* [*simp*]: *pred-spmf P* (*spmf-of-set A*) = (*finite A* $\longrightarrow$ *Ball A P*)
⟨*proof*⟩

lemma *pred-spmf-assert-spmf* [*simp*]: *pred-spmf P* (*assert-spmf b*) = (*b* $\longrightarrow$ *P* ())
⟨*proof*⟩

lemma *pred-spmf-pair* [*simp*]:
pred-spmf P (*pair-spmf p q*) = *pred-spmf* (λx. *pred-spmf* ($P \circ$ *Pair x*) *q*) *p*
⟨*proof*⟩

lemma *set-spmf-try* [*simp*]:
set-spmf (*try-spmf p q*) = *set-spmf p* $\cup$ (*if lossless-spmf p then* {} *else set-spmf q*)
⟨*proof*⟩

lemma *try-spmf-bind-out1*:
($\bigwedge x$. *lossless-spmf* (*f x*)) $\Longrightarrow$ *bind-spmf* (*TRY p ELSE q*) *f* = *TRY* (*bind-spmf p f*) *ELSE* (*bind-spmf q f*)
⟨*proof*⟩

lemma *pred-spmf-try* [*simp*]:
pred-spmf P (*try-spmf p q*) = (*pred-spmf P p* $\wedge$ ($\neg$ *lossless-spmf p* $\longrightarrow$ *pred-spmf P q*))
⟨*proof*⟩

lemma *pred-spmf-cond* [*simp*]:
pred-spmf P (*cond-spmf p A*) = *pred-spmf* (λx. $x \in A \longrightarrow P\ x$) *p*

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pred-spmf-conj*: *pred-spmf* (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *pred-spmf P x* ∧ *pred-spmf Q x*)
⟨*proof*⟩

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

lemma *mono2mono-return-pmf*[*THEN spmf.mono2mono*, *simp*, *cont-intro*]:
shows *monotone-return-pmf*: *monotone option-ord* (*ord-spmf* (=)) *return-pmf*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pred-spmf-top*:
pred-spmf (λ-. *True*) = (λ-. *True*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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

rel-witness-spmf A = *map-pmf rel-witness-option* ∘ *rel-witness-pmf* (*rel-option A*)

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

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *mk-lossless-eq-return-pmf-None* [*simp*]: *mk-lossless p* = *return-pmf None* ⟷ *p* = *return-pmf None*
⟨*proof*⟩

lemma *return-pmf-None-eq-mk-lossless* [*simp*]: *return-pmf None* = *mk-lossless p* ⟷ *p* = *return-pmf None*
⟨*proof*⟩

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

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

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

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

lemma *rel-spmf-restrict-spmfI*:
rel-spmf (λ*x y*. (*x* ∈ *A* ∧ *y* ∈ *B* ∧ *R x y*) ∨ *x* ∉ *A* ∧ *y* ∉ *B*) *p q*
⟹ *rel-spmf R* (*restrict-spmf p A*) (*restrict-spmf q B*)
⟨*proof*⟩

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

lemma *cond-spmf-bind*:
cond-spmf (*bind-spmf p f*) *A* = *mk-lossless* (*p* ≫= (λ*x*. *f x* ↿ *A*))
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *type-definition-option-spmf*: *type-definition return-pmf the-pmf* {*x.* ∃ *y* :: *'a option. x* = *return-pmf y*}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-option-spmf-transfer* [*transfer-parametric map-spmf-parametric*, *transfer-rule*]:
(((=) ===> (=)) ===> *cr-spmf-option* ===> *cr-spmf-option*) *map-spmf map-option*
⟨*proof*⟩

lemma *fail-option-spmf-transfer* [*transfer-parametric return-spmf-None-parametric*, *transfer-rule*]:
cr-spmf-option (*return-pmf None*) *None*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set-option-spmf-transfer* [*transfer-parametric set-spmf-parametric*, *transfer-rule*]:
(*cr-spmf-option* ===> *rel-set* (=)) *set-spmf set-option*
⟨*proof*⟩

lemma *rel-option-spmf-transfer* [*transfer-parametric rel-spmf-parametric*, *transfer-rule*]:
(((=) ===> (=) ===> (=)) ===> *cr-spmf-option* ===> *cr-spmf-option* ===> (=)) *rel-spmf rel-option*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-option-le-spmf-transfer* [*transfer-rule*]:
(((=) ===> (=)) ===> *cr-option-le-spmf* ===> *cr-option-le-spmf*) *map-option*

map-spmf
⟨*proof*⟩

lemma *bind-option-le-spmf-transfer* [*transfer-rule*]:
(*cr-option-le-spmf* ===> ((=) ===> *cr-option-le-spmf*) ===> *cr-option-le-spmf*)
Option.bind bind-spmf
⟨*proof*⟩

end

end

interpretation *rel-spmf-characterisation* ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *if-distrib-bind-spmf2* [*if-distribs*]:
bind-spmf x (*λy. if b then f y else g y*) = (*if b then bind-spmf x f else bind-spmf x g*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* (*λf. Set.bind x* (*λx.* {*f x*}))
⟨*proof*⟩

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

lemma *in-ap-setI*: $[\![f' \in f;\ x' \in x]\!] \Longrightarrow f'\ x' \in f \diamond x$
⟨*proof*⟩

lemma *in-ap-setE* [*elim*!]:
$[\![x \in f \diamond y;\ \bigwedge f'\ y'.\ [\![x = f'\ y';\ f' \in f;\ y' \in y]\!] \Longrightarrow thesis]\!] \Longrightarrow thesis$
⟨*proof*⟩

lemma *in-ap-pure-set* [*iff*]: $x \in \{f\} \diamond y \longleftrightarrow (\exists\, y' \in y.\ x = f\ y')$
⟨*proof*⟩

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*)))
⟨*proof*⟩

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

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

lemma *ap-spmf-id*: *pure-spmf* (λ*x*. *x*) ⋄ *x* = *x*
⟨*proof*⟩

lemma *ap-spmf-comp*: *pure-spmf* (∘) ⋄ *u* ⋄ *v* ⋄ *w* = *u* ⋄ (*v* ⋄ *w*)
⟨*proof*⟩

lemma *ap-spmf-homo*: *pure-spmf f* ⋄ *pure-spmf x* = *pure-spmf* (*f x*)
⟨*proof*⟩

lemma *ap-spmf-interchange*: *u* ⋄ *pure-spmf x* = *pure-spmf* (λ*f*. *f x*) ⋄ *u*
⟨*proof*⟩

lemma *ap-spmf-C*: *return-spmf* ($\lambda f\ x\ y.\ f\ y\ x$) $\diamond$ *f* $\diamond$ *x* $\diamond$ *y* = *f* $\diamond$ *y* $\diamond$ *x*
⟨*proof*⟩

applicative *spmf* (*C*)
for
pure: *pure-spmf*
ap: *ap-spmf*
⟨*proof*⟩

lemma *set-ap-spmf* [*simp*]: *set-spmf* (*p* $\diamond$ *q*) = *set-spmf p* $\diamond$ *set-spmf q*
⟨*proof*⟩

lemma *bind-ap-spmf*: *bind-spmf* (*p* $\diamond$ *x*) *f* = *bind-spmf p* ($\lambda p.\ x \gg= (\lambda x.\ f\ (p\ x))$)
⟨*proof*⟩

lemma *bind-pmf-ap-return-spmf* [*simp*]: *bind-pmf* (*ap-spmf* (*return-spmf f*) *p*) *g* = *bind-pmf p* (*g* $\circ$ *map-option f*)
⟨*proof*⟩

lemma *map-spmf-conv-ap* [*applicative-unfold*]: *map-spmf f p* = *return-spmf f* $\diamond$ *p*
⟨*proof*⟩

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** $x \oplus y \longleftrightarrow x \neq y$
⟨*proof*⟩

lemma *xor-commute*:
fixes *x y* :: *'a* :: {*semilattice-sup*,*semilattice-inf*,*uminus*}
shows $x \oplus y = y \oplus x$
⟨*proof*⟩

lemma *xor-assoc*:
fixes *x y* :: *'a* :: *boolean-algebra*
shows $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
⟨*proof*⟩

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

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *xor-inverse* [*simp*]:
  fixes *x* :: *'a* :: *boolean-algebra*
  shows *x* ⊕ *x* = *bot*
⟨*proof*⟩

lemma *xor-left-inverse* [*simp*]:
  fixes *x* :: *'a* :: *boolean-algebra*
  shows *x* ⊕ *x* ⊕ *y* = *y*
⟨*proof*⟩

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'*))
⟨*proof*⟩

lemma *xor-list-commute*: **fixes** *xs ys* :: *'a* :: {*semilattice-sup*,*semilattice-inf*,*uminus*} *list*
  shows *xs* [⊕] *ys* = *ys* [⊕] *xs*
⟨*proof*⟩

lemma *xor-list-assoc* [*simp*]:
  fixes *xs ys* :: *'a* :: *boolean-algebra list*
  shows (*xs* [⊕] *ys*) [⊕] *zs* = *xs* [⊕] (*ys* [⊕] *zs*)
⟨*proof*⟩

lemma *xor-list-left-commute*:
  fixes *xs ys zs* :: *'a* :: *boolean-algebra list*
  shows *xs* [⊕] (*ys* [⊕] *zs*) = *ys* [⊕] (*xs* [⊕] *zs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *xor-replicate-bot-right* [*simp*]:
fixes *xs* :: *'a* :: *boolean-algebra list*
shows ⟦ *length xs* ≤ *n*; *x* = *bot* ⟧ ⟹ *xs* [⊕] *replicate n x* = *xs*
⟨*proof*⟩

lemma *xor-replicate-bot-left* [*simp*]:
fixes *xs* :: *'a* :: *boolean-algebra list*
shows ⟦ *length xs* ≤ *n*; *x* = *bot* ⟧ ⟹ *replicate n x* [⊕] *xs* = *xs*
⟨*proof*⟩

lemma *xor-list-left-inverse* [*simp*]:
fixes *xs* :: *'a* :: *boolean-algebra list*
shows *length ys* ≤ *length xs* ⟹ *xs* [⊕] (*xs* [⊕] *ys*) = *ys*
⟨*proof*⟩

lemma *length-xor-list* [*simp*]: *length* (*xor-list xs ys*) = *min* (*length xs*) (*length ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *ap-envir-apply* [*simp*]: (*f* ⋄ *x*) *i* = *f i* (*x i*)

⟨*proof*⟩

definition *all-envir* :: (*'i*, *bool*) *envir* ⇒ *bool*
where *all-envir p* ⟷ (∀ *x*. *p x*)

lemma *all-envirI* [*Pure.intro*!, *intro*!]: (⋀*x*. *p x*) ⟹ *all-envir p*
⟨*proof*⟩

lemma *all-envirE* [*Pure.elim 2*, *elim*]: *all-envir p* ⟹ (*p x* ⟹ *thesis*) ⟹ *thesis*
⟨*proof*⟩

lemma *all-envirD*: *all-envir p* ⟹ *p x*
⟨*proof*⟩

definition *pred-envir* :: (*'a* ⇒ *bool*) ⇒ (*'i*, *'a*) *envir* ⇒ *bool*
where *pred-envir p f* = *all-envir* (*const p* ⋄ *f*)

lemma *pred-envir-conv*: *pred-envir p f* ⟷ (∀ *x*. *p* (*f x*))
⟨*proof*⟩

lemma *pred-envirI* [*Pure.intro*!, *intro*!]: (⋀*x*. *p* (*f x*)) ⟹ *pred-envir p f*
⟨*proof*⟩

lemma *pred-envirD*: *pred-envir p f* ⟹ *p* (*f x*)
⟨*proof*⟩

lemma *pred-envirE* [*Pure.elim 2*, *elim*]: *pred-envir p f* ⟹ (*p* (*f x*) ⟹ *thesis*) ⟹ *thesis*
⟨*proof*⟩

lemma *pred-envir-mono*: ⟦ *pred-envir p f*; ⋀*x*. *p* (*f x*) ⟹ *q* (*g x*) ⟧ ⟹ *pred-envir q g*
⟨*proof*⟩

definition *rel-envir* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'i*, *'a*) *envir* ⇒ (*'i*, *'b*) *envir* ⇒ *bool*
where *rel-envir p f g* ⟷ *all-envir* (*const p* ⋄ *f* ⋄ *g*)

lemma *rel-envir-conv*: *rel-envir p f g* ⟷ (∀ *x*. *p* (*f x*) (*g x*))
⟨*proof*⟩

lemma *rel-envir-conv-rel-fun*: *rel-envir* = *rel-fun* (=)
⟨*proof*⟩

lemma *rel-envirI* [*Pure.intro*!, *intro*!]: (⋀*x*. *p* (*f x*) (*g x*)) ⟹ *rel-envir p f g*
⟨*proof*⟩

lemma *rel-envirD*: *rel-envir p f g* ⟹ *p* (*f x*) (*g x*)
⟨*proof*⟩

lemma *rel-envirE* [*Pure.elim 2*, *elim*]: *rel-envir p f g* $\Longrightarrow$ (*p* (*f x*) (*g x*) $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*
⟨*proof*⟩

lemma *rel-envir-mono*: ⟦ *rel-envir p f g*; $\bigwedge x$. *p* (*f x*) (*g x*) $\Longrightarrow$ *q* (*f′ x*) (*g′ x*) ⟧ $\Longrightarrow$ *rel-envir q f′ g′*
⟨*proof*⟩

lemma *rel-envir-mono1*: ⟦ *pred-envir p f*; $\bigwedge x$. *p* (*f x*) $\Longrightarrow$ *q* (*f′ x*) (*g′ x*) ⟧ $\Longrightarrow$ *rel-envir q f′ g′*
⟨*proof*⟩

lemma *pred-envir-mono2*: ⟦ *rel-envir p f g*; $\bigwedge x$. *p* (*f x*) (*g x*) $\Longrightarrow$ *q* (*f′ x*) ⟧ $\Longrightarrow$ *pred-envir q f′*
⟨*proof*⟩

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*
⟨*proof*⟩

interpretation *set*: *partial-function-definitions* (⊆) *Union*
⟨*proof*⟩

lemma *fun-lub-Sup*: *fun-lub Sup* = (*Sup* :: - ⇒ - :: *complete-lattice*)
⟨*proof*⟩

lemma *set-admissible*: *set.admissible* (λ*f* :: ′*a* ⇒ ′*b set*. ∀ *x y*. *y* ∈ *f x* ⟶ *P x y*)
⟨*proof*⟩

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*: $\bigwedge x$. *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*: $\bigwedge f.\ U\ (C\ f) = f$
and *step*: $\bigwedge f\ x\ y.\ [\![\ \bigwedge x\ y.\ y \in U\ f\ x \Longrightarrow P\ x\ y;\ y \in U\ (F\ f)\ x\ ]\!] \Longrightarrow P\ x\ y$
and *enforce-variable-ordering*: $x = x$
and *elem*: $y \in U\ f\ x$
shows $P\ x\ y$
⟨*proof*⟩

lemma *fixp-Sup-le*:
defines $le \equiv ((\leq) :: \text{-} :: complete\text{-}lattice \Rightarrow \text{-})$
shows $ccpo.fixp\ Sup\ le = ccpo\text{-}class.fixp$
⟨*proof*⟩

lemma *fun-ord-le*: $fun\text{-}ord\ (\leq) = (\leq)$
⟨*proof*⟩

lemma *fixp-induct-set*:
fixes $F :: {}'c \Rightarrow {}'c$
and $U :: {}'c \Rightarrow {}'b \Rightarrow {}'a\ set$
and $C :: ({}'b \Rightarrow {}'a\ set) \Rightarrow {}'c$
and $P :: {}'b \Rightarrow {}'a \Rightarrow bool$
and x **and** y
assumes *mono*: $\bigwedge x.\ mono\text{-}set\ (\lambda f.\ U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C\ (ccpo.fixp\ (fun\text{-}lub\ Sup)\ (fun\text{-}ord\ (\leq))\ (\lambda f.\ U\ (F\ (C\ f))))$
and *inverse2*: $\bigwedge f.\ U\ (C\ f) = f$

and *step*: $\bigwedge f'\ x\ y.\ [\![\ \bigwedge x.\ U\ f'\ x = U\ f'\ x;\ y \in U\ (F\ (C\ (inf\ (U\ f)\ (\lambda x.\ \{y.\ P\ x\ y\}))))\ x\ ]\!] \Longrightarrow P\ x\ y$
— partial_function requires a quantifier over f', so let's have a fake one
and *elem*: $y \in U\ f\ x$
shows $P\ x\ y$
⟨*proof*⟩

⟨*ML*⟩

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*: $[\![\ mono\text{-}set\ A;\ mono\text{-}set\ B\ ]\!] \Longrightarrow mono\text{-}set\ (\lambda f.\ A\ f \cap B\ f)$
and *Diff-mono1*: $mono\text{-}set\ A \Longrightarrow mono\text{-}set\ (\lambda f.\ A\ f - X)$
and *image-mono*: $mono\text{-}set\ A \Longrightarrow mono\text{-}set\ (\lambda f.\ g\ `\ A\ f)$
and *vimage-mono*: $mono\text{-}set\ A \Longrightarrow mono\text{-}set\ (\lambda f.\ g\ -`\ A\ f)$
⟨*proof*⟩

partial-function (*set*) *test* :: $'a\ list \Rightarrow nat \Rightarrow bool \Rightarrow int\ set$

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*
⟨*proof*⟩

lemma *fun-lub-Inf*: *fun-lub Inf* = (*Inf* :: - ⇒ - :: *complete-lattice*)
⟨*proof*⟩

lemma *fun-ord-ge*: *fun-ord* (≥) = (≥)
⟨*proof*⟩

lemma *coset-admissible*: *coset.admissible* (λ*f* :: ′*a* ⇒ ′*b set*. ∀ *x y*. *P x y* ⟶ *y* ∈ *f x*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

abbreviation *mono-set*′ ≡ *monotone* (*fun-ord* (⊇)) (⊇)

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*′

($\lambda f.\ \bigcup y{\in}B\, f.\ C\ y\ f$)
and *set-bind-mono′*: $[\![$*mono-set′* B; $\bigwedge y.$ *mono-set′* ($\lambda f.\ C\ y\ f$)$]\!] \Longrightarrow$ *mono-set′* ($\lambda f.$ *Set.bind* ($B\ f$) ($\lambda y.\ C\ y\ f$))
and *Un-mono′*: $[\![$ *mono-set′* A; *mono-set′* B $]\!] \Longrightarrow$ *mono-set′* ($\lambda f.\ A\ f \cup B\ f$)
and *Int-mono′*: $[\![$ *mono-set′* A; *mono-set′* B $]\!] \Longrightarrow$ *mono-set′* ($\lambda f.\ A\ f \cap B\ f$)
⟨*proof*⟩

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 ∗: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow y = x \vee (P\ (Suc\ x)\ y \vee y \in test2\ (Suc\ x))$
shows $y \in test2\ x$
⟨*proof*⟩

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* ⟷ ($\forall c{>}0.\ f \in o(\lambda x.\ inverse\ (x\ powr\ c))$)

lemma *negligibleI* [*intro?*]:
($\bigwedge c.\ c > 0 \Longrightarrow f \in o(\lambda x.\ inverse\ (x\ powr\ c))) \Longrightarrow negligible\ f$
⟨*proof*⟩

lemma *negligibleD*:
$[\![$ *negligible f*; $c > 0$ $]\!] \Longrightarrow f \in o(\lambda x.\ inverse\ (x\ powr\ c))$
⟨*proof*⟩

lemma *negligibleD-real*:
assumes *negligible f*
shows $f \in o(\lambda x.\ inverse\ (x\ powr\ c))$
⟨*proof*⟩

lemma *negligible-mono*: $[\![$ *negligible g*; $f \in O(g)$ $]\!] \Longrightarrow$ *negligible f*
⟨*proof*⟩

lemma *negligible-le*: ⟦ *negligible g*; $\bigwedge\eta$. $|f\ \eta| \leq g\ \eta$ ⟧ $\Longrightarrow$ *negligible f*
⟨*proof*⟩

lemma *negligible-K0* [*negligible-intros*, *simp*, *intro*!]: *negligible* (λ-. *0*)
⟨*proof*⟩

lemma *negligible-0* [*negligible-intros*, *simp*, *intro*!]: *negligible 0*
⟨*proof*⟩

lemma *negligible-const-iff* [*simp*]: *negligible* (λ-. *c* :: *real*) ⟷ *c* = *0*
⟨*proof*⟩

lemma *not-negligible-1*: ¬ *negligible* (λ-. *1* :: *real*)
⟨*proof*⟩

lemma *negligible-plus* [*negligible-intros*]:
⟦ *negligible f*; *negligible g* ⟧ $\Longrightarrow$ *negligible* ($\lambda\eta.\ f\ \eta + g\ \eta$)
⟨*proof*⟩

lemma *negligible-uminus* [*simp*]: *negligible* ($\lambda\eta.\ -\ f\ \eta$) ⟷ *negligible f*
⟨*proof*⟩

lemma *negligible-uminusI* [*negligible-intros*]: *negligible f* $\Longrightarrow$ *negligible* ($\lambda\eta.\ -\ f\ \eta$)
⟨*proof*⟩

lemma *negligible-minus* [*negligible-intros*]:
⟦ *negligible f*; *negligible g* ⟧ $\Longrightarrow$ *negligible* ($\lambda\eta.\ f\ \eta - g\ \eta$)
⟨*proof*⟩

lemma *negligible-cmult*: *negligible* ($\lambda\eta.\ c * f\ \eta$) ⟷ *negligible f* ∨ *c* = *0*
⟨*proof*⟩

lemma *negligible-cmultI* [*negligible-intros*]:
($c \neq 0 \Longrightarrow$ *negligible f*) $\Longrightarrow$ *negligible* ($\lambda\eta.\ c * f\ \eta$)
⟨*proof*⟩

lemma *negligible-multc*: *negligible* ($\lambda\eta.\ f\ \eta * c$) ⟷ *negligible f* ∨ *c* = *0*
⟨*proof*⟩

lemma *negligible-multcI* [*negligible-intros*]:
($c \neq 0 \Longrightarrow$ *negligible f*) $\Longrightarrow$ *negligible* ($\lambda\eta.\ f\ \eta * c$)
⟨*proof*⟩

lemma *negligible-times* [*negligible-intros*]:
assumes *f*: *negligible f*
and *g*: *negligible g*
shows *negligible* ($\lambda\eta.\ f\ \eta * g\ \eta$ :: *real*)
⟨*proof*⟩

lemma *negligible-power* [*negligible-intros*]:
  assumes *negligible f*
  and $n > 0$
  shows *negligible* ($\lambda\eta$. $f\ \eta$ ^ n :: *real*)
⟨*proof*⟩

lemma *negligible-powr* [*negligible-intros*]:
  assumes *f*: *negligible f*
  and *p*: $p > 0$
  shows *negligible* (λx. $|f\ x|$ *powr p* :: *real*)
⟨*proof*⟩

lemma *negligible-abs* [*simp*]: *negligible* (λx. $|f\ x|$) $\longleftrightarrow$ *negligible f*
⟨*proof*⟩

lemma *negligible-absI* [*negligible-intros*]: *negligible f* $\Longrightarrow$ *negligible* (λx. $|f\ x|$)
⟨*proof*⟩

lemma *negligible-powrI* [*negligible-intros*]:
  assumes $0 \leq k$ $k < 1$
  shows *negligible* (λx. *k powr x*)
⟨*proof*⟩

lemma *negligible-powerI* [*negligible-intros*]:
  fixes *k* :: *real*
  assumes $|k| < 1$
  shows *negligible* (λn. k ^ n)
⟨*proof*⟩

lemma *negligible-inverse-powerI* [*negligible-intros*]: $|k| > 1 \Longrightarrow$ *negligible* ($\lambda\eta$. 1 / k ^ η)
⟨*proof*⟩

inductive *polynomial* :: (*nat* $\Rightarrow$ *real*) $\Rightarrow$ *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* (λx. $f\ x * g\ x$)
⟨*proof*⟩

lemma *negligible-poly-times*:
  ⟦ $f \in O(\lambda x.\ x$ *powr* $n)$; *negligible g* ⟧ $\Longrightarrow$ *negligible* (λx. $f\ x * g\ x$)
⟨*proof*⟩

lemma *negligible-times-polynomial* [*negligible-intros*]:
  ⟦ *negligible f*; *polynomial g* ⟧ $\Longrightarrow$ *negligible* (λx. $f\ x * g\ x$)

⟨*proof*⟩

lemma *negligible-polynomial-times* [*negligible-intros*]:
  ⟦ *polynomial f*; *negligible g* ⟧ ⟹ *negligible* (*λx. f x ∗ g x*)
⟨*proof*⟩

lemma *negligible-divide-poly1*:
  ⟦ *f* ∈ *O*(*λx. x powr n*); *negligible* (*λη. 1 / g η*) ⟧ ⟹ *negligible* (*λη. real* (*f η*) / *g η*)
⟨*proof*⟩

lemma *negligible-divide-polynomial1* [*negligible-intros*]:
  ⟦ *polynomial f*; *negligible* (*λη. 1 / g η*) ⟧ ⟹ *negligible* (*λη. real* (*f η*) / *g η*)
⟨*proof*⟩

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*) = (*λinp. 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*) = (*λinp. 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 ($x \gg= f$) $\longleftrightarrow$ *is-Done* $x \wedge$ (*result* $x \neq$ *None* $\longrightarrow$ *is-Done* (f (*the* (*result* x))))
⟨*proof*⟩

lemma *result-bind-resumption* [*simp*]:
is-Done ($x \gg= f$) $\Longrightarrow$ *result* ($x \gg= f$) = *result* $x \gg=$ *result* $\circ f$
⟨*proof*⟩

lemma *output-bind-resumption* [*simp*]:
$\neg$ *is-Done* ($x \gg= f$) $\Longrightarrow$ *output* ($x \gg= f$) = (*if is-Done* x *then output* (f (*the* (*result* x))) *else output* x)
⟨*proof*⟩

lemma *resume-bind-resumption* [*simp*]:
$\neg$ *is-Done* ($x \gg= f$) $\Longrightarrow$
resume ($x \gg= f$) =
(*if is-Done* x *then resume* (f (*the* (*result* x)))
else (λ*inp. resume* x *inp* $\gg= f$))
⟨*proof*⟩

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-Done* (*DONE* a)
and *is-Done-ABORT*: *is-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*
⟨*proof*⟩

lemma *resume-ABORT* [*simp*]:
resume (*Done* r) = (λ*inp. ABORT*)
⟨*proof*⟩

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

lemma *results-DONE* [*simp*]: *results* (*DONE x*) = {*x*}
⟨*proof*⟩

lemma *results-ABORT* [*simp*]: *results ABORT* = {}
⟨*proof*⟩

lemma *outputs-ABORT* [*simp*]: *outputs ABORT* = {}
⟨*proof*⟩

lemma *outputs-DONE* [*simp*]: *outputs* (*DONE x*) = {}
⟨*proof*⟩

lemma *is-Done-cases* [*cases pred*]:
  assumes *is-Done r*
  obtains (*DONE*) *x* **where** *r* = *DONE x* | (*ABORT*) *r* = *ABORT*
⟨*proof*⟩

lemma *not-is-Done-conv-Pause*: ¬ *is-Done r* ⟷ (∃ *out c*. *r* = *Pause out c*)
⟨*proof*⟩

lemma *Done-bind* [*code*]:
  Done a ≫= *f* = (*case a of None* ⇒ *Done None* | *Some a* ⇒ *f a*)
⟨*proof*⟩

lemma *DONE-bind* [*simp*]:
  DONE a ≫= *f* = *f a*
⟨*proof*⟩

lemma *bind-resumption-Pause* [*simp*, *code*]: **fixes** *cont* **shows**
  Pause out cont ≫= *f*
  = *Pause out* (λ*inp*. *cont inp* ≫= *f*)
⟨*proof*⟩

lemma *bind-DONE* [*simp*]:
  x ≫= *DONE* = *x*
⟨*proof*⟩

lemma *bind-bind-resumption*:
  fixes *r* :: (*'a*, *'in*, *'out*) *resumption*
  shows (*r* ≫= *f*) ≫= *g* = *do* { *x* ← *r*; *f x* ≫= *g* }
⟨*proof*⟩

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

lemma *ABORT-bind* [*simp*]: *ABORT* ≫= *f* = *ABORT*
⟨*proof*⟩

lemma *bind-resumption-is-Done*: *is-Done f* $\Longrightarrow$ *f* $\gg=$ *g* = (*if result f* = *None then ABORT else g* (*the* (*result f*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *results-bind-resumption*:
results (*bind-resumption x f*) = ($\bigcup$ *a*$\in$*results x*. *results* (*f a*))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *outputs-bind-resumption* [*simp*]:
outputs (*bind-resumption r f*) = *outputs r* $\cup$ ($\bigcup$ *x*$\in$*results r*. *outputs* (*f x*))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

primrec *ensure* :: *bool* $\Rightarrow$ (*unit*, *'out*, *'in*) *resumption*
where
ensure True = *DONE* ()
| *ensure False* = *ABORT*

lemma *is-Done-map-resumption* [*simp*]:
is-Done (*map-resumption f1 f2 r*) $\longleftrightarrow$ *is-Done r*
⟨*proof*⟩

lemma *result-map-resumption* [*simp*]:
is-Done r $\Longrightarrow$ *result* (*map-resumption f1 f2 r*) = *map-option f1* (*result r*)
⟨*proof*⟩

lemma *output-map-resumption* [*simp*]:
$\neg$ *is-Done r* $\Longrightarrow$ *output* (*map-resumption f1 f2 r*) = *f2* (*output r*)
⟨*proof*⟩

lemma *resume-map-resumption* [*simp*]:
$\neg$ *is-Done r*
$\Longrightarrow$ *resume* (*map-resumption f1 f2 r*) = *map-resumption f1 f2* $\circ$ *resume r*
⟨*proof*⟩

lemma *rel-resumption-is-DoneD*: *rel-resumption A B r1 r2* $\Longrightarrow$ *is-Done r1* $\longleftrightarrow$ *is-Done r2*

⟨*proof*⟩

lemma *rel-resumption-resultD1*:
$\llbracket$ *rel-resumption A B r1 r2*; *is-Done r1* $\rrbracket \Longrightarrow$ *rel-option A* (*result r1*) (*result r2*)
⟨*proof*⟩

lemma *rel-resumption-resultD2*:
$\llbracket$ *rel-resumption A B r1 r2*; *is-Done r2* $\rrbracket \Longrightarrow$ *rel-option A* (*result r1*) (*result r2*)
⟨*proof*⟩

lemma *rel-resumption-outputD1*:
$\llbracket$ *rel-resumption A B r1 r2*; ¬ *is-Done r1* $\rrbracket \Longrightarrow$ *B* (*output r1*) (*output r2*)
⟨*proof*⟩

lemma *rel-resumption-outputD2*:
$\llbracket$ *rel-resumption A B r1 r2*; ¬ *is-Done r2* $\rrbracket \Longrightarrow$ *B* (*output r1*) (*output r2*)
⟨*proof*⟩

lemma *rel-resumption-resumeD1*:
$\llbracket$ *rel-resumption A B r1 r2*; ¬ *is-Done r1* $\rrbracket$
$\Longrightarrow$ *rel-resumption A B* (*resume r1 inp*) (*resume r2 inp*)
⟨*proof*⟩

lemma *rel-resumption-resumeD2*:
$\llbracket$ *rel-resumption A B r1 r2*; ¬ *is-Done r2* $\rrbracket$
$\Longrightarrow$ *rel-resumption A B* (*resume r1 inp*) (*resume r2 inp*)
⟨*proof*⟩

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*: $\bigwedge$*r1 r2*. *X r1 r2* $\Longrightarrow$ (*is-Done r1* $\longleftrightarrow$ *is-Done r2*) ∧ (*is-Done r1* $\longrightarrow$ *is-Done r2* $\longrightarrow$ *rel-option A* (*result r1*) (*result r2*))
and *Pause*: $\bigwedge$*r1 r2*. $\llbracket$ *X r1 r2*; ¬ *is-Done r1*; ¬ *is-Done r2* $\rrbracket \Longrightarrow$ *B* (*output r1*) (*output r2*) ∧ (∀ *inp*. *X* (*resume r1 inp*) (*resume r2 inp*))
shows *rel-resumption A B r1 r2*
⟨*proof*⟩

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'* $\Longrightarrow$ *resumption-ord* (*Done a*) (*Done a'*)

| *Done-Pause*: *resumption-ord ABORT* (*Pause out c*)
| *Pause-Pause*: ((=) ===> *resumption-ord*) *c c'* $\Longrightarrow$ *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'* ⟧ $\Longrightarrow$ *is-Done r*
⟨*proof*⟩

lemma *resumption-ord-resultD*:
⟦ *resumption-ord r r'*; *is-Done r'* ⟧ $\Longrightarrow$ *flat-ord None* (*result r*) (*result r'*)
⟨*proof*⟩

lemma *resumption-ord-outputD*:
⟦ *resumption-ord r r'*; ¬ *is-Done r* ⟧ $\Longrightarrow$ *output r* = *output r'*
⟨*proof*⟩

lemma *resumption-ord-resumeD*:
⟦ *resumption-ord r r'*; ¬ *is-Done r* ⟧ $\Longrightarrow$ ((=) ===> *resumption-ord*) (*resume r*) (*resume r'*)
⟨*proof*⟩

lemma *resumption-ord-abort*:
⟦ *resumption-ord r r'*; *is-Done r*; ¬ *is-Done r'* ⟧ $\Longrightarrow$ *result r* = *None*
⟨*proof*⟩

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'* ⟧ $\Longrightarrow$ *is-Done r* ∧ *flat-ord None* (*result r*) (*result r'*)
and *Abort*: ⋀*r r'*. ⟦ *X r r'*; ¬ *is-Done r'*; *is-Done r* ⟧ $\Longrightarrow$ *result r* = *None*
and *Pause*: ⋀*r r'*. ⟦ *X r r'*; ¬ *is-Done r*; ¬ *is-Done r'* ⟧
$\Longrightarrow$ *output r* = *output r'* ∧ ((=) ===> (λ*r r'*. *X r r'* ∨ *resumption-ord r r'*)) (*resume r*) (*resume r'*)
shows *resumption-ord r r'*
⟨*proof*⟩

end

lemma *resumption-ord-ABORT* [*intro*!, *simp*]: *resumption-ord ABORT r*
⟨*proof*⟩

lemma *resumption-ord-ABORT2* [*simp*]: *resumption-ord r ABORT* ⟷ *r* = *ABORT*
⟨*proof*⟩

lemma *resumption-ord-DONE1* [*simp*]: *resumption-ord* (*DONE* *x*) *r* $\longleftrightarrow$ *r* = *DONE* *x*
⟨*proof*⟩

lemma *resumption-ord-refl*: *resumption-ord* *r* *r*
⟨*proof*⟩

lemma *resumption-ord-antisym*:
$[\![$ *resumption-ord* *r* *r*′; *resumption-ord* *r*′ *r* $]\!]$
$\Longrightarrow$ *r* = *r*′
⟨*proof*⟩

lemma *resumption-ord-trans*:
$[\![$ *resumption-ord* *r* *r*′; *resumption-ord* *r*′ *r*′′ $]\!]$
$\Longrightarrow$ *resumption-ord* *r* *r*′′
⟨*proof*⟩

primcorec *resumption-lub* :: (′*a*, ′*out*, ′*in*) *resumption set* ⇒ (′*a*, ′*out*, ′*in*) *resumption*
where
∀ *r* ∈ *R*. *is-Done* *r* $\Longrightarrow$ *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*) = (λ*inp*. *resumption-lub* ((λ*c*. *c* *inp*) ‘ *resume* ‘ (*R* ∩ {*r*. ¬ *is-Done* *r*})))

lemma *is-Done-resumption-lub* [*simp*]:
is-Done (*resumption-lub* *R*) $\longleftrightarrow$ (∀ *r* ∈ *R*. *is-Done* *r*)
⟨*proof*⟩

lemma *result-resumption-lub* [*simp*]:
∀ *r* ∈ *R*. *is-Done* *r* $\Longrightarrow$ *result* (*resumption-lub* *R*) = *flat-lub* *None* (*result* ‘ *R*)
⟨*proof*⟩

lemma *output-resumption-lub* [*simp*]:
∃ *r*∈*R*. ¬ *is-Done* *r* $\Longrightarrow$ *output* (*resumption-lub* *R*) = (*THE* *out*. *out* ∈ *output* ‘ (*R* ∩ {*r*. ¬ *is-Done* *r*}))
⟨*proof*⟩

lemma *resume-resumption-lub* [*simp*]:
∃ *r*∈*R*. ¬ *is-Done* *r*
$\Longrightarrow$ *resume* (*resumption-lub* *R*) *inp* =
resumption-lub ((λ*c*. *c* *inp*) ‘ *resume* ‘ (*R* ∩ {*r*. ¬ *is-Done* *r*}))
⟨*proof*⟩

lemma *resumption-lub-empty*: *resumption-lub* {} = *ABORT*
⟨*proof*⟩

context

fixes *R state inp R′*
defines *R′-def*: *R′* ≡ (λ*c. 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*⟩

end

lemma *resumption-partial-function-definition*:
partial-function-definitions resumption-ord resumption-lub
⟨*proof*⟩

interpretation *resumption*:
partial-function-definitions resumption-ord resumption-lub
rewrites *resumption-lub* {} = (*ABORT* :: (*′a*, *′b*, *′c*) *resumption*)
⟨*proof*⟩

⟨*ML*⟩

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*⟩

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*⟩

lemma fixes *f F*
defines *F* ≡ λ*results r*. *case r of resumption.Done x* ⇒ *set-option x* | *resumption.Pause out c* ⇒ ⋃*input*. *results* (*c input*)
shows *results-conv-fixp*: *results* ≡ *ccpo.fixp* (*fun-lub Union*) (*fun-ord* (⊆)) *F* (**is** - ≡ *?fixp*)
and *results-mono*: ⋀*x*. *monotone* (*fun-ord* (⊆)) (⊆) (λ*f*. *F f x*) (**is** *PROP ?mono*)
⟨*proof*⟩

lemma *mcont-case-resumption*:
fixes *f g*
defines *h* ≡ λ*r*. *if is-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*: ⋀*out*. *mcont* (*fun-lub resumption-lub*) (*fun-ord resumption-ord*) *lub*

ord (*λc. g out c* (*Pause out c*))
and *ccpo*: *class.ccpo lub ord* (*mk-less ord*)
and *bot*: ⋀*x. ord* (*f None*) *x*
shows *mcont resumption-lub resumption-ord lub ord* (*λr. case r of Done x* ⇒ *f x* | *Pause out c* ⇒ *g out c r*)
(**is** *mcont ?lub ?ord - - ?f*)
⟨*proof*⟩

lemma *mcont2mcont-results*[*THEN mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-results*: *mcont resumption-lub resumption-ord Union* (⊆) *results*
⟨*proof*⟩

lemma *mono2mono-results*[*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-results*: *monotone resumption-ord* (⊆) *results*
⟨*proof*⟩

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*⟩

lemma *mcont2mcont-outputs*[*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-outputs*: *mcont resumption-lub resumption-ord Union* (⊆) *outputs*
⟨*proof*⟩

lemma *mono2mono-outputs*[*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-outputs*: *monotone resumption-ord* (⊆) *outputs*
⟨*proof*⟩

lemma *pred-resumption-antimono*:
assumes *r*: *pred-resumption A C r′*
and *le*: *resumption-ord r r′*
shows *pred-resumption A C r*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *left-total-rel-resumption* [*transfer-rule*]:

⟦ *left-total R1*; *left-total R2* ⟧ ⟹ *left-total* (*rel-resumption R1 R2*)
⟨*proof*⟩

lemma *left-unique-rel-resumption* [*transfer-rule*]:
⟦ *left-unique R1*; *left-unique R2* ⟧ ⟹ *left-unique* (*rel-resumption R1 R2*)
⟨*proof*⟩

lemma *right-total-rel-resumption* [*transfer-rule*]:
⟦ *right-total R1*; *right-total R2* ⟧ ⟹ *right-total* (*rel-resumption R1 R2*)
⟨*proof*⟩

lemma *right-unique-rel-resumption* [*transfer-rule*]:
⟦ *right-unique R1*; *right-unique R2* ⟧ ⟹ *right-unique* (*rel-resumption R1 R2*)
⟨*proof*⟩

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

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

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*)
⟨*proof*⟩

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'* ⟹ *IO out c* = *IO out' c* ⟨*proof*⟩
⟨*ML*⟩

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

⟨*proof*⟩

lemma *result-map-generat* [*simp*]: *is-Pure* $x \Longrightarrow$ *result* (*map-generat f g h x*) = *f* (*result x*)
⟨*proof*⟩

lemma *output-map-generat* [*simp*]: ¬ *is-Pure* $x \Longrightarrow$ *output* (*map-generat f g h x*) = *g* (*output x*)
⟨*proof*⟩

lemma *continuation-map-generat* [*simp*]: ¬ *is-Pure* $x \Longrightarrow$ *continuation* (*map-generat f g h x*) = *h* (*continuation x*)
⟨*proof*⟩

lemma [*simp*]:
shows *map-generat-eq-Pure*:
map-generat f g h generat = *Pure x* $\longleftrightarrow$ ($\exists x'$. *generat* = *Pure* $x' \wedge x = f\ x'$)
and *Pure-eq-map-generat*:
Pure x = *map-generat f g h generat* $\longleftrightarrow$ ($\exists x'$. *generat* = *Pure* $x' \wedge x = f\ x'$)
⟨*proof*⟩

lemma [*simp*]:
shows *map-generat-eq-IO*:
map-generat f g h generat = *IO out c* $\longleftrightarrow$ ($\exists$ *out'* c'. *generat* = *IO out'* $c' \wedge$ *out* = *g out'* $\wedge c = h\ c'$)
and *IO-eq-map-generat*:
IO out c = *map-generat f g h generat* $\longleftrightarrow$ ($\exists$ *out'* c'. *generat* = *IO out'* $c' \wedge$ *out* = *g out'* $\wedge c = h\ c'$)
⟨*proof*⟩

lemma *is-PureE* [*cases pred*]:
assumes *is-Pure generat*
obtains (*Pure*) *x* **where** *generat* = *Pure x*
⟨*proof*⟩

lemma *not-is-PureE*:
assumes ¬ *is-Pure generat*
obtains (*IO*) *out c* **where** *generat* = *IO out c*
⟨*proof*⟩

lemma *rel-generatI*:
⟦ *is-Pure x* $\longleftrightarrow$ *is-Pure y*;
⟦ *is-Pure x*; *is-Pure y* ⟧ $\Longrightarrow$ *A* (*result x*) (*result y*);
⟦ ¬ *is-Pure x*; ¬ *is-Pure y* ⟧ $\Longrightarrow$ *Out* (*output x*) (*output y*) $\wedge$ *R* (*continuation x*) (*continuation y*) ⟧
$\Longrightarrow$ *rel-generat A Out R x y*
⟨*proof*⟩

lemma *rel-generatD'*:

rel-generat A Out R x y
$\Longrightarrow$ (*is-Pure x* $\longleftrightarrow$ *is-Pure y*) $\wedge$
(*is-Pure x* $\longrightarrow$ *is-Pure y* $\longrightarrow$ *A* (*result x*) (*result y*)) $\wedge$
($\neg$ *is-Pure x* $\longrightarrow$ $\neg$ *is-Pure y* $\longrightarrow$ *Out* (*output x*) (*output y*) $\wedge$ *R* (*continuation x*) (*continuation y*))
⟨*proof*⟩

lemma *rel-generatD*:
assumes *rel-generat A Out R x y*
shows *rel-generat-is-PureD*: *is-Pure x* $\longleftrightarrow$ *is-Pure y*
and *rel-generat-resultD*: *is-Pure x* $\vee$ *is-Pure y* $\Longrightarrow$ *A* (*result x*) (*result y*)
and *rel-generat-outputD*: $\neg$ *is-Pure x* $\vee$ $\neg$ *is-Pure y* $\Longrightarrow$ *Out* (*output x*) (*output y*)
and *rel-generat-continuationD*: $\neg$ *is-Pure x* $\vee$ $\neg$ *is-Pure y* $\Longrightarrow$ *R* (*continuation x*) (*continuation y*)
⟨*proof*⟩

lemma *rel-generat-mono*:
⟦ *rel-generat A B C x y*; $\bigwedge$*x y. A x y* $\Longrightarrow$ *A′ x y*; $\bigwedge$*x y. B x y* $\Longrightarrow$ *B′ x y*; $\bigwedge$*x y. C x y* $\Longrightarrow$ *C′ x y* ⟧
$\Longrightarrow$ *rel-generat A′ B′ C′ x y*
⟨*proof*⟩

lemma *rel-generat-mono′* [*mono*]:
⟦ $\bigwedge$*x y. A x y* $\longrightarrow$ *A′ x y*; $\bigwedge$*x y. B x y* $\longrightarrow$ *B′ x y*; $\bigwedge$*x y. C x y* $\longrightarrow$ *C′ x y* ⟧
$\Longrightarrow$ *rel-generat A B C x y* $\longrightarrow$ *rel-generat A′ B′ C′ x y*
⟨*proof*⟩

lemma *rel-generat-same*:
rel-generat A B C r r $\longleftrightarrow$
($\forall$ *x* $\in$ *generat-pures r. A x x*) $\wedge$
($\forall$ *out* $\in$ *generat-outs r. B out out*) $\wedge$
($\forall$ *c* $\in$*generat-conts r. C c c*)
⟨*proof*⟩

lemma *rel-generat-reflI*:
⟦ $\bigwedge$*y. y* $\in$ *generat-pures x* $\Longrightarrow$ *A y y*;
$\bigwedge$*out. out* $\in$ *generat-outs x* $\Longrightarrow$ *B out out*;
$\bigwedge$*cont. cont* $\in$ *generat-conts x* $\Longrightarrow$ *C cont cont* ⟧
$\Longrightarrow$ *rel-generat A B C x x*
⟨*proof*⟩

lemma *reflp-rel-generat* [*simp*]: *reflp* (*rel-generat A B C*) $\longleftrightarrow$ *reflp A* $\wedge$ *reflp B* $\wedge$ *reflp C*
⟨*proof*⟩

lemma *transp-rel-generatI*:
assumes *transp A transp B transp C*
shows *transp* (*rel-generat A B C*)

⟨*proof*⟩

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*⟩

lemma *rel-generat-Pure1*: *rel-generat A B C* (*Pure x*) = (*λr*. ∃ *y*. *r* = *Pure y* ∧ *A x y*)
⟨*proof*⟩

lemma *rel-generat-IO1*: *rel-generat A B C* (*IO out c*) = (*λr*. ∃ *out′ c′*. *r* = *IO out′ c′* ∧ *B out out′* ∧ *C c c′*)
⟨*proof*⟩

lemma *not-is-Pure-conv*: ¬ *is-Pure r* ⟷ (∃ *out c*. *r* = *IO out c*)
⟨*proof*⟩

lemma *finite-generat-outs* [*simp*]: *finite* (*generat-outs generat*)
⟨*proof*⟩

lemma *countable-generat-outs* [*simp*]: *countable* (*generat-outs generat*)
⟨*proof*⟩

lemma *case-map-generat*:
case-generat pure io (*map-generat a b d r*) =
case-generat (*pure* ∘ *a*) (*λout*. *io* (*b out*) ∘ *d*) *r*
⟨*proof*⟩

lemma *continuation-in-generat-conts*:
¬ *is-Pure r* ⟹ *continuation r* ∈ *generat-conts r*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dest-IO-eq-None-iff* [*simp*]: *dest-IO generat* = *None* ⟷ *is-Pure generat*
⟨*proof*⟩

lemma *dest-IO-comp-Pure* [*simp*]: *dest-IO* ∘ *Pure* = (λ-. *None*)
⟨*proof*⟩

lemma *dom-dest-IO*: *dom dest-IO* = {*x*. ¬ *is-Pure x*}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *result-generat-lub* [*simp*]:
∃ *x*∈*A*. *is-Pure x* ⟹ *result* (*generat-lub lub1 lub2 lub3 A*) = *lub1* (*result* ‘ (*A* ∩ {*f*. *is-Pure f*}))
⟨*proof*⟩

lemma *output-generat-lub*:
∀ *x*∈*A*. ¬ *is-Pure x* ⟹ *output* (*generat-lub lub1 lub2 lub3 A*) = *lub2* (*output* ‘ (*A* ∩ {*f*. ¬ *is-Pure f*}))
⟨*proof*⟩

lemma *continuation-generat-lub*:
∀ *x*∈*A*. ¬ *is-Pure x* ⟹ *continuation* (*generat-lub lub1 lub2 lub3 A*) = *lub3* (*continuation* ‘ (*A* ∩ {*f*. ¬ *is-Pure f*}))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

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

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*
⟨*ML*⟩
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*
⟨*proof*⟩

declare *gpv.set*[*simp del*]

declare *gpv.set-map*[*simp*]

lemma *rel-gpv-def'*:
rel-gpv A B gpv gpv' ⟷
(∃ *gpv''*. (∀ (*x*, *y*) ∈ *results'-gpv gpv''*. *A x y*) ∧ (∀ (*x*, *y*) ∈ *outs'-gpv gpv''*. *B x y*) ∧
map-gpv fst fst gpv'' = *gpv* ∧ *map-gpv snd snd gpv''* = *gpv'*)
⟨*proof*⟩

definition *results'-rpv* :: ('*a*, '*out*, '*in*) *rpv* ⇒ '*a set*
where *results'-rpv rpv* = *range rpv* ⋙= *results'-gpv*

definition *outs'-rpv* :: ('*a*, '*out*, '*in*) *rpv* ⇒ '*out set*
where *outs'-rpv rpv* = *range rpv* ⋙= *outs'-gpv*

abbreviation *rel-rpv*
:: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*out* ⇒ '*out'* ⇒ *bool*)
⇒ ('*in* ⇒ ('*a*, '*out*, '*in*) *gpv*) ⇒ ('*in* ⇒ ('*b*, '*out'*, '*in*) *gpv*) ⇒ *bool*
where *rel-rpv A B* ≡ *rel-fun* (=) (*rel-gpv A B*)

lemma *in-results'-rpv* [*iff*]: *x* ∈ *results'-rpv rpv* ⟷ (∃ *input*. *x* ∈ *results'-gpv* (*rpv input*))
⟨*proof*⟩

lemma *in-outs-rpv* [*iff*]: *out* ∈ *outs'-rpv rpv* ⟷ (∃ *input*. *out* ∈ *outs'-gpv* (*rpv input*))
⟨*proof*⟩

lemma *results'-GPV* [*simp*]:
results'-gpv (*GPV r*) =
(*set-spmf r* ⋙= *generat-pures*) ∪
((*set-spmf r* ⋙= *generat-conts*) ⋙= *results'-rpv*)

⟨proof⟩

lemma *outs′-GPV* [*simp*]:
  outs′-gpv (*GPV r*) =
  (*set-spmf r* ≫= *generat-outs*) ∪
  ((*set-spmf r* ≫= *generat-conts*) ≫= *outs′-rpv*)
⟨proof⟩

lemma *outs′-gpv-unfold*:
  outs′-gpv r =
  (*set-spmf* (*the-gpv r*) ≫= *generat-outs*) ∪
  ((*set-spmf* (*the-gpv r*) ≫= *generat-conts*) ≫= *outs′-rpv*)
⟨proof⟩

lemma *outs′-gpv-induct* [*consumes 1*, *case-names Out Cont*, *induct set*: *outs′-gpv*]:
  assumes *x*: *x* ∈ *outs′-gpv gpv*
  and *Out*: ⋀*generat gpv*. ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *x* ∈ *generat-outs generat* ⟧ ⟹ *P gpv*
  and *Cont*: ⋀*generat gpv c input*.
  ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *c* ∈ *generat-conts generat*; *x* ∈ *outs′-gpv* (*c input*); *P* (*c input*) ⟧ ⟹ *P gpv*
  shows *P gpv*
⟨proof⟩

lemma *outs′-gpv-cases* [*consumes 1*, *case-names Out Cont*, *cases set*: *outs′-gpv*]:
  assumes *x* ∈ *outs′-gpv gpv*
  obtains (*Out*) *generat* **where** *generat* ∈ *set-spmf* (*the-gpv gpv*) *x* ∈ *generat-outs generat*
  | (*Cont*) *generat c input* **where** *generat* ∈ *set-spmf* (*the-gpv gpv*) *c* ∈ *generat-conts generat x* ∈ *outs′-gpv* (*c input*)
⟨proof⟩

lemma *outs′-gpvI* [*intro?*]:
  shows *outs′-gpv-Out*: ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *x* ∈ *generat-outs generat* ⟧ ⟹ *x* ∈ *outs′-gpv gpv*
  and *outs′-gpv-Cont*: ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *c* ∈ *generat-conts generat*; *x* ∈ *outs′-gpv* (*c input*) ⟧ ⟹ *x* ∈ *outs′-gpv gpv*
⟨proof⟩

lemma *results′-gpv-induct* [*consumes 1*, *case-names Pure Cont*, *induct set*: *results′-gpv*]:
  assumes *x*: *x* ∈ *results′-gpv gpv*
  and *Pure*: ⋀*generat gpv*. ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *x* ∈ *generat-pures generat* ⟧ ⟹ *P gpv*
  and *Cont*: ⋀*generat gpv c input*.
  ⟦ *generat* ∈ *set-spmf* (*the-gpv gpv*); *c* ∈ *generat-conts generat*; *x* ∈ *results′-gpv* (*c input*); *P* (*c input*) ⟧ ⟹ *P gpv*
  shows *P gpv*
⟨proof⟩

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* $\in$ *set-spmf* (*the-gpv gpv*) $c \in$ *generat-conts generat* $x \in$ *results′-gpv* (*c input*)
⟨*proof*⟩

lemma *results′-gpvI* [*intro?*]:
shows *results′-gpv-Pure*: ⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $x \in$ *generat-pures generat* ⟧ $\Longrightarrow x \in$ *results′-gpv gpv*
and *results′-gpv-Cont*: ⟦ *generat* $\in$ *set-spmf* (*the-gpv gpv*); $c \in$ *generat-conts generat*; $x \in$ *results′-gpv* (*c input*) ⟧ $\Longrightarrow x \in$ *results′-gpv gpv*
⟨*proof*⟩

lemma *left-unique-rel-gpv* [*transfer-rule*]:
⟦ *left-unique A*; *left-unique B* ⟧ $\Longrightarrow$ *left-unique* (*rel-gpv A B*)
⟨*proof*⟩

lemma *right-unique-rel-gpv* [*transfer-rule*]:
⟦ *right-unique A*; *right-unique B* ⟧ $\Longrightarrow$ *right-unique* (*rel-gpv A B*)
⟨*proof*⟩

lemma *bi-unique-rel-gpv* [*transfer-rule*]:
⟦ *bi-unique A*; *bi-unique B* ⟧ $\Longrightarrow$ *bi-unique* (*rel-gpv A B*)
⟨*proof*⟩

lemma *left-total-rel-gpv* [*transfer-rule*]:
⟦ *left-total A*; *left-total B* ⟧ $\Longrightarrow$ *left-total* (*rel-gpv A B*)
⟨*proof*⟩

lemma *right-total-rel-gpv* [*transfer-rule*]:
⟦ *right-total A*; *right-total B* ⟧ $\Longrightarrow$ *right-total* (*rel-gpv A B*)
⟨*proof*⟩

lemma *bi-total-rel-gpv* [*transfer-rule*]: ⟦ *bi-total A*; *bi-total B* ⟧ $\Longrightarrow$ *bi-total* (*rel-gpv A B*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *gpv-pred-mono-strong*:
⟦ *pred-gpv P Q x*; $\bigwedge a$. ⟦ $a \in$ *results′-gpv x*; *P a* ⟧ $\Longrightarrow$ *P′ a*; $\bigwedge b$. ⟦ $b \in$ *outs′-gpv*

x; *Q b* ⟧ $\Longrightarrow$ *Q′ b* ⟧ $\Longrightarrow$ *pred-gpv P′ Q′ x*
⟨*proof*⟩

lemma *pred-gpv-top* [*simp*]:
pred-gpv (λ-. *True*) (λ-. *True*) = (λ-. *True*)
⟨*proof*⟩

lemma *pred-gpv-conj* [*simp*]:
shows *pred-gpv-conj1*: ⋀*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*: ⋀*P Q R*. *pred-gpv P* (λ*x*. *Q x* ∧ *R x*) = (λ*x*. *pred-gpv P Q x* ∧ *pred-gpv P R x*)
⟨*proof*⟩

lemma *rel-gpv-restrict-relp1I* [*intro?*]:
⟦ *rel-gpv R R′ x y*; *pred-gpv P P′ x*; *pred-gpv Q Q′ y* ⟧ $\Longrightarrow$ *rel-gpv* (*R* ↿ *P* ⊗ *Q*) (*R′* ↿ *P′* ⊗ *Q′*) *x y*
⟨*proof*⟩

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*⟩

lemma *gpv-pred-map* [*simp*]: *pred-gpv P Q* (*map-gpv f g gpv*) = *pred-gpv* (*P* ∘ *f*) (*Q* ∘ *g*) *gpv*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *map-gpv′-id*: *map-gpv′ id id id* = *id*
⟨*proof*⟩

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

functor *gpv*: *map-gpv′* ⟨*proof*⟩

lemma *map-gpv-conv-map-gpv′*: *map-gpv f g* = *map-gpv′ f g id*
⟨*proof*⟩

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′*
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-gpv-conv-rel-gpv′′*: *rel-gpv A C* = *rel-gpv′′ A C* (=)
⟨*proof*⟩

lemma *rel-gpv′′-eq* :
rel-gpv′′ (=) (=) (=) = (=)
⟨*proof*⟩

lemma *rel-gpv''-mono*:
assumes $A \leq A'$ $C \leq C'$ $R' \leq R$
shows *rel-gpv''* A C $R \leq$ *rel-gpv''* A' C' R'
⟨*proof*⟩

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

lemma *rel-gpv''-pos-distr*:
rel-gpv'' A C R *OO* *rel-gpv''* A' C' $R' \leq$ *rel-gpv''* (A *OO* A') (C *OO* C') (R *OO* R')
⟨*proof*⟩

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

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

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

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

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

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

lemma *rel-gpv''-map-gpv'* [*simp*]:
shows $\bigwedge f$ g h *gpv*. *NO-MATCH id* f ∨ *NO-MATCH id* g
⟹ *rel-gpv''* A C R (*map-gpv'* f g h *gpv*) = *rel-gpv''* ($\lambda a.$ A (f a)) ($\lambda c.$ C (g c)) R (*map-gpv'* *id id* h *gpv*)
and $\bigwedge f$ 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*⟩

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

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

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

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

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

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

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

lemma *right-total-rel-gpv′*:
⟦ *right-total A*; *right-total C*; *right-unique R*; *left-total R* ⟧ ⟹ *right-total* (*rel-gpv′′ A C R*)

⟨*proof*⟩

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

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *GPV-parametric′*:
(*rel-spmf* (*rel-generat A C* (*R* ===> *rel-gpv′′ A C R*)) ===> *rel-gpv′′ A C R*) *GPV GPV*
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

lemma *map-gpv-parametric′*: ((*A* ===> *A′*) ===> (*C* ===> *C′*) ===> *rel-gpv′′ A C R* ===> *rel-gpv′′ A′ C′ R*) *map-gpv map-gpv*
⟨*proof*⟩

end

4.4 Simple, derived operations

primcorec *Done* :: $'a \Rightarrow ('a, 'out, 'in)$ *gpv*
where *the-gpv* (*Done a*) = *return-spmf* (*Pure a*)

primcorec *Pause* :: $'out \Rightarrow ('in \Rightarrow ('a, 'out, 'in)$ *gpv*$) \Rightarrow ('a, 'out, 'in)$ *gpv*
where *the-gpv* (*Pause out c*) = *return-spmf* (*IO out c*)

primcorec *lift-spmf* :: $'a$ *spmf* $\Rightarrow ('a, 'out, 'in)$ *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* :: $('in \Rightarrow 'out \times ('a, 'out, 'in)$ *rpv*$) \Rightarrow ('a, 'out, 'in)$ *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* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

lemma *Pause-inject* [*simp*]: *Pause out c* = *Pause out' c'* $\longleftrightarrow$ *out* = *out'* $\wedge$ *c* = *c'*
⟨*proof*⟩

lemma [*simp*]:
  shows *Done-neq-Pause*: *Done x* $\neq$ *Pause out c*
  and *Pause-neq-Done*: *Pause out c* $\neq$ *Done x*
⟨*proof*⟩

lemma *outs'-gpv-Done* [*simp*]: *outs'-gpv* (*Done x*) = {}
⟨*proof*⟩

lemma *results'-gpv-Done* [*simp*]: *results'-gpv* (*Done x*) = {*x*}
⟨*proof*⟩

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

lemma *outs'-gpv-Pause* [*simp*]: *outs'-gpv* (*Pause out c*) = *insert out* ($\bigcup$*input*. *outs'-gpv* (*c input*))
⟨*proof*⟩

lemma *results'-gpv-Pause* [*simp*]: *results'-gpv* (*Pause out rpv*) = *results'-rpv rpv*
⟨*proof*⟩

lemma *pred-gpv-Pause* [*simp*]: *pred-gpv P Q* (*Pause x c*) = (*Q x* $\wedge$ *All* (*pred-gpv P Q* $\circ$ *c*))
⟨*proof*⟩

lemma *lift-spmf-return* [*simp*]: *lift-spmf* (*return-spmf x*) = *Done x*
⟨*proof*⟩

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

lemma *the-gpv-lift-spmf* [*simp*]: *the-gpv* (*lift-spmf r*) = *map-spmf Pure r*
⟨*proof*⟩

lemma *outs'-gpv-lift-spmf* [*simp*]: *outs'-gpv* (*lift-spmf p*) = {}
⟨*proof*⟩

lemma *results'-gpv-lift-spmf* [*simp*]: *results'-gpv* (*lift-spmf p*) = *set-spmf p*
⟨*proof*⟩

lemma *pred-gpv-lift-spmf* [*simp*]: *pred-gpv P Q* (*lift-spmf p*) = *pred-spmf P p*
⟨*proof*⟩

lemma *lift-spmf-inject* [*simp*]: *lift-spmf p* = *lift-spmf q* ⟷ *p* = *q*
⟨*proof*⟩

lemma *map-lift-spmf*: *map-gpv f g* (*lift-spmf p*) = *lift-spmf* (*map-spmf f p*)
⟨*proof*⟩

lemma *lift-map-spmf*: *lift-spmf* (*map-spmf f p*) = *map-gpv f id* (*lift-spmf p*)
⟨*proof*⟩

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*
⟨*proof*⟩

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'* ()
⟨*proof*⟩

lemma *Fail'-code* [*code*]:
  Fail' x = *GPV* (*return-pmf None*)
⟨*proof*⟩

lemma *Fail-sel* [*simp*]:
  the-gpv Fail = *return-pmf None*
⟨*proof*⟩

lemma *Fail-eq-GPV-iff* [*simp*]: *Fail* = *GPV f* ⟷ *f* = *return-pmf None*
⟨*proof*⟩

lemma *outs′-gpv-Fail* [*simp*]: *outs′-gpv Fail* = {}
⟨*proof*⟩

lemma *results′-gpv-Fail* [*simp*]: *results′-gpv Fail* = {}
⟨*proof*⟩

lemma *pred-gpv-Fail* [*simp*]: *pred-gpv P Q Fail*
⟨*proof*⟩

lemma *React-inject* [*iff*]: *React f* = *React f′* ⟷ *f* = *f′*
⟨*proof*⟩

lemma *React-apply* [*simp*]: *f input* = (*out*, *c*) ⟹ *React f input* = *Pause out c*
⟨*proof*⟩

lemma *rFail-apply* [*simp*]: *rFail input* = *Fail*
⟨*proof*⟩

lemma [*simp*]:
shows *rFail-neq-React*: *rFail* ≠ *React f*
and *React-neq-rFail*: *React f* ≠ *rFail*
⟨*proof*⟩

lemma *rel-gpv-FailI* [*simp*]: *rel-gpv A C Fail Fail*
⟨*proof*⟩

lemma *rel-gpv-Done* [*iff*]: *rel-gpv A C* (*Done x*) (*Done y*) ⟷ *A x y*
⟨*proof*⟩

lemma *rel-gpv′′-Done* [*iff*]: *rel-gpv′′ A C R* (*Done x*) (*Done y*) ⟷ *A x y*
⟨*proof*⟩

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*))
⟨*proof*⟩

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′*))
⟨*proof*⟩

lemma *rel-gpv-lift-spmf* [*iff*]: *rel-gpv A C* (*lift-spmf p*) (*lift-spmf q*) ⟷ *rel-spmf A p q*
⟨*proof*⟩

lemma *rel-gpv″-lift-spmf* [*iff*]:
rel-gpv″ A C R (*lift-spmf p*) (*lift-spmf q*) ⟷ *rel-spmf A p q*
⟨*proof*⟩

context includes *lifting-syntax* **begin**
lemmas *Fail-parametric* [*transfer-rule*] = *rel-gpv-FailI*

lemma *Fail-parametric′* [*simp*]: *rel-gpv″ A C R Fail Fail*
⟨*proof*⟩

lemma *Done-parametric* [*transfer-rule*]: (*A* ===> *rel-gpv A C*) *Done Done*
⟨*proof*⟩

lemma *Done-parametric′*: (*A* ===> *rel-gpv″ A C R*) *Done Done*
⟨*proof*⟩

lemma *Pause-parametric* [*transfer-rule*]:
(*C* ===> ((=) ===> *rel-gpv A C*) ===> *rel-gpv A C*) *Pause Pause*
⟨*proof*⟩

lemma *Pause-parametric′*:
(*C* ===> (*R* ===> *rel-gpv″ A C R*) ===> *rel-gpv″ A C R*) *Pause Pause*
⟨*proof*⟩

lemma *lift-spmf-parametric* [*transfer-rule*]:
(*rel-spmf A* ===> *rel-gpv A C*) *lift-spmf lift-spmf*
⟨*proof*⟩

lemma *lift-spmf-parametric′*:
(*rel-spmf A* ===> *rel-gpv″ A C R*) *lift-spmf lift-spmf*
⟨*proof*⟩
end

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

lemma *map-gpv′-Done* [*simp*]: *map-gpv′ f g h* (*Done x*) = *Done* (*f x*)
⟨*proof*⟩

lemma *map-gpv-Pause* [*simp*]: *map-gpv f g* (*Pause x c*) = *Pause* (*g x*) (*map-gpv f g* ∘ *c*)
⟨*proof*⟩

lemma *map-gpv′-Pause* [*simp*]: *map-gpv′ f g h* (*Pause x c*) = *Pause* (*g x*) (*map-gpv′ f g h* ∘ *c* ∘ *h*)
⟨*proof*⟩

lemma *map-gpv-Fail* [*simp*]: *map-gpv f g Fail* = *Fail*

⟨*proof*⟩

lemma *map-gpv′-Fail* [*simp*]: *map-gpv′ f g h Fail* = *Fail*
⟨*proof*⟩

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* (λ*r*. *bind-gpv r f*))))
    (*the-gpv r* ≫=
     (*case-generat*
       (λ*x*. *map-spmf* (*map-generat id id* ((∘) *Inl*)) (*the-gpv* (*f x*)))
       (λ*out c*. *return-spmf* (*IO out* (λ*input*. *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* (λ*input*. *c input* ≫= *f*))
   })
⟨*proof*⟩

lemma *bind-gpv-code-cong*: *f* = *f′* ⟹ *bind-gpv f g* = *bind-gpv f′ g* ⟨*proof*⟩
⟨*ML*⟩

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*))
   }
⟨*proof*⟩

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*))

}
⟨*proof*⟩

lemma *Done-bind-gpv* [*simp*]: *Done a* ≫= *f* = *f a*
⟨*proof*⟩

lemma *bind-gpv-Done* [*simp*]: *f* ≫= *Done* = *f*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lift-spmf-bind*: *lift-spmf r* ≫= *f* = *GPV* (*r* ≫= *the-gpv* ∘ *f*)
⟨*proof*⟩

lemma *the-gpv-bind-gpv-lift-spmf* [*simp*]:
the-gpv (*bind-gpv* (*lift-spmf p*) *f*) = *bind-spmf p* (*the-gpv* ∘ *f*)
⟨*proof*⟩

lemma *lift-spmf-bind-spmf*: *lift-spmf* (*p* ≫= *f*) = *lift-spmf p* ≫= (λ*x*. *lift-spmf* (*f x*))
⟨*proof*⟩

lemma *lift-bind-spmf*: *lift-spmf* (*bind-spmf p f*) = *bind-gpv* (*lift-spmf p*) (*lift-spmf* ∘ *f*)
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*))))
⟨*proof*⟩

lemma *bind-gpv-assoc*:
fixes *f* :: (′*a*, ′*out*, ′*in*) *gpv*
shows (*f* ≫= *g*) ≫= *h* = *f* ≫= (λ*x*. *g x* ≫= *h*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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

lemma *map-gpv-conv-bind*:
map-gpv f (λ*x*. *x*) *x* = *bind-gpv x* (λ*x*. *Done* (*f x*))
⟨*proof*⟩

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

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*⟩

lemma *bind-gpv-Fail* [*simp*]: *Fail* ≫= *f* = *Fail*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *bind-gpv-parametric′*:
(*rel-gpv′′ A C R* ===> (*A* ===> *rel-gpv′′ B C R*) ===> *rel-gpv′′ B C R*) *bind-gpv bind-gpv*
⟨*proof*⟩

end

lemma *monad-gpv* [*locale-witness*]: *monad Done bind-gpv*
⟨*proof*⟩

lemma *monad-fail-gpv* [*locale-witness*]: *monad-fail Done bind-gpv Fail*
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *bind-gpv-cong*:
⟦ *gpv* = *gpv'*; ⋀*x*. *x* ∈ *results'-gpv gpv'* ⟹ *f x* = *g x* ⟧ ⟹ *bind-gpv gpv f* = *bind-gpv gpv' g*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *bind-rpv-code-cong*: *rpv* = *rpv'* ⟹ *bind-rpv rpv f* = *bind-rpv rpv' f* ⟨*proof*⟩
⟨*ML*⟩

lemma *bind-rpv-rDone* [*simp*]: *bind-rpv rpv Done* = *rpv*
⟨*proof*⟩

lemma *bind-gpv-Pause* [*simp*]: *bind-gpv* (*Pause out rpv*) *f* = *Pause out* (*bind-rpv rpv f*)
⟨*proof*⟩

lemma *bind-rpv-React* [*simp*]: *bind-rpv* (*React f*) *g* = *React* (*apsnd* (λ*rpv*. *bind-rpv rpv g*) ∘ *f*)
⟨*proof*⟩

lemma *bind-rpv-assoc*: *bind-rpv* (*bind-rpv rpv f*) *g* = *bind-rpv rpv* ((λ*gpv*. *bind-gpv gpv g*) ∘ *f*)
⟨*proof*⟩

lemma *bind-rpv-Done* [*simp*]: *bind-rpv Done f* = *f*
⟨*proof*⟩

lemma *results'-rpv-Done* [*simp*]: *results'-rpv Done* = *UNIV*
⟨*proof*⟩

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'*))
⟨*proof*⟩

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'* (=) = (=)
⟨*proof*⟩

lemma *rel-gpv'-mono* [*relator-mono*]: *A* ≤ *B* ⟹ *rel-gpv'* *A* ≤ *rel-gpv'* *B*
⟨*proof*⟩

lemma *rel-gpv'-distr* [*relator-distr*]: *rel-gpv'* *A* *OO* *rel-gpv'* *B* = *rel-gpv'* (*A* *OO* *B*)
⟨*proof*⟩

lemma *left-unique-rel-gpv'* [*transfer-rule*]: *left-unique* *A* ⟹ *left-unique* (*rel-gpv'* *A*)
⟨*proof*⟩

lemma *right-unique-rel-gpv'* [*transfer-rule*]: *right-unique* *A* ⟹ *right-unique* (*rel-gpv'* *A*)
⟨*proof*⟩

lemma *bi-unique-rel-gpv'* [*transfer-rule*]: *bi-unique* *A* ⟹ *bi-unique* (*rel-gpv'* *A*)
⟨*proof*⟩

lemma *left-total-rel-gpv'* [*transfer-rule*]: *left-total* *A* ⟹ *left-total* (*rel-gpv'* *A*)
⟨*proof*⟩

lemma *right-total-rel-gpv'* [*transfer-rule*]: *right-total* *A* ⟹ *right-total* (*rel-gpv'* *A*)
⟨*proof*⟩

lemma *bi-total-rel-gpv'* [*transfer-rule*]: *bi-total* *A* ⟹ *bi-total* (*rel-gpv'* *A*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-spmf-setD2*:
⟦ *rel-spmf A p q*; *y* ∈ *set-spmf q* ⟧ ⟹ ∃ *x*∈*set-spmf p*. *A x y*
⟨*proof*⟩

lemma *rel-gpv-lift-spmf1*: *rel-gpv A B* (*lift-spmf p*) *gpv* ⟷ (∃ *q*. *gpv* = *lift-spmf q* ∧ *rel-spmf A p q*)
⟨*proof*⟩

lemma *rel-gpv-lift-spmf2*: *rel-gpv A B gpv* (*lift-spmf q*) ⟷ (∃ *p*. *gpv* = *lift-spmf p* ∧ *rel-spmf A p q*)
⟨*proof*⟩

definition *pcr-spmf-gpv* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a spmf* ⇒ (*'b*, *'out*, *'in*) *gpv* ⇒ *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*
⟨*proof*⟩

lemma *left-unique-cr-spmf-gpv*: *left-unique cr-spmf-gpv*
⟨*proof*⟩

lemma *left-unique-pcr-spmf-gpv* [*transfer-rule*]:
left-unique A ⟹ *left-unique* (*pcr-spmf-gpv A*)
⟨*proof*⟩

lemma *right-unique-cr-spmf-gpv*: *right-unique cr-spmf-gpv*
⟨*proof*⟩

lemma *right-unique-pcr-spmf-gpv* [*transfer-rule*]:
right-unique A ⟹ *right-unique* (*pcr-spmf-gpv A*)
⟨*proof*⟩

lemma *bi-unique-cr-spmf-gpv*: *bi-unique cr-spmf-gpv*
⟨*proof*⟩

lemma *bi-unique-pcr-spmf-gpv* [*transfer-rule*]: *bi-unique A* ⟹ *bi-unique* (*pcr-spmf-gpv A*)
⟨*proof*⟩

lemma *left-total-cr-spmf-gpv*: *left-total cr-spmf-gpv*
⟨*proof*⟩

lemma *left-total-pcr-spmf-gpv* [*transfer-rule*]: *left-total A* ==> *left-total* (*pcr-spmf-gpv A*)
⟨*proof*⟩

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

lemma *return-spmf-gpv-transfer′*:
((=) ===> *cr-spmf-gpv*) *return-spmf Done*
⟨*proof*⟩

lemma *return-spmf-gpv-transfer* [*transfer-rule*]:
(*A* ===> *pcr-spmf-gpv A*) *return-spmf Done*
⟨*proof*⟩

lemma *bind-spmf-gpv-transfer′*:
(*cr-spmf-gpv* ===> ((=) ===> *cr-spmf-gpv*) ===> *cr-spmf-gpv*) *bind-spmf bind-gpv*
⟨*proof*⟩

lemma *bind-spmf-gpv-transfer* [*transfer-rule*]:
(*pcr-spmf-gpv A* ===> (*A* ===> *pcr-spmf-gpv B*) ===> *pcr-spmf-gpv B*) *bind-spmf bind-gpv*
⟨*proof*⟩

lemma *lift-spmf-gpv-transfer′*:
((=) ===> *cr-spmf-gpv*) (λ*x*. *x*) *lift-spmf*
⟨*proof*⟩

lemma *lift-spmf-gpv-transfer* [*transfer-rule*]:
(*rel-spmf A* ===> *pcr-spmf-gpv A*) (λ*x*. *x*) *lift-spmf*
⟨*proof*⟩

lemma *fail-spmf-gpv-transfer′*: *cr-spmf-gpv* (*return-pmf None*) *Fail*
⟨*proof*⟩

lemma *fail-spmf-gpv-transfer* [*transfer-rule*]: *pcr-spmf-gpv A* (*return-pmf None*) *Fail*
⟨*proof*⟩

lemma *map-spmf-gpv-transfer′*:
((=) ===> *R* ===> *cr-spmf-gpv* ===> *cr-spmf-gpv*) (λ*f g*. *map-spmf f*) *map-gpv*
⟨*proof*⟩

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*
⟨*proof*⟩

end

end

4.7 Embedding *'a option* as a monad

locale *option-to-gpv* **begin**

interpretation *option-to-spmf* ⟨*proof*⟩
interpretation *spmf-to-gpv* ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bind-option-gpv-transfer* [*transfer-rule*]:
(*cr-option-gpv* ===> ((=) ===> *cr-option-gpv*) ===> *cr-option-gpv*) *Option.bind bind-gpv*
⟨*proof*⟩

lemma *fail-option-gpv-transfer* [*transfer-rule*]: *cr-option-gpv None Fail*
⟨*proof*⟩

lemma *map-option-gpv-transfer* [*transfer-rule*]:
((=) ===> *R* ===> *cr-option-gpv* ===> *cr-option-gpv*) (λ*f g*. *map-option f*) *map-gpv*
⟨*proof*⟩

end

end

locale *option-le-gpv* **begin**

interpretation *option-le-spmf* ⟨*proof*⟩
interpretation *spmf-to-gpv* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bind-option-gpv-transfer* [*transfer-rule*]:
(*cr-option-le-gpv* ===> ((=) ===> *cr-option-le-gpv*) ===> *cr-option-le-gpv*)
Option.bind bind-gpv
⟨*proof*⟩

lemma *fail-option-gpv-transfer* [*transfer-rule*]:
cr-option-le-gpv None Fail
⟨*proof*⟩

lemma *map-option-gpv-transfer* [*transfer-rule*]:
(((=) ===> (=)) ===> *cr-option-le-gpv* ===> *cr-option-le-gpv*) *map-option*
(λf. *map-gpv f id*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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')
⟨*proof*⟩

lemma *lift-resumption-DONE* [*simp*]:
lift-resumption (*DONE x*) = *Done x*
⟨*proof*⟩

lemma *lift-resumption-ABORT* [*simp*]:
lift-resumption ABORT = *Fail*
⟨*proof*⟩

lemma *lift-resumption-Pause* [*simp*, *code*]:
lift-resumption (*resumption.Pause out c*) = *Pause out* (*lift-resumption* ∘ *c*)
⟨*proof*⟩

lemma *lift-resumption-Done-Some* [*simp*]: *lift-resumption* (*resumption.Done* (*Some x*)) = *Done x*
⟨*proof*⟩

lemma *results'-gpv-lift-resumption* [*simp*]:
results'-gpv (*lift-resumption r*) = *results r* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *outs'-gpv-lift-resumption* [*simp*]:
outs'-gpv (*lift-resumption r*) = *outputs r* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *pred-gpv-lift-resumption* [*simp*]:
⋀*A*. *pred-gpv A C* (*lift-resumption r*) = *pred-resumption A C r*
⟨*proof*⟩

lemma *lift-resumption-bind*: *lift-resumption* (*r* ≫= *f*) = *lift-resumption r* ≫= *lift-resumption* ∘ *f*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *assert-gpv-inject* [*simp*]: *assert-gpv b* = *assert-gpv b'* ⟷ *b* = *b'*
⟨*proof*⟩

lemma *assert-gpv-sel* [*simp*]:
the-gpv (*assert-gpv b*) = *map-spmf Pure* (*assert-spmf b*)
⟨*proof*⟩

lemma *the-gpv-bind-assert* [*simp*]:
the-gpv (*bind-gpv* (*assert-gpv b*) *f*) =
bind-spmf (*assert-spmf b*) (*the-gpv* ∘ *f*)
⟨*proof*⟩

lemma *pred-gpv-assert* [*simp*]: *pred-gpv P Q* (*assert-gpv b*) = (*b* ⟶ *P* ())
⟨*proof*⟩

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* (λ*c 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* (λ*c input*. *TRY c input ELSE gpv'*)) (*the-gpv gpv*) *ELSE the-gpv gpv'*
⟨*proof*⟩

lemma *try-gpv-Done* [*simp*]: *TRY Done x ELSE gpv'* = *Done x*
⟨*proof*⟩

lemma *try-gpv-Fail* [*simp*]: *TRY Fail ELSE gpv'* = *gpv'*
⟨*proof*⟩

lemma *try-gpv-Pause* [*simp*]: *TRY Pause out c ELSE gpv'* = *Pause out* (λ*input*. *TRY c input ELSE gpv'*)
⟨*proof*⟩

lemma *try-gpv-Fail2* [*simp*]: *TRY gpv ELSE Fail* = *gpv*
⟨*proof*⟩

lemma *lift-try-spmf*: *lift-spmf* (*TRY p ELSE q*) = *TRY lift-spmf p ELSE lift-spmf q*
⟨*proof*⟩

lemma *try-assert-gpv*: *TRY assert-gpv b ELSE gpv'* = (*if b then Done* () *else gpv'*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *try-gpv-parametric′*:
(*rel-gpv′′ A C R* ===> *rel-gpv′′ A C R* ===> *rel-gpv′′ A C R*) *try-gpv try-gpv*
⟨*proof*⟩
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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *try-bind-assert-gpv*:
TRY (*assert-gpv b* ≫= *f*) *ELSE gpv* = (*if b then TRY* (*f* ()) *ELSE gpv else gpv*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *ord-gpv-the-gpvD*:
ord-gpv f g ⟹ *ord-spmf* (*rel-generat* (=) (=) (*rel-fun* (=) *ord-gpv*)) (*the-gpv f*) (*the-gpv g*)
⟨*proof*⟩

lemma *reflp-equality*: *reflp* (=)
⟨*proof*⟩

lemma *ord-gpv-reflI* [*simp*]: *ord-gpv f f*
⟨*proof*⟩

lemma *reflp-ord-gpv*: *reflp ord-gpv*
⟨*proof*⟩

lemma *ord-gpv-trans*:
assumes *ord-gpv f g ord-gpv g h*
shows *ord-gpv f h*
⟨*proof*⟩

lemma *ord-gpv-compp*: (*ord-gpv OO ord-gpv*) = *ord-gpv*
⟨*proof*⟩

lemma *transp-ord-gpv* [*simp*]: *transp ord-gpv*
⟨*proof*⟩

lemma *ord-gpv-antisym*:
⟦ *ord-gpv f g*; *ord-gpv g f* ⟧ ⟹ *f* = *g*
⟨*proof*⟩

lemma *RFail-least* [*simp*]: *ord-gpv Fail f*
⟨*proof*⟩

4.11 Bounds on interaction

context
fixes *consider* :: ′*out* ⇒ *bool*
notes *monotone-SUP*[*partial-function-mono*] [[*function-internals*]]
begin
⟨*ML*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *interaction-bound-IO-consider*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *consider out* ⟧
⟹ *eSuc* (*interaction-bound* (*c input*)) ≤ *interaction-bound gpv*
⟨*proof*⟩

lemma *interaction-bound-IO-ignore*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); ¬ *consider out* ⟧
⟹ *interaction-bound* (*c input*) ≤ *interaction-bound gpv*
⟨*proof*⟩

lemma *interaction-bound-Done* [*simp*]: *interaction-bound* (*Done x*) = *0*
⟨*proof*⟩

lemma *interaction-bound-Fail* [*simp*]: *interaction-bound Fail* = *0*
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *interaction-bound-lift-spmf* [*simp*]: *interaction-bound* (*lift-spmf p*) = *0*
⟨*proof*⟩

lemma *interaction-bound-assert-gpv* [*simp*]: *interaction-bound* (*assert-gpv b*) = *0*
⟨*proof*⟩

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*⟩

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*⟩

lemma *interaction-bound-bind-lift-spmf* [*simp*]:
interaction-bound (*lift-spmf p* ≫= *f*) = (*SUP x*∈*set-spmf p. interaction-bound* (*f x*))
⟨*proof*⟩

end

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *interaction-bound-parametric* [*transfer-rule*]:
((*C* ===> (=)) ===> *rel-gpv A C* ===> (=)) *interaction-bound interaction-bound*
⟨*proof*⟩
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*
⟨*proof*⟩

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
⟨*proof*⟩
end

lemma *interaction-bounded-by-mono*:
⟦ *interaction-bounded-by consider gpv n*; *n* ≤ *m* ⟧ ⟹ *interaction-bounded-by consider gpv m*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *interaction-bounded-by-0*: *interaction-bounded-by consider gpv 0* ⟷ *interaction-bound consider gpv* = *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *interaction-bounded-by-DoneI* [*interaction-bound*]:
interaction-bounded-by consider (*Done x*) *0*
⟨*proof*⟩

lemma *interaction-bounded-by-Fail* [*simp*]: *interaction-bounded-by consider Fail n*
⟨*proof*⟩

lemma *interaction-bounded-by-FailI* [*interaction-bound*]: *interaction-bounded-by consider Fail 0*

⟨*proof*⟩

lemma *interaction-bounded-by-lift-spmf* [*simp*]: *interaction-bounded-by consider* (*lift-spmf p*) *n*
⟨*proof*⟩

lemma *interaction-bounded-by-lift-spmfI* [*interaction-bound*]:
interaction-bounded-by consider (*lift-spmf p*) *0*
⟨*proof*⟩

lemma *interaction-bounded-by-assert-gpv* [*simp*]: *interaction-bounded-by consider* (*assert-gpv b*) *n*
⟨*proof*⟩

lemma *interaction-bounded-by-assert-gpvI* [*interaction-bound*]:
interaction-bounded-by consider (*assert-gpv b*) *0*
⟨*proof*⟩

lemma *interaction-bounded-by-Pause* [*simp*]:
interaction-bounded-by consider (*Pause out c*) *n* ⟷
(*if consider out then 0* < *n* ∧ (∀ *input. interaction-bounded-by consider* (*c input*) (*n* − *1*)) *else* (∀ *input. interaction-bounded-by consider* (*c input*) *n*))
⟨*proof*⟩

lemma *interaction-bounded-by-PauseI* [*interaction-bound*]:
(⋀*input. interaction-bounded-by consider* (*c input*) (*n input*))
⟹ *interaction-bounded-by consider* (*Pause out c*) (*if consider out then 1* + (*SUP input. n input*) *else* (*SUP input. n input*))
⟨*proof*⟩

lemma *interaction-bounded-by-bindI* [*interaction-bound*]:
⟦ *interaction-bounded-by consider gpv n*; ⋀*x. x* ∈ *results′-gpv gpv* ⟹ *interaction-bounded-by consider* (*f x*) (*m x*) ⟧
⟹ *interaction-bounded-by consider* (*gpv* ≫= *f*) (*n* + (*SUP x*∈*results′-gpv gpv. m x*))
⟨*proof*⟩

lemma *interaction-bounded-by-bind-PauseI* [*interaction-bound*]:
(⋀*input. interaction-bounded-by consider* (*c input* ≫= *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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *interaction-bounded-by-bind-DoneI* [*interaction-bound*]:
interaction-bounded-by consider (*f x*) *n* ⟹ *interaction-bounded-by consider* (*Done x* ≫= *f*) *n*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

typedef ('*call*, '*ret*) $\mathcal{I}$ = *UNIV* :: ('*call* ⇒ '*ret* *set*) *set* ⟨*proof*⟩

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* ≠ {}})
⟨*proof*⟩

lift-definition *outs-*$\mathcal{I}$ :: ('*call*, '*ret*) $\mathcal{I}$ ⇒ '*call* *set* **is** λ*resps*. {*out*. *resps* *out* ≠ {}} **parametric** *outs-*$\mathcal{I}$*-tparametric* ⟨*proof*⟩
lift-definition *responses-*$\mathcal{I}$ :: ('*call*, '*ret*) $\mathcal{I}$ ⇒ '*call* ⇒ '*ret* *set* **is** λ*x*. *x* **parametric** *id-transfer*[*unfolded* *id-def*] ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rel-*$\mathcal{I}$*-eq* [*relator-eq*]: *rel-*$\mathcal{I}$ (=) (=) = (=)
⟨*proof*⟩

lemma *rel-*$\mathcal{I}$*-conversep* [*simp*]: *rel-*$\mathcal{I}$ C^{-1-1} R^{-1-1} = (*rel-*$\mathcal{I}$ *C* *R*)$^{-1-1}$
⟨*proof*⟩

lemma *rel-*$\mathcal{I}$*-conversep1-eq* [*simp*]: *rel-*$\mathcal{I}$ C^{-1-1} (=) = (*rel-*$\mathcal{I}$ *C* (=))$^{-1-1}$
⟨*proof*⟩

lemma *rel-*$\mathcal{I}$*-conversep2-eq* [*simp*]: *rel-*$\mathcal{I}$ (=) R^{-1-1} = (*rel-*$\mathcal{I}$ (=) *R*)$^{-1-1}$
⟨*proof*⟩

lemma *responses-$\mathcal{I}$-empty-iff*: *responses-$\mathcal{I}$ $\mathcal{I}$ out* = {} ⟷ *out* ∉ *outs-$\mathcal{I}$ $\mathcal{I}$*
including *$\mathcal{I}$.lifting* ⟨*proof*⟩

lemma *in-outs-$\mathcal{I}$-iff-responses-$\mathcal{I}$*: *out* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟷ *responses-$\mathcal{I}$ $\mathcal{I}$ out* ≠ {}
⟨*proof*⟩

lift-definition *$\mathcal{I}$-full* :: (*'call*, *'ret*) *$\mathcal{I}$* **is** λ-. *UNIV* ⟨*proof*⟩

lemma *$\mathcal{I}$-full-sel* [*simp*]:
 shows *outs-$\mathcal{I}$-full*: *outs-$\mathcal{I}$ $\mathcal{I}$-full* = *UNIV*
 and *responses-$\mathcal{I}$-full*: *responses-$\mathcal{I}$ $\mathcal{I}$-full x* = *UNIV*
⟨*proof*⟩

context includes *lifting-syntax* **begin**
lemma *outs-$\mathcal{I}$-parametric* [*transfer-rule*]: (*rel-$\mathcal{I}$ C R* ===> *rel-set C*) *outs-$\mathcal{I}$ outs-$\mathcal{I}$*
⟨*proof*⟩

lemma *responses-$\mathcal{I}$-parametric* [*transfer-rule*]:
 (*rel-$\mathcal{I}$ C R* ===> *C* ===> *rel-set R*) *responses-$\mathcal{I}$ responses-$\mathcal{I}$*
⟨*proof*⟩

end

definition *$\mathcal{I}$-trivial* :: (*'out*, *'in*) *$\mathcal{I}$* ⇒ *bool*
where *$\mathcal{I}$-trivial $\mathcal{I}$* ⟷ *outs-$\mathcal{I}$ $\mathcal{I}$* = *UNIV*

lemma *$\mathcal{I}$-trivialI* [*intro?*]: (⋀*x*. *x* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$*) ⟹ *$\mathcal{I}$-trivial $\mathcal{I}$*
⟨*proof*⟩

lemma *$\mathcal{I}$-trivialD*: *$\mathcal{I}$-trivial $\mathcal{I}$* ⟹ *outs-$\mathcal{I}$ $\mathcal{I}$* = *UNIV*
⟨*proof*⟩

lemma *$\mathcal{I}$-trivial-$\mathcal{I}$-full* [*simp*]: *$\mathcal{I}$-trivial $\mathcal{I}$-full*
⟨*proof*⟩

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

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

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

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

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

lemma *$\mathcal{I}$-uniform-UNIV* [*simp*]: *$\mathcal{I}$-uniform UNIV UNIV* = *$\mathcal{I}$-full*
⟨*proof*⟩

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

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

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

lemma *map-$\mathcal{I}$-$\mathcal{I}$-uniform* [*simp*]:
map-$\mathcal{I}$ f g (*$\mathcal{I}$-uniform A B*) = *$\mathcal{I}$-uniform* (*f* −' *A*) (*g* ' *B*)
⟨*proof*⟩

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

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

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

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

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

functor *map-$\mathcal{I}$* ⟨*proof*⟩

lemma *$\mathcal{I}$-eqI*: ⟦ *outs-$\mathcal{I}$ $\mathcal{I}$* = *outs-$\mathcal{I}$ $\mathcal{I}'$*; ⋀*x*. *x* ∈ *outs-$\mathcal{I}$ $\mathcal{I}'$* ⟹ *responses-$\mathcal{I}$ $\mathcal{I}$ x* = *responses-$\mathcal{I}$ $\mathcal{I}'$ x* ⟧ ⟹ $\mathcal{I}$ = $\mathcal{I}'$
including *$\mathcal{I}$.lifting* ⟨*proof*⟩

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

definition *less-eq-$\mathcal{I}$* :: ('*a*, '*b*) $\mathcal{I}$ ⇒ ('*a*, '*b*) $\mathcal{I}$ ⇒ *bool*
where *le-$\mathcal{I}$-def*: *less-eq-$\mathcal{I}$ $\mathcal{I}$ $\mathcal{I}'$* ⟷ *outs-$\mathcal{I}$ $\mathcal{I}$* ⊆ *outs-$\mathcal{I}$ $\mathcal{I}'$* ∧ (∀ *x*∈*outs-$\mathcal{I}$ $\mathcal{I}$*. *responses-$\mathcal{I}$ $\mathcal{I}'$ x* ⊆ *responses-$\mathcal{I}$ $\mathcal{I}$ x*)

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

instance
⟨*proof*⟩
end

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

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

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

lemma *outs-$\mathcal{I}$-mono*: $\mathcal{I}$ ≤ $\mathcal{I}'$ ⟹ *outs-$\mathcal{I}$ $\mathcal{I}$* ⊆ *outs-$\mathcal{I}$ $\mathcal{I}'$*
⟨*proof*⟩

lemma *responses-$\mathcal{I}$-mono*: ⟦ $\mathcal{I}$ ≤ $\mathcal{I}'$; *x* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟧ ⟹ *responses-$\mathcal{I}$ $\mathcal{I}'$ x* ⊆ *responses-$\mathcal{I}$ $\mathcal{I}$ x*
⟨*proof*⟩

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

lemma *$\mathcal{I}$-uniform-mono*:
$\mathcal{I}$-uniform A B ≤ *$\mathcal{I}$-uniform C D* **if** *A* ⊆ *C D* ⊆ *B D* = {} ⟶ *B* = {}
⟨*proof*⟩

context begin
qualified inductive *resultsp-gpv* :: ('*out*, '*in*) $\mathcal{I}$ ⇒ '*a* ⇒ ('*a*, '*out*, '*in*) *gpv* ⇒ *bool*
for Γ *x*
where
Pure: *Pure x* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *resultsp-gpv* Γ *x gpv*
| *IO*:
⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-$\mathcal{I}$* Γ *out*; *resultsp-gpv* Γ *x* (*c input*) ⟧
⟹ *resultsp-gpv* Γ *x gpv*

definition *results-gpv* :: ('*out*, '*in*) $\mathcal{I}$ ⇒ ('*a*, '*out*, '*in*) *gpv* ⇒ '*a set*

where *results-gpv* Γ *gpv* ≡ {*x*. *resultsp-gpv* Γ *x gpv*}

lemma *resultsp-gpv-results-gpv-eq* [*pred-set-conv*]: *resultsp-gpv* Γ *x gpv* ⟷ *x* ∈ *results-gpv* Γ *gpv*
⟨*proof*⟩

context begin
⟨*ML*⟩

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*}
⟨*proof*⟩

lemma *results-gpv-Fail* [*iff*]: *results-gpv* Γ *Fail* = {}
⟨*proof*⟩

lemma *results-gpv-Pause* [*simp*]:
 results-gpv Γ (*Pause out c*) = (⋃ *input*∈*responses-ℐ* Γ *out*. *results-gpv* Γ (*c input*))
⟨*proof*⟩

lemma *results-gpv-lift-spmf* [*iff*]: *results-gpv* Γ (*lift-spmf p*) = *set-spmf p*
⟨*proof*⟩

lemma *results-gpv-assert-gpv* [*simp*]: *results-gpv* Γ (*assert-gpv b*) = (*if b then* {()} *else* {})
⟨*proof*⟩

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*⟩

lemma *results-gpv-ℐ-full*: *results-gpv ℐ-full* = *results′-gpv*
⟨*proof*⟩

lemma *results′-bind-gpv* [*simp*]:
 results′-gpv (*bind-gpv gpv f*) = (⋃ *x*∈*results′-gpv gpv*. *results′-gpv* (*f x*))

⟨*proof*⟩

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

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

lemma *pred-gpv-bind* [*simp*]: *pred-gpv P Q* (*bind-gpv gpv f*) = *pred-gpv* (*pred-gpv P Q* ∘ *f*) *Q gpv*
⟨*proof*⟩

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

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

lemma *bind-gpv-bind-option-assoc*:
bind-gpv (*monad.bind-option Fail x f*) *g* = *monad.bind-option Fail x* (λx. *bind-gpv* (*f x*) *g*)
⟨*proof*⟩

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

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

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

context begin
⟨*ML*⟩

lemmas *intros* [*intro?*] = *outsp-gpv.intros*[*to-set*]
and *IO* = *IO*[*to-set*]
and *Cont* = *Cont*[*to-set*]

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

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

end

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

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

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

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

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

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

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

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

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

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

lemma *outs′-gpv-bind-option* [*simp*]:

outs'-gpv (*monad.bind-option Fail x f*) = ($\bigcup$ *y*∈*set-option x*. *outs'-gpv* (*f y*))
⟨*proof*⟩

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

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 $\bigwedge$*x*. *x* ∈ *results-gpv* (*I-uniform UNIV X*) *gpv* ⟹ *P x* $\bigwedge$*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*
⟨*proof*⟩

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

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

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

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

and *input* $\in$ *responses-$\mathcal{I}$* ($\mathcal{I}$*-uniform UNIV* {*x. Domainp R x*}) *out'*
and *out* $\in$ *outs-gpv* ($\mathcal{I}$*-uniform UNIV* {*x. Domainp R x*}) (*rpv input*)
and *Domainp* (*rel-gpv'' A C R*) *gpv*
shows *Domainp* (*rel-gpv'' A C R*) (*rpv input*)
⟨*proof*⟩

lemma *Domainp-rel-gpv''-le*:
Domainp (*rel-gpv'' A C R*) $\leq$ *pred-gpv'* (*Domainp A*) (*Domainp C*) {*x. Domainp R x*}
⟨*proof*⟩

end

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

lemma *rel-gpv''-refl*: ⟦ (=) $\leq$ *A*; (=) $\leq$ *C*; *R* $\leq$ (=) ⟧ $\Longrightarrow$ (=) $\leq$ *rel-gpv'' A C R*
⟨*proof*⟩

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

begin

private abbreviation *foo* **where**
foo $\equiv$ (λ *fx fy gpvx gpvy g1 g2*.
$\forall$ *x y*. *x* $\in$ *fx* ($\mathcal{I}$*-uniform UNIV* (*Collect* (*Domainp R'*))) *gpvx* $\longrightarrow$
y $\in$ *fy* ($\mathcal{I}$*-uniform UNIV* (*Collect* (*Rangep R'*))) *gpvy* $\longrightarrow$ *g1 x y* $\longrightarrow$ *g2 x y*)

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

lemma *rel-gpv''-mono-strong*:
⟦ *rel-gpv'' A C R gpv gpv'*;
$\bigwedge$*x y*. ⟦ *x* $\in$ *results-gpv* ($\mathcal{I}$*-uniform UNIV* {*x. Domainp R' x*}) *gpv*; *y* $\in$ *results-gpv* ($\mathcal{I}$*-uniform UNIV* {*x. Rangep R' x*}) *gpv'*; *A x y* ⟧ $\Longrightarrow$ *A' x y*;
$\bigwedge$*x y*. ⟦ *x* $\in$ *outs-gpv* ($\mathcal{I}$*-uniform UNIV* {*x. Domainp R' x*}) *gpv*; *y* $\in$ *outs-gpv* ($\mathcal{I}$*-uniform UNIV* {*x. Rangep R' x*}) *gpv'*; *C x y* ⟧ $\Longrightarrow$ *C' x y*;

$R' \leq R$ ⟧
$\Longrightarrow$ *rel-gpv''* A' C' R' *gpv gpv'*
⟨*proof*⟩

end

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

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

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

4.12.2 Type judgements

coinductive *WT-gpv* :: (*'out*, *'in*) $\mathcal{I}$ ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *bool* (‹((-)/ ⊢*g* (-) √)› [*100*, *0*] *99*)
for Γ
where
($\bigwedge$*out c*. *IO out c* ∈ *set-spmf gpv* $\Longrightarrow$ *out* ∈ *outs-*$\mathcal{I}$ Γ ∧ (∀ *input*∈*responses-*$\mathcal{I}$ Γ *out*. Γ ⊢*g c input* √))
$\Longrightarrow$ Γ ⊢*g GPV gpv* √

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* ∈ *set-spmf* (*the-gpv gpv*) ⟧
$\Longrightarrow$ *out* ∈ *outs-*$\mathcal{I}$ Γ ∧ (∀ *input* ∈ *responses-*$\mathcal{I}$ Γ *out*. *X* (*c input*) ∨ Γ ⊢*g c input* √)
shows Γ ⊢*g gpv* √
⟨*proof*⟩

lemma *WT-gpv-simps*:
Γ ⊢*g GPV gpv* √ ⟷
(∀ *out c*. *IO out c* ∈ *set-spmf gpv* ⟶ *out* ∈ *outs-*$\mathcal{I}$ Γ ∧ (∀ *input*∈*responses-*$\mathcal{I}$ Γ *out*. Γ ⊢*g c input* √))

$\langle proof \rangle$

lemma *WT-gpvI*:
$(\bigwedge out\ c.\ IO\ out\ c \in set\text{-}spmf\ (the\text{-}gpv\ gpv) \Longrightarrow out \in outs\text{-}\mathcal{I}\ \Gamma \wedge (\forall\ input{\in}responses\text{-}\mathcal{I}\ \Gamma\ out.\ \Gamma \vdash g\ c\ input\ \surd))$
$\Longrightarrow \Gamma \vdash g\ gpv\ \surd$
$\langle proof \rangle$

lemma *WT-gpvD*:
assumes $\Gamma \vdash g\ gpv\ \surd$
shows *WT-gpv-OutD*: $IO\ out\ c \in set\text{-}spmf\ (the\text{-}gpv\ gpv) \Longrightarrow out \in outs\text{-}\mathcal{I}\ \Gamma$
and *WT-gpv-ContD*: $[\![\ IO\ out\ c \in set\text{-}spmf\ (the\text{-}gpv\ gpv);\ input \in responses\text{-}\mathcal{I}\ \Gamma\ out\ ]\!] \Longrightarrow \Gamma \vdash g\ c\ input\ \surd$
$\langle proof \rangle$

lemma *WT-gpv-mono*:
assumes *WT*: $\mathcal{I}1 \vdash g\ gpv\ \surd$
and *outs*: $outs\text{-}\mathcal{I}\ \mathcal{I}1 \subseteq outs\text{-}\mathcal{I}\ \mathcal{I}2$
and *responses*: $\bigwedge x.\ x \in outs\text{-}\mathcal{I}\ \mathcal{I}1 \Longrightarrow responses\text{-}\mathcal{I}\ \mathcal{I}2\ x \subseteq responses\text{-}\mathcal{I}\ \mathcal{I}1\ x$
shows $\mathcal{I}2 \vdash g\ gpv\ \surd$
$\langle proof \rangle$

lemma *WT-gpv-Done* [*iff*]: $\Gamma \vdash g\ Done\ x\ \surd$
$\langle proof \rangle$

lemma *WT-gpv-Fail* [*iff*]: $\Gamma \vdash g\ Fail\ \surd$
$\langle proof \rangle$

lemma *WT-gpv-PauseI*:
$[\![\ out \in outs\text{-}\mathcal{I}\ \Gamma;\ \bigwedge input.\ input \in responses\text{-}\mathcal{I}\ \Gamma\ out \Longrightarrow \Gamma \vdash g\ c\ input\ \surd\ ]\!]$
$\Longrightarrow \Gamma \vdash g\ Pause\ out\ c\ \surd$
$\langle proof \rangle$

lemma *WT-gpv-Pause* [*iff*]:
$\Gamma \vdash g\ Pause\ out\ c\ \surd \longleftrightarrow out \in outs\text{-}\mathcal{I}\ \Gamma \wedge (\forall\ input \in responses\text{-}\mathcal{I}\ \Gamma\ out.\ \Gamma \vdash g\ c\ input\ \surd)$
$\langle proof \rangle$

lemma *WT-gpv-bindI*:
$[\![\ \Gamma \vdash g\ gpv\ \surd;\ \bigwedge x.\ x \in results\text{-}gpv\ \Gamma\ gpv \Longrightarrow \Gamma \vdash g\ f\ x\ \surd\ ]\!]$
$\Longrightarrow \Gamma \vdash g\ gpv \gg= f\ \surd$
$\langle proof \rangle$

lemma *WT-gpv-bindD2*:
assumes *WT*: $\Gamma \vdash g\ gpv \gg= f\ \surd$
and *x*: $x \in results\text{-}gpv\ \Gamma\ gpv$
shows $\Gamma \vdash g\ f\ x\ \surd$
$\langle proof \rangle$

lemma *WT-gpv-bindD1*: $\Gamma \vdash g\ gpv \gg= f\ \surd \Longrightarrow \Gamma \vdash g\ gpv\ \surd$
⟨*proof*⟩

lemma *WT-gpv-bind* [*simp*]: $\Gamma \vdash g\ gpv \gg= f\ \surd \longleftrightarrow \Gamma \vdash g\ gpv\ \surd \wedge (\forall x \in results\text{-}gpv$
$\Gamma\ gpv.\ \Gamma \vdash g\ f\ x\ \surd)$
⟨*proof*⟩

lemma *WT-gpv-full* [*simp*, *intro*!]: $\mathcal{I}\text{-}full \vdash g\ gpv\ \surd$
⟨*proof*⟩

lemma *WT-gpv-lift-spmf* [*simp*, *intro*!]: $\mathcal{I} \vdash g\ lift\text{-}spmf\ p\ \surd$
⟨*proof*⟩

lemma *WT-gpv-coinduct-bind* [*consumes 1*, *case-names WT-gpv*, *case-conclusion WT-gpv out cont*]:
  assumes $*$: $X\ gpv$
  and *step*: $\bigwedge gpv\ out\ c.\ [\![\ X\ gpv;\ IO\ out\ c \in set\text{-}spmf\ (the\text{-}gpv\ gpv)\ ]\!]$
    $\Longrightarrow out \in outs\text{-}\mathcal{I}\ \mathcal{I} \wedge (\forall input \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out.$
          $X\ (c\ input) \vee$
          $\mathcal{I} \vdash g\ c\ input\ \surd \vee$
          $(\exists (gpv' :: ('b, 'call, 'ret)\ gpv)\ f.\ c\ input = gpv' \gg= f \wedge \mathcal{I} \vdash g\ gpv'\ \surd \wedge$
$(\forall x \in results\text{-}gpv\ \mathcal{I}\ gpv'.\ X\ (f\ x))))$
  shows $\mathcal{I} \vdash g\ gpv\ \surd$
⟨*proof*⟩

lemma *I-trivial-WT-gpvD* [*simp*]: $\mathcal{I}\text{-}trivial\ \mathcal{I} \Longrightarrow \mathcal{I} \vdash g\ gpv\ \surd$
⟨*proof*⟩

lemma *I-trivial-WT-gpvI*:
  assumes $\bigwedge gpv :: ('a, 'out, 'in)\ gpv.\ \mathcal{I} \vdash g\ gpv\ \surd$
  shows $\mathcal{I}\text{-}trivial\ \mathcal{I}$
⟨*proof*⟩

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

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

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

lemma *WT-gpv-map-gpv'*: $\mathcal{I} \vdash g\ map\text{-}gpv'\ f\ g\ h\ gpv\ \surd$ **if** $map\text{-}\mathcal{I}\ g\ h\ \mathcal{I} \vdash g\ gpv\ \surd$
  ⟨*proof*⟩

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

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

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

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

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

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

context begin
⟨*ML*⟩

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 $\mathcal{I} \vdash g\ gpv\ \surd$ **and** $gpv' \in sub\text{-}gpvs\ \mathcal{I}\ gpv$
shows $\mathcal{I} \vdash g\ gpv'\ \surd$
⟨*proof*⟩

lemma *WT-sub-gpvsI*:
$[\![\bigwedge out\ c.\ IO\ out\ c \in set\text{-}spmf\ (the\text{-}gpv\ gpv) \Longrightarrow out \in outs\text{-}\mathcal{I}\ \Gamma;$
$\bigwedge gpv'.\ gpv' \in sub\text{-}gpvs\ \Gamma\ gpv \Longrightarrow \Gamma \vdash g\ gpv'\ \surd]\!]$
$\Longrightarrow \Gamma \vdash g\ gpv\ \surd$
⟨*proof*⟩

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** $\mathcal{I}$ :: ('*out*, '*in*) $\mathcal{I}$
and *F* :: (('*a*, '*out*, '*in*) *gpv* ⇒ *bool*) ⇒ (('*a*, '*out*, '*in*) *gpv* ⇒ *bool*)
and *co'* :: *bool*
defines $F \equiv \lambda gen\text{-}lossless\text{-}gpv\ gpv.\ \exists pa.\ gpv = GPV\ pa\ \wedge$
$lossless\text{-}spmf\ pa \wedge (\forall out\ c\ input.\ IO\ out\ c \in set\text{-}spmf\ pa \longrightarrow input \in responses\text{-}\mathcal{I}$
$\mathcal{I}\ out \longrightarrow gen\text{-}lossless\text{-}gpv\ (c\ input))$
and $co' \equiv co$ — 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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *gen-lossless-gpv-True*: *co'* = *True* ⟹ *gen-lossless-gpv* ≡ *gfp F*
and *gen-lossless-gpv-False*: *co'* = *False* ⟹ *gen-lossless-gpv* ≡ *lfp F*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *gen-lossless-gpv-Done* [*iff*]: *gen-lossless-gpv* (*Done x*)
⟨*proof*⟩

lemma *gen-lossless-gpv-Fail* [*iff*]: ¬ *gen-lossless-gpv Fail*
⟨*proof*⟩

lemma *gen-lossless-gpv-Pause* [*simp*]:
gen-lossless-gpv (*Pause out c*) ⟷ (∀ *input* ∈ *responses-ℐ ℐ out*. *gen-lossless-gpv* (*c input*))
⟨*proof*⟩

lemma *gen-lossless-gpv-lift-spmf* [*iff*]: *gen-lossless-gpv* (*lift-spmf p*) ⟷ *lossless-spmf p*
⟨*proof*⟩

end

lemma *gen-lossless-gpv-assert-gpv* [*iff*]: *gen-lossless-gpv co ℐ* (*assert-gpv b*) ⟷ *b*
⟨*proof*⟩

abbreviation *lossless-gpv* :: (*'out*, *'in*) *ℐ* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *bool*
where *lossless-gpv* ≡ *gen-lossless-gpv False*

abbreviation *colossless-gpv* :: (*'out*, *'in*) *ℐ* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *bool*
where *colossless-gpv* ≡ *gen-lossless-gpv True*

lemma *lossless-gpv-induct* [*consumes 1*, *case-names lossless-gpv*, *induct pred*]:
assumes ∗: *lossless-gpv ℐ gpv*
and *step*: ⋀*p*. ⟦ *lossless-spmf p*;
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *input* ∈ *responses-ℐ ℐ out*⟧ ⟹ *lossless-gpv ℐ* (*c input*);
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *input* ∈ *responses-ℐ ℐ out*⟧ ⟹ *P* (*c input*) ⟧
⟹ *P* (*GPV p*)
shows *P gpv*
⟨*proof*⟩

lemma *colossless-gpv-coinduct*
[*consumes 1*, *case-names colossless-gpv*, *case-conclusion colossless-gpv lossless-spmf continuation*, *coinduct pred*]:
assumes ∗: *X gpv*
and *step*: ⋀*gpv*. *X gpv* ⟹ *lossless-spmf* (*the-gpv gpv*) ∧ (∀ *out c input*.
IO out c ∈ *set-spmf* (*the-gpv gpv*) ⟶ *input* ∈ *responses-ℐ ℐ out* ⟶ *X* (*c input*) ∨ *colossless-gpv ℐ* (*c input*))
shows *colossless-gpv ℐ gpv*
⟨*proof*⟩

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*⟩

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* $\mathcal{I}$ (*gpv* $\gg=$ *f*)
shows *gen-lossless-gpv co* $\mathcal{I}$ *gpv*
⟨*proof*⟩

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* $\gg=$ *f*)
and $x \in$ *results-gpv* $\mathcal{I}$ *gpv*
shows *gen-lossless-gpv co* $\mathcal{I}$ (*f x*)
⟨*proof*⟩

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* $\gg=$ *f*) $\longleftrightarrow$ *gen-lossless-gpv co* $\mathcal{I}$ *gpv* $\wedge$ ($\forall$ *x*$\in$*results-gpv* $\mathcal{I}$ *gpv*. *gen-lossless-gpv co* $\mathcal{I}$ (*f x*))
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *rel-gpv''-lossless-gpvD2*:

⟦ *rel-gpv'' A C R gpv gpv'*; *lossless-gpv* $\mathcal{I}'$ *gpv'*; *rel-*$\mathcal{I}$ *C R* $\mathcal{I}$ $\mathcal{I}'$ ⟧
⟹ *lossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *rel-gpv-lossless-gpvD1*:
⟦ *rel-gpv A C gpv gpv'*; *lossless-gpv* $\mathcal{I}$ *gpv*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}'$ ⟧ ⟹ *lossless-gpv* $\mathcal{I}'$ *gpv'*
⟨*proof*⟩

lemma *rel-gpv-lossless-gpvD2*:
⟦ *rel-gpv A C gpv gpv'*; *lossless-gpv* $\mathcal{I}'$ *gpv'*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}'$ ⟧
⟹ *lossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *rel-gpv''-colossless-gpvD1*:
assumes *rel*: *rel-gpv'' A C R gpv gpv'*
and *gpv*: *colossless-gpv* $\mathcal{I}$ *gpv*
and [*transfer-rule*]: *rel-*$\mathcal{I}$ *C R* $\mathcal{I}$ $\mathcal{I}'$
shows *colossless-gpv* $\mathcal{I}'$ *gpv'*
⟨*proof*⟩

lemma *rel-gpv''-colossless-gpvD2*:
⟦ *rel-gpv'' A C R gpv gpv'*; *colossless-gpv* $\mathcal{I}'$ *gpv'*; *rel-*$\mathcal{I}$ *C R* $\mathcal{I}$ $\mathcal{I}'$ ⟧
⟹ *colossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *rel-gpv-colossless-gpvD1*:
⟦ *rel-gpv A C gpv gpv'*; *colossless-gpv* $\mathcal{I}$ *gpv*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}'$ ⟧ ⟹ *colossless-gpv* $\mathcal{I}'$ *gpv'*
⟨*proof*⟩

lemma *rel-gpv-colossless-gpvD2*:
⟦ *rel-gpv A C gpv gpv'*; *colossless-gpv* $\mathcal{I}'$ *gpv'*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}'$ ⟧
⟹ *colossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

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

lemma *gen-lossless-gpv-parametric* [*transfer-rule*]:
((=) ===> *rel-*$\mathcal{I}$ *C* (=) ===> *rel-gpv A C* ===> (=))
gen-lossless-gpv gen-lossless-gpv
⟨*proof*⟩

end

lemma *gen-lossless-gpv-map-full* [*simp*]:

gen-lossless-gpv b $\mathcal{I}$-full (*map-gpv f g gpv*) = *gen-lossless-gpv b $\mathcal{I}$-full gpv*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

lemma *results-gpv-try-gpv* [*simp*]:
results-gpv $\mathcal{I}$ (*TRY gpv ELSE gpv′*) =
results-gpv $\mathcal{I}$ gpv ∪ (*if colossless-gpv $\mathcal{I}$ gpv then* {} *else results-gpv $\mathcal{I}$ gpv′*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *results′-gpv-try-gpv* [*simp*]:
results′-gpv (*TRY gpv ELSE gpv′*) =
results′-gpv gpv ∪ (*if colossless-gpv $\mathcal{I}$-full gpv then* {} *else results′-gpv gpv′*)
⟨*proof*⟩

lemma *outs′-gpv-try-gpv* [*simp*]:
outs′-gpv (*TRY gpv ELSE gpv′*) =
outs′-gpv gpv ∪ (*if colossless-gpv $\mathcal{I}$-full gpv then* {} *else outs′-gpv gpv′*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *pred-gpv-try* [*simp*]:
pred-gpv P Q (*try-gpv gpv gpv′*) = (*pred-gpv P Q gpv* ∧ (¬ *colossless-gpv $\mathcal{I}$-full gpv* ⟶ *pred-gpv P Q gpv′*))
⟨*proof*⟩

lemma *lossless-WT-gpv-induct* [*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-$\mathcal{I}$ $\mathcal{I}$*;
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *out* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟹ *input* ∈ *responses-$\mathcal{I}$ $\mathcal{I}$ out*⟧ ⟹ *lossless-gpv $\mathcal{I}$* (*c input*);
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *out* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟹ *input* ∈ *responses-$\mathcal{I}$ $\mathcal{I}$ out*⟧ ⟹ $\mathcal{I}$ ⊢g *c input* √;
⋀*out c input*. ⟦*IO out c* ∈ *set-spmf p*; *out* ∈ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟹ *input* ∈ *responses-$\mathcal{I}$ $\mathcal{I}$ out*⟧ ⟹ *P* (*c input*)⟧
⟹ *P* (*GPV p*)
shows *P gpv*
⟨*proof*⟩

lemma *lossless-gpv-induct-strong* [*consumes 1*, *case-names lossless-gpv*]:
assumes *gpv*: *lossless-gpv $\mathcal{I}$ gpv*
and *step*:

$\bigwedge p.$ ⟦ *lossless-spmf p*;
$\bigwedge gpv.$ *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ *lossless-gpv* $\mathcal{I}$ *gpv*;
$\bigwedge gpv.$ *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ *P gpv* ⟧
⟹ *P* (*GPV p*)
shows *P gpv*
⟨*proof*⟩

lemma *lossless-sub-gpvsI*:
assumes *spmf*: *lossless-spmf* (*the-gpv gpv*)
and *sub*: $\bigwedge gpv'.$ *gpv'* ∈ *sub-gpvs* $\mathcal{I}$ *gpv* ⟹ *lossless-gpv* $\mathcal{I}$ *gpv'*
shows *lossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *lossless-sub-gpvsD*:
assumes *lossless-gpv* $\mathcal{I}$ *gpv gpv'* ∈ *sub-gpvs* $\mathcal{I}$ *gpv*
shows *lossless-gpv* $\mathcal{I}$ *gpv'*
⟨*proof*⟩

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*: $\bigwedge p.$ ⟦ *lossless-spmf p*;
$\bigwedge out\ c.$ *IO out c* ∈ *set-spmf p* ⟹ *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$;
$\bigwedge gpv.$ *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ *lossless-gpv* $\mathcal{I}$ *gpv*;
$\bigwedge gpv.$ *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ $\mathcal{I}$ ⊢g *gpv* √;
$\bigwedge gpv.$ *gpv* ∈ *sub-gpvs* $\mathcal{I}$ (*GPV p*) ⟹ *P gpv* ⟧
⟹ *P* (*GPV p*)
shows *P gpv*
⟨*proof*⟩

lemma *try-gpv-gen-lossless*: — TODO: generalise to arbitrary typings ?
gen-lossless-gpv b $\mathcal{I}$*-full gpv* ⟹ (*TRY gpv ELSE gpv'*) = *gpv*
⟨*proof*⟩
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* ⟹ *TRY bind-gpv gpv f ELSE gpv'* = *bind-gpv gpv* (λ*x. TRY f x ELSE gpv'*)
⟨*proof*⟩
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''* ⟹ *gpv'* = *gpv'''* ⟧
⟹ *try-gpv gpv gpv'* = *try-gpv gpv'' gpv'''*
⟨*proof*⟩

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

primcorec *mk-lossless-gpv* :: (*'a*, *'b*, *'c*) *gpv* ⇒ (*'a*, *'b*, *'c*) *gpv* **where**
the-gpv (*mk-lossless-gpv gpv*) =
map-spmf (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* √
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *catch-gpv-Fail* [*simp*]: *catch-gpv Fail* = *Done None*
⟨*proof*⟩

lemma *catch-gpv-Pause* [*simp*]: *catch-gpv* (*Pause out rpv*) = *Pause out* (λ*input. catch-gpv* (*rpv input*))
⟨*proof*⟩

lemma *catch-gpv-lift-spmf* [*simp*]: *catch-gpv* (*lift-spmf p*) = *lift-spmf* (*spmf-of-pmf p*)
⟨*proof*⟩

lemma *catch-gpv-assert* [*simp*]: *catch-gpv* (*assert-gpv b*) = *Done* (*assert-option b*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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'*))

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩
end

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

lemma *catch-gpv-map*: *catch-gpv* (*map-gpv f g gpv*) = *map-gpv* (*map-option f*) *g* (*catch-gpv gpv*)
⟨*proof*⟩

lemma *colossless-gpv-catch-gpv* [*simp*]: *colossless-gpv* $\mathcal{I}$*-full* (*catch-gpv gpv*)
⟨*proof*⟩

lemma *colosless-gpv-catch-gpv-conv-map*:
colossless-gpv $\mathcal{I}$*-full gpv* ⟹ *catch-gpv gpv* = *map-gpv Some id gpv*
⟨*proof*⟩

lemma *catch-gpv-catch-gpv* [*simp*]: *catch-gpv* (*catch-gpv gpv*) = *map-gpv Some id* (*catch-gpv gpv*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *catch-gpv-lift-resumption* [*simp*]: *catch-gpv* (*lift-resumption r*) = *lift-resumption* (*map-resumption Some id r*)
⟨*proof*⟩

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*})
⟨*proof*⟩

lemma *Some-in-results-gpv-catch-gpv* [*simp*]:
Some x ∈ *results-gpv* $\mathcal{I}$ (*catch-gpv gpv*) ⟷ *x* ∈ *results-gpv* $\mathcal{I}$ *gpv*

⟨*proof*⟩

lemma *None-in-results-gpv-catch-gpv* [*simp*]:
None ∈ *results-gpv* $\mathcal{I}$ (*catch-gpv gpv*) ⟷ ¬ *colossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *results′-gpv-catch-gpv*:
results′-gpv (*catch-gpv gpv*) = *Some* ‘ *results′-gpv gpv* ∪ (*if colossless-gpv* $\mathcal{I}$*-full gpv then* {} *else* {*None*})
⟨*proof*⟩

lemma *Some-in-results′-gpv-catch-gpv* [*simp*]:
Some x ∈ *results′-gpv* (*catch-gpv gpv*) ⟷ *x* ∈ *results′-gpv gpv*
⟨*proof*⟩

lemma *None-in-results′-gpv-catch-gpv* [*simp*]:
None ∈ *results′-gpv* (*catch-gpv gpv*) ⟷ ¬ *colossless-gpv* $\mathcal{I}$*-full gpv*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *outs′-gpv-catch-gpv* [*simp*]: *outs′-gpv* (*catch-gpv gpv*) = *outs′-gpv gpv*
⟨*proof*⟩

lemma *pred-gpv-catch-gpv* [*simp*]: *pred-gpv* (*pred-option P*) *Q* (*catch-gpv gpv*) = *pred-gpv P Q gpv*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *bind-gpv′-cong*:
⟦ *gpv* = *gpv′*; ⋀*x*. *x* ∈ *Some* ‘ *results′-gpv gpv′* ∨ (¬ *colossless-gpv* $\mathcal{I}$*-full gpv* ∧ *x*

$=$ *None*) $\Longrightarrow$ *f x* $=$ *f′ x* ⟧
  $\Longrightarrow$ *bind-gpv′ gpv f* $=$ *bind-gpv′ gpv′ f′*
⟨*proof*⟩

lemma *bind-gpv′-cong2*:
  ⟦ *gpv* $=$ *gpv′*; $\bigwedge$*x*. *x* $\in$ *results′-gpv gpv′* $\Longrightarrow$ *f* (*Some x*) $=$ *f′* (*Some x*); $\neg$ *colossless-gpv* $\mathcal{I}$*-full gpv* $\Longrightarrow$ *f None* $=$ *f′ None* ⟧
  $\Longrightarrow$ *bind-gpv′ gpv f* $=$ *bind-gpv′ gpv′ f′*
⟨*proof*⟩

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*: $\bigwedge$*gpv gpv′*. *R gpv gpv′*
    $\Longrightarrow$ *rel-spmf* (*rel-generat* (=) (=) (*rel-fun* (=) (λ*gpv gpv′*. *R gpv gpv′* $\vee$ *gpv* $=$ *gpv′* $\vee$
      ($\exists$ *gpv2* :: (*′b*, *′call*, *′ret*) *gpv*. $\exists$ *gpv2′* :: (*′c*, *′call*, *′ret*) *gpv*. $\exists$ *f f′*. *gpv* $=$ *bind-gpv gpv2 f* $\wedge$ *gpv′* $=$ *bind-gpv gpv2′ f′* $\wedge$
        rel-gpv (λ*x y*. *R* (*f x*) (*f′ y*)) (=) *gpv2 gpv2′*))))
      (*the-gpv gpv*) (*the-gpv gpv′*)
  shows *gpv* $=$ *gpv′*
⟨*proof*⟩

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* $\Rightarrow$ *′call* $\Rightarrow$ (*′ret* $\times$ *′s*, *′call′*, *′ret′*) *gpv*
  notes [[*function-internals*]]
begin

partial-function (*spmf*) *inline1*
  :: (*′a*, *′call*, *′ret*) *gpv* $\Rightarrow$ *′s*
  $\Rightarrow$ (*′a* $\times$ *′s* + *′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*, *rpv′*, *rpv*))))

lemma *inline1-unfold*:
  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*, *rpv*′, *rpv*))))
⟨*proof*⟩

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 ⋀*inline1*′. *P inline1*′ ⟹ *P* (λ*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*)))))
shows *P inline1*
⟨*proof*⟩

lemma *inline1-fixp-induct-strong* [*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 ⋀*inline1*′. ⟦ ⋀*gpv s*. *ord-spmf* (=) (*inline1*′ *gpv s*) (*inline1 gpv s*); *P inline1*′ ⟧
⟹ *P* (λ*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*)))))
shows *P inline1*
⟨*proof*⟩

lemma *inline1-fixp-induct-strong2* [*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 ⋀*inline1*′.
⟦ ⋀*gpv s*. *ord-spmf* (=) (*inline1*′ *gpv s*) (*inline1 gpv s*);
⋀*gpv s*. *ord-spmf* (=) (*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*)))));
P inline1′ ⟧
⟹ *P* (λ*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*)))))
shows *P inline1*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *inline1-Fail* [*simp*]: *inline1 Fail s* = *return-pmf None*
⟨*proof*⟩

lemma *inline-Fail* [*simp*]: *inline Fail s* = *Fail*
⟨*proof*⟩

lemma *inline1-Done* [*simp*]: *inline1* (*Done x*) *s* = *return-spmf* (*Inl* (*x*, *s*))
⟨*proof*⟩

lemma *inline-Done* [*simp*]: *inline* (*Done x*) *s* = *Done* (*x*, *s*)
⟨*proof*⟩

lemma *inline1-lift-spmf* [*simp*]: *inline1* (*lift-spmf p*) *s* = *map-spmf* (*λx*. *Inl* (*x*, *s*)) *p*

⟨*proof*⟩

lemma *inline-lift-spmf* [*simp*]: *inline* (*lift-spmf p*) *s* = *lift-spmf* (*map-spmf* (λ*x*. (*x*, *s*)) *p*)
⟨*proof*⟩

lemma *inline1-Pause*:
inline1 (*Pause out c*) *s* =
the-gpv (*callee s out*) ⟫= (λ*react*. *case react of Pure* (*x*, *s*′) ⇒ *inline1* (*c x*) *s*′ | *IO out*′ *c*′ ⇒ *return-spmf* (*Inr* (*out*′, *c*′, *c*)))
⟨*proof*⟩

lemma *inline-Pause* [*simp*]:
inline (*Pause out c*) *s* = *callee s out* ⟫= (λ(*x*, *s*′). *inline* (*c x*) *s*′)
⟨*proof*⟩

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*)
(λ*res*. *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*⟩

lemma *inline-bind-gpv* [*simp*]:
inline (*bind-gpv gpv f*) *s* = *bind-gpv* (*inline gpv s*) (λ(*x*, *s*′). *inline* (*f x*) *s*′)
⟨*proof*⟩

end

lemma *set-inline1-lift-spmf1*: *set-spmf* (*inline1* (λ*s x*. *lift-spmf* (*p s x*)) *gpv s*) ⊆ *range Inl*
⟨*proof*⟩

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*′)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩
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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

end

lemma *set-inline2-lift-spmf1*: *set-spmf* (*inline2* (λ*s x*. *lift-spmf* (*p s x*)) *callee gpv s s′*) ⊆ *range Inl*
⟨*proof*⟩

lemma *in-set-inline2-lift-spmf1*: *y* ∈ *set-spmf* (*inline2* (λ*s x*. *lift-spmf* (*p s x*)) *callee gpv s s′*) ⟹ ∃ *r s s′*. *y* = *Inl* (*r*, *s*, *s′*)
⟨*proof*⟩

context
fixes *consider′* :: ′*call* ⇒ *bool*
and *consider* :: ′*call′* ⇒ *bool*
and *callee* :: ′*s* ⇒ ′*call* ⇒ (′*ret* × ′*s*, ′*call′*, ′*ret′*) *gpv*

notes [[*function-internals*]]
begin

private partial-function (*spmf*) *inline1′*
:: (*′a*, *′call*, *′ret*) *gpv* ⇒ *′s*
⇒ (*′a* × *′s* + *′call* × *′call′* × (*′ret* × *′s*, *′call′*, *′ret′*) *rpv* × (*′a*, *′call*, *′ret*) *rpv*) *spmf*
where
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*, *out′*, *rpv′*, *rpv*))))

private lemma *inline1′-fixp-induct* [*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′*. *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*, *out′*, *rpv′*, *rpv*)))))
shows *P inline1′*
⟨*proof*⟩ **lemma** *inline1-conv-inline1′*: *inline1 callee gpv s* = *map-spmf* (*map-sum id snd*) (*inline1′ gpv s*)
⟨*proof*⟩

context
fixes *q* :: *enat*
assumes *q*: ⋀*s x*. *consider′ x* ⟹ *interaction-bound consider* (*callee s x*) ≤ *q*
and *ignore*: ⋀*s x*. ¬ *consider′ x* ⟹ *interaction-bound consider* (*callee s x*) = *0*
begin

private lemma *interaction-bound-inline1′-aux*:
interaction-bound consider′ gpv ≤ *p*
⟹ *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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

context
fixes $\mathcal{I}$ :: ('*call*, '*ret*) $\mathcal{I}$
and $\mathcal{I}'$ :: ('*call'*, '*ret'*) $\mathcal{I}$
and *callee* :: '*s* ⇒ '*call* ⇒ ('*ret* × '*s*, '*call'*, '*ret'*) *gpv*
assumes *results*: ⋀*s x*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *results-gpv* $\mathcal{I}'$ (*callee s x*) ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* × *UNIV*
begin

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

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

context
assumes *WT*: ⋀*x s*. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ $\mathcal{I}'$ ⊢*g callee s x* √
begin

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

lemma *WT-gpv-inline*:
assumes $\mathcal{I}$ ⊢*g gpv* √
shows $\mathcal{I}'$ ⊢*g inline callee gpv s* √
⟨*proof*⟩

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*: $\bigwedge s\ x.\ x \in outs\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow lossless\text{-}spmf\ (the\text{-}gpv\ (callee\ s\ x))$
  shows *lossless-spmf* (*inline1 callee gpv s*)
⟨*proof*⟩

lemma *lossless-gpv-inline1*:
  assumes ∗: $Inr\ (out,\ rpv,\ rpv') \in set\text{-}spmf\ (inline1\ callee\ gpv\ s)$
  and ∗∗: $input \in responses\text{-}\mathcal{I}\ \mathcal{I}'\ out$
  and *lossless*: $\bigwedge s\ x.\ x \in outs\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow lossless\text{-}gpv\ \mathcal{I}'\ (callee\ s\ x)$
  shows $lossless\text{-}gpv\ \mathcal{I}'\ (rpv\ input)$
⟨*proof*⟩

lemma *lossless-results-inline1*:
  assumes $Inr\ (out,\ rpv,\ rpv') \in set\text{-}spmf\ (inline1\ callee\ gpv\ s)$
  and $(x,\ s') \in results\text{-}gpv\ \mathcal{I}'\ (rpv\ input)$
  and $input \in responses\text{-}\mathcal{I}\ \mathcal{I}'\ out$
  shows $lossless\text{-}gpv\ \mathcal{I}\ (rpv'\ x)$
⟨*proof*⟩

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*: $lossless\text{-}gpv\ \mathcal{I}\ gpv\ \mathcal{I} \vdash g\ gpv\ \surd$
  and *lossless*: $\bigwedge s\ x.\ x \in outs\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow lossless\text{-}gpv\ \mathcal{I}'\ (callee\ s\ x)$
  shows $lossless\text{-}gpv\ \mathcal{I}'\ (inline\ callee\ gpv\ s)$
⟨*proof*⟩

end

definition *id-oracle* :: $'s \Rightarrow 'call \Rightarrow ('ret \times 's,\ 'call,\ 'ret)\ gpv$
where $id\text{-}oracle\ s\ x = Pause\ x\ (\lambda x.\ Done\ (x,\ s))$

lemma *inline1-id-oracle*:
  inline1 id-oracle gpv s =
   $map\text{-}spmf\ (\lambda generat.\ case\ generat\ of\ Pure\ x \Rightarrow Inl\ (x,\ s)\ |\ IO\ out\ c \Rightarrow Inr\ (out,\ \lambda x.\ Done\ (x,\ s),\ c))\ (the\text{-}gpv\ gpv)$
⟨*proof*⟩

lemma *inline-id-oracle* [*simp*]: $inline\ id\text{-}oracle\ gpv\ s = map\text{-}gpv\ (\lambda x.\ (x,\ s))\ id\ gpv$
⟨*proof*⟩

locale *raw-converter-invariant* =
  fixes $\mathcal{I}$ :: (*'call*, *'ret*) $\mathcal{I}$
   and $\mathcal{I}'$ :: (*'call'*, *'ret'*) $\mathcal{I}$
   and *callee* :: $'s \Rightarrow 'call \Rightarrow ('ret \times 's,\ 'call',\ 'ret')\ gpv$

and I :: $'s \Rightarrow bool$
assumes *results-callee*: $\bigwedge s\ x.\ [\![\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow results\text{-}gpv\ \mathcal{I}'\ (callee\ s\ x) \subseteq responses\text{-}\mathcal{I}\ \mathcal{I}\ x \times \{s.\ I\ s\}$
and *WT-callee*: $\bigwedge x\ s.\ [\![\ x \in outs\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ]\!] \Longrightarrow \mathcal{I}' \vdash g\ callee\ s\ x\ \surd$
begin

context begin
private lemma *aux*:
$set\text{-}spmf\ (inline1\ callee\ gpv\ s) \subseteq \{Inr\ (out,\ callee',\ rpv')\ |\ out\ callee'\ rpv'.$
$\exists\, call{\in}outs\text{-}\mathcal{I}\ \mathcal{I}.\ \exists\, s.\ I\ s \wedge (\forall\, x \in responses\text{-}\mathcal{I}\ \mathcal{I}'\ out.\ callee'\ x \in sub\text{-}gpvs\ \mathcal{I}'\ (callee\ s\ call))\} \cup$
$\{Inl\ (x,\ s')\ |\ x\ s'.\ x \in results\text{-}gpv\ \mathcal{I}\ gpv \wedge I\ s'\}$
(**is** *?concl* (*inline1 callee*) *gpv s* **is** - $\subseteq$ *?rhs1* $\cup$ *?rhs2 gpv*)
if $\mathcal{I} \vdash g\ gpv\ \surd\ I\ s$
⟨*proof*⟩

lemma *inline1-in-sub-gpvs-callee*:
assumes $Inr\ (out,\ callee',\ rpv') \in set\text{-}spmf\ (inline1\ callee\ gpv\ s)$
and *WT*: $\mathcal{I} \vdash g\ gpv\ \surd$
and *s*: $I\ s$
shows $\exists\, call{\in}outs\text{-}\mathcal{I}\ \mathcal{I}.\ \exists\, s.\ I\ s \wedge (\forall\, x \in responses\text{-}\mathcal{I}\ \mathcal{I}'\ out.\ callee'\ x \in sub\text{-}gpvs\ \mathcal{I}'\ (callee\ s\ call))$
⟨*proof*⟩

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

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

lemma *WT-gpv-inline1*:
assumes $Inr\ (out,\ rpv,\ rpv') \in set\text{-}spmf\ (inline1\ callee\ gpv\ s)$
and $\mathcal{I} \vdash g\ gpv\ \surd$
and $I\ s$
shows $out \in outs\text{-}\mathcal{I}\ \mathcal{I}'$ (**is** *?thesis1*)
and $input \in responses\text{-}\mathcal{I}\ \mathcal{I}'\ out \Longrightarrow \mathcal{I}' \vdash g\ rpv\ input\ \surd$ (**is** *PROP ?thesis2*)
and $[\![\ input \in responses\text{-}\mathcal{I}\ \mathcal{I}'\ out;\ (x,\ s') \in results\text{-}gpv\ \mathcal{I}'\ (rpv\ input)\ ]\!] \Longrightarrow \mathcal{I} \vdash g\ rpv'\ x\ \surd \wedge I\ s'$ (**is** *PROP ?thesis3*)

⟨*proof*⟩

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

end

lemma *WT-gpv-inline'*:
assumes $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *results-gpv* $\mathcal{I}'$ (*callee s x*) $\subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ $x \times$ *UNIV*
and $\bigwedge x\ s.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow \mathcal{I}' \vdash g$ *callee s x* $\surd$
and $\mathcal{I} \vdash g$ *gpv* $\surd$
shows $\mathcal{I}' \vdash g$ *inline callee gpv s* $\surd$
⟨*proof*⟩

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

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

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

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

end

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

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 (λ*x*. *return-spmf* (*x*, *s*))
    (λ*out 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* (=))) (λ*f*. *P* (λ*c* *s*. *f* (*c*, *s*)))
 and *P* (λ- -. *return-pmf None*)
 and ⋀*exec-gpv*. *P exec-gpv* ⟹
   P (λ*c s*. *the-gpv c* ≫= *case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out c*. *callee s* *out* ≫= (λ(*x*, *y*). *exec-gpv* (*c x*) *y*)))
 shows *P exec-gpv*
⟨*proof*⟩

lemma *exec-gpv-fixp-induct-strong* [*case-names adm bottom step*]:
 assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*f*. *P* (λ*c* *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* (λ*c s*. *the-gpv c* ≫= *case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out c*. *callee* *s out* ≫= (λ(*x*, *y*). *exec-gpv'* (*c x*) *y*)))
 shows *P exec-gpv*
⟨*proof*⟩

lemma *exec-gpv-fixp-induct-strong2* [*case-names adm bottom step*]:
 assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*f*. *P* (λ*c* *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* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out c*. *callee s out* ≫= (λ(*x*, *y*). *exec-gpv'* (*c x*) *y*)));
    P exec-gpv' ⟧
  ⟹ *P* (λ*c s*. *the-gpv c* ≫= *case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out c*. *callee* *s out* ≫= (λ(*x*, *y*). *exec-gpv'* (*c x*) *y*)))
 shows *P exec-gpv*
⟨*proof*⟩

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)
⟨*proof*⟩

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))
⟨*proof*⟩

lemma *exec-gpv-lift-spmf* [*simp*]:
exec-gpv callee (lift-spmf p) s = bind-spmf p (λx. return-spmf (x, s))
⟨*proof*⟩

lemma *exec-gpv-Done* [*simp*]: *exec-gpv callee (Done x) s = return-spmf (x, s)*
⟨*proof*⟩

lemma *exec-gpv-Fail* [*simp*]: *exec-gpv callee Fail s = return-pmf None*
⟨*proof*⟩

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)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

end

lemma *exec-gpv-bind*: *exec-gpv callee* $(c \gg= f)$ *s* = *exec-gpv callee c s* $\gg=$ $(\lambda(x, s') \Rightarrow$ *exec-gpv callee* $(f\ x)$ $s')$
⟨*proof*⟩

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*⟩

lemma *exec-gpv-Pause* [*simp*]:
exec-gpv callee (*Pause out f*) *s* = *callee s out* $\gg=$ $(\lambda(x, s').$ *exec-gpv callee* $(f\ x)$ $s')$
⟨*proof*⟩

lemma *exec-gpv-bind-lift-spmf*:
exec-gpv callee (*bind-gpv* (*lift-spmf p*) *f*) *s* = *bind-spmf p* (λx. *exec-gpv callee* $(f\ x)$ *s*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *exec-gpv-inline*:
fixes *callee* :: ($'c$, $'r$, $'s$) *callee*
and *gpv* :: $'s' \Rightarrow 'c' \Rightarrow ('r' \times 's', 'c, 'r)$ *gpv*
shows *exec-gpv callee* (*inline gpv c' s'*) *s* =
map-spmf $(\lambda(x, s', s). ((x, s'), s))$ (*exec-gpv* $(\lambda(s', s)\ y.$ *map-spmf* $(\lambda((x, s'), s). (x, s', s))$ (*exec-gpv callee* (*gpv s' y*) *s*)) *c'* $(s', s))$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

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* (*λa.* (*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*
⟨*proof*⟩

lemma *execp-resumption-DONE* [*simp*]: *execp-resumption* (*DONE x*) *s* = *return-spmf* (*x*, *s*)
⟨*proof*⟩

lemma *exec-gpv-lift-resumption*: *exec-gpv callee* (*lift-resumption r*) *s* = *execp-resumption r s*
⟨*proof*⟩

lemma *mcont2mcont-execp-resumption* [*THEN spmf.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-execp-resumption*:
mcont resumption-lub resumption-ord lub-spmf (*ord-spmf* (=)) (*λr. execp-resumption r s*)
⟨*proof*⟩

lemma *execp-resumption-bind* [*simp*]:
execp-resumption (*r* ≫= *f*) *s* = *execp-resumption r s* ≫= (*λ*(*x*, *s'*). *execp-resumption* (*f x*) *s'*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ⟧

$\Longrightarrow \mathcal{I} \vdash c$ *callee* $\surd$

lemmas *WT-calleeI* = *WT-callee*
hide-fact *WT-callee*

lemma *WT-calleeD*: $[\![$ $\mathcal{I} \vdash c$ *callee* $\surd$; (*ret*, *s*) $\in$ *set-spmf* (*callee out*); *out* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}$ $]\!]$ $\Longrightarrow$ *ret* $\in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *out*
⟨*proof*⟩

lemma *WT-callee-full* [*intro*!, *simp*]: *$\mathcal{I}$-full* $\vdash c$ *callee* $\surd$
⟨*proof*⟩

lemma *WT-callee-parametric* [*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique R*
shows (*rel-$\mathcal{I}$ C R* ===> (*C* ===> *rel-spmf* (*rel-prod R S*)) ===> (=)) *WT-callee WT-callee*
⟨*proof*⟩

locale *callee-invariant-on-base* =
fixes *callee* :: *'s* $\Rightarrow$ *'a* $\Rightarrow$ (*'b* $\times$ *'s*) *spmf*
and *I* :: *'s* $\Rightarrow$ *bool*
and $\mathcal{I}$ :: (*'a*, *'b*) $\mathcal{I}$

locale *callee-invariant-on* = *callee-invariant-on-base callee I* $\mathcal{I}$
for *callee* :: *'s* $\Rightarrow$ *'a* $\Rightarrow$ (*'b* $\times$ *'s*) *spmf*
and *I* :: *'s* $\Rightarrow$ *bool*
and $\mathcal{I}$ :: (*'a*, *'b*) $\mathcal{I}$
\+
assumes *callee-invariant*: $\bigwedge$*s x y s'*. $[\![$ (*y*, *s'*) $\in$ *set-spmf* (*callee s x*); *I s*; *x* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}$ $]\!]$ $\Longrightarrow$ *I s'*
and *WT-callee*: $\bigwedge$*s*. *I s* $\Longrightarrow$ $\mathcal{I} \vdash c$ *callee s* $\surd$
begin

lemma *callee-invariant'*: $[\![$ (*y*, *s'*) $\in$ *set-spmf* (*callee s x*); *I s*; *x* $\in$ *outs-$\mathcal{I}$* $\mathcal{I}$ $]\!]$ $\Longrightarrow$ *I s'* $\wedge$ *y* $\in$ *responses-$\mathcal{I}$* $\mathcal{I}$ *x*
⟨*proof*⟩

lemma *exec-gpv-invariant'*:
$[\![$ *I s*; $\mathcal{I} \vdash g$ *gpv* $\surd$ $]\!]$ $\Longrightarrow$ *set-spmf* (*exec-gpv callee gpv s*) $\subseteq$ {(*x*, *s'*). *I s'*}
⟨*proof*⟩

lemma *exec-gpv-invariant*:
$[\![$ (*x*, *s'*) $\in$ *set-spmf* (*exec-gpv callee gpv s*); *I s*; $\mathcal{I} \vdash g$ *gpv* $\surd$ $]\!]$ $\Longrightarrow$ *I s'*
⟨*proof*⟩

lemma *interaction-bounded-by-exec-gpv-count'*:
fixes *count*
assumes *bound*: *interaction-bounded-by consider gpv n*

and *count*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; *consider x*; $x \in$ *outs-ℐ ℐ* ⟧ $\Longrightarrow$ *count* $s' \leq$ *eSuc* (*count s*)
and *ignore*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; $x \in$ *outs-ℐ ℐ* ⟧ $\Longrightarrow$ *count* $s' \leq$ *count s*
and *WT*: *ℐ* ⊢*g gpv* √
and *I*: *I s*
shows *set-spmf* (*exec-gpv callee gpv s*) $\subseteq \{(x, s').\ count\ s' \leq n + count\ s\}$
⟨*proof*⟩

lemma *interaction-bounded-by-exec-gpv-count*:
fixes *count*
assumes *bound*: *interaction-bounded-by consider gpv n*
and *xs'*: $(x, s') \in$ *set-spmf* (*exec-gpv callee gpv s*)
and *count*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; *consider x*; $x \in$ *outs-ℐ ℐ* ⟧ $\Longrightarrow$ *count* $s' \leq$ *eSuc* (*count s*)
and *ignore*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; $x \in$ *outs-ℐ ℐ* ⟧ $\Longrightarrow$ *count* $s' \leq$ *count s*
and *WT*: *ℐ* ⊢*g gpv* √
and *I*: *I s*
shows *count* $s' \leq n +$ *count s*
⟨*proof*⟩

lemma *interaction-bounded-by'-exec-gpv-count*:
fixes *count*
assumes *bound*: *interaction-bounded-by' consider gpv n*
and *xs'*: $(x, s') \in$ *set-spmf* (*exec-gpv callee gpv s*)
and *count*: $\bigwedge s\ x\ y\ s'$. ⟦ $(y, s') \in$ *set-spmf* (*callee s x*); *I s*; *consider x*; $x \in$ *outs-ℐ 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*; ¬ *consider x*; $x \in$ *outs-ℐ ℐ* ⟧ $\Longrightarrow$ *count* $s' \leq$ *count s*
and *outs*: *ℐ* ⊢*g gpv* √
and *I*: *I s*
shows *count* $s' \leq n +$ *count s*
⟨*proof*⟩

lemma *pred-spmf-calleeI*: ⟦ *I s*; $x \in$ *outs-ℐ ℐ* ⟧ $\Longrightarrow$ *pred-spmf* (*pred-prod* (λ-. *True*) *I*) (*callee s x*)
⟨*proof*⟩

lemma *lossless-exec-gpv*:
assumes *gpv*: *lossless-gpv ℐ gpv*
and *callee*: $\bigwedge s\ out$. ⟦ $out \in$ *outs-ℐ ℐ*; *I s* ⟧ $\Longrightarrow$ *lossless-spmf* (*callee s out*)
and *WT-gpv*: *ℐ* ⊢*g gpv* √
and *I*: *I s*
shows *lossless-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

lemma *in-set-spmf-exec-gpv-into-results-gpv*:
assumes ∗: $(x, s') \in$ *set-spmf* (*exec-gpv callee gpv s*)

and *WT-gpv* : $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I*: *I s*
shows $x \in$ *results-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

end

lemma *callee-invariant-on-alt-def*:
callee-invariant-on = (λ*callee I* $\mathcal{I}$.
($\forall s \in$ *Collect I*. $\forall x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$. $\forall (y, s') \in$ *set-spmf* (*callee s x*). *I s'*) $\wedge$
($\forall s \in$ *Collect I*. $\mathcal{I} \vdash c$ *callee s* $\surd$))
⟨*proof*⟩

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
⟨*proof*⟩

lemma *callee-invariant-on-cong*:
⟦ *I* = *I'*; *outs-*$\mathcal{I}$ $\mathcal{I}$ = *outs-*$\mathcal{I}$ $\mathcal{I}'$;
$\bigwedge$*s x*. ⟦ *I' s*; $x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}'$ ⟧ $\Longrightarrow$ *set-spmf* (*callee s x*) $\subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* $\times$ *Collect I'* $\longleftrightarrow$ *set-spmf* (*callee' s x*) $\subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}'$ *x* $\times$ *Collect I'* ⟧
$\Longrightarrow$ *callee-invariant-on callee I* $\mathcal{I}$ = *callee-invariant-on callee' I'* $\mathcal{I}'$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *callee-invariant-on-return-spmf* [*simp*]:
callee-invariant-on (λ*s x*. *return-spmf* (*f s x*)) *I* $\mathcal{I}$ $\longleftrightarrow$ ($\forall s$. $\forall x \in$*outs-*$\mathcal{I}$ $\mathcal{I}$. *I s* $\longrightarrow$ *I* (*snd* (*f s x*)) $\wedge$ *fst* (*f s x*) $\in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x*)
⟨*proof*⟩

lemma *callee-invariant-return-spmf* [*simp*]:
callee-invariant (λ*s x*. *return-spmf* (*f s x*)) *I* $\longleftrightarrow$ ($\forall s\ x$. *I s* $\longrightarrow$ *I* (*snd* (*f s x*)))
⟨*proof*⟩

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*⟩

lemma *callee-invariant-on-True* [*simp*]: *callee-invariant-on callee* (*λ-. True*) $\mathcal{I}$ ⟷ (∀ *s*. $\mathcal{I}$ ⊢*c callee s* √)
⟨*proof*⟩

lemma *lossless-exec-gpv*:
⟦ *lossless-gpv* $\mathcal{I}$ *gpv*; ⋀*s out. out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *lossless-spmf* (*callee s out*);
$\mathcal{I}$ ⊢*g gpv* √; ⋀*s*. $\mathcal{I}$ ⊢*c callee s* √ ⟧
⟹ *lossless-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

lemma *in-set-spmf-exec-gpv-into-results′-gpv*:
assumes ∗: (*x*, *s′*) ∈ *set-spmf* (*exec-gpv callee gpv s*)
shows *x* ∈ *results′-gpv gpv*
⟨*proof*⟩

context fixes $\mathcal{I}$ :: (*′out*, *′in*) $\mathcal{I}$ **begin**

primcorec *restrict-gpv* :: (*′a*, *′out*, *′in*) *gpv* ⇒ (*′a*, *′out*, *′in*) *gpv*
where
restrict-gpv gpv = *GPV* (
map-pmf (*case-option None* (*case-generat* (*Some* ∘ *Pure*)
(*λout c. if out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ *then Some* (*IO out* (*λinput. if input* ∈ *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*
⟨*proof*⟩

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

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

lemma *restrict-gpv-bind* [*simp*]: *restrict-gpv* (*bind-gpv gpv f*) = *bind-gpv* (*restrict-gpv gpv*) (*λx. restrict-gpv* (*f x*))
⟨*proof*⟩

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

lemma *exec-gpv-restrict-gpv*:
assumes $\mathcal{I}$ ⊢*g gpv* √ **and** *WT-callee*: ⋀*s*. $\mathcal{I}$ ⊢*c callee s* √

shows *exec-gpv callee* (*restrict-gpv gpv*) *s* = *exec-gpv callee gpv s*
⟨*proof*⟩

lemma *in-outs′-restrict-gpvD*: *x* ∈ *outs′-gpv* (*restrict-gpv gpv*) ⟹ *x* ∈ *outs-ℐ ℐ*
⟨*proof*⟩

lemma *outs′-restrict-gpv*: *outs′-gpv* (*restrict-gpv gpv*) ⊆ *outs-ℐ ℐ* ⟨*proof*⟩

lemma *lossless-restrict-gpvI*: ⟦ *lossless-gpv ℐ gpv*; *ℐ* ⊢*g gpv* √ ⟧ ⟹ *lossless-gpv ℐ* (*restrict-gpv gpv*)
⟨*proof*⟩

lemma *lossless-restrict-gpvD*: ⟦ *lossless-gpv ℐ* (*restrict-gpv gpv*); *ℐ* ⊢*g gpv* √ ⟧ ⟹ *lossless-gpv ℐ gpv*
⟨*proof*⟩

lemma *colossless-restrict-gpvD*:
⟦ *colossless-gpv ℐ* (*restrict-gpv gpv*); *ℐ* ⊢*g gpv* √ ⟧ ⟹ *colossless-gpv ℐ gpv*
⟨*proof*⟩

lemma *colossless-restrict-gpvI*:
⟦ *colossless-gpv ℐ gpv*; *ℐ* ⊢*g gpv* √ ⟧ ⟹ *colossless-gpv ℐ* (*restrict-gpv gpv*)
⟨*proof*⟩

lemma *gen-colossless-restrict-gpv* [*simp*]:
ℐ ⊢*g gpv* √ ⟹ *gen-lossless-gpv b ℐ* (*restrict-gpv gpv*) ⟷ *gen-lossless-gpv b ℐ gpv*
⟨*proof*⟩

lemma *interaction-bound-restrict-gpv*:
interaction-bound consider (*restrict-gpv gpv*) ≤ *interaction-bound consider gpv*
⟨*proof*⟩

lemma *interaction-bounded-by-restrict-gpvI* [*interaction-bound*, *simp*]:
interaction-bounded-by consider gpv n ⟹ *interaction-bounded-by consider* (*restrict-gpv gpv*) *n*
⟨*proof*⟩

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*
⟨*proof*⟩

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

bi-unique C ⟹ (*rel-ℐ C* (=) ===> *rel-gpv A C* ===> *rel-gpv A C*) *restrict-gpv restrict-gpv*
⟨*proof*⟩

lemma *map-restrict-gpv*: *map-gpv f id* (*restrict-gpv ℐ gpv*) = *restrict-gpv ℐ* (*map-gpv f id gpv*)
for *gpv* :: (*'a*, *'out*, *'ret*) *gpv*
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *exec-gpv-restrict-gpv-invariant*:
assumes *ℐ* ⊢*g gpv* √ **and** *I s*
shows *exec-gpv callee* (*restrict-gpv ℐ gpv*) *s* = *exec-gpv callee gpv s*
⟨*proof*⟩

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

lemma *results-gpv-restrict-gpv*:
results-gpv ℐ (*restrict-gpv ℐ' gpv*) ⊆ *results-gpv ℐ gpv*
⟨*proof*⟩

lemma *in-results'-gpv-restrict-gpvD*:
x ∈ *results'-gpv* (*restrict-gpv ℐ' gpv*) ⟹ *x* ∈ *results'-gpv gpv*
⟨*proof*⟩

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

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

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

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

lemma *enforce-ℐ-gpv-lift-spmf* [*simp*]: *enforce-ℐ-gpv ℐ* (*lift-spmf p*) = *lift-spmf p*
⟨*proof*⟩

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

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

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

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

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-$\mathcal{I}$* $\mathcal{I}$ *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-$\mathcal{I}$* $\mathcal{I}$ *out* ⟧ ⟹ *finite-gpv* (*c input*)
⟨*proof*⟩

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

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

lemma *finite-gpv-Pause* [*simp*]: *finite-gpv* (*Pause x c*) ⟷ (∀ *input* ∈ *responses-$\mathcal{I}$* $\mathcal{I}$ *x*. *finite-gpv* (*c input*))
⟨*proof*⟩

lemma *finite-gpv-lift-spmf* [*simp*]: *finite-gpv* (*lift-spmf p*)
⟨*proof*⟩

lemma *finite-gpv-bind* [*simp*]:
finite-gpv (*gpv* $\gg=$ *f*) $\longleftrightarrow$ *finite-gpv gpv* $\wedge$ ($\forall$ *x*$\in$*results-gpv* $\mathcal{I}$ *gpv*. *finite-gpv* (*f* *x*))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

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

lemma *finite-gpv-rel''D1*:
assumes *rel-gpv''* *A C R gpv gpv'* **and** *finite-gpv* $\mathcal{I}$ *gpv* **and** $\mathcal{I}$: *rel-*$\mathcal{I}$ *C R* $\mathcal{I}$ $\mathcal{I}'$
shows *finite-gpv* $\mathcal{I}'$ *gpv'*
⟨*proof*⟩

lemma *finite-gpv-relD1*: ⟦ *rel-gpv A C gpv gpv'*; *finite-gpv* $\mathcal{I}$ *gpv*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *finite-gpv* $\mathcal{I}$ *gpv'*
⟨*proof*⟩

lemma *finite-gpv-rel''D2*: ⟦ *rel-gpv'' A C R gpv gpv'*; *finite-gpv* $\mathcal{I}$ *gpv'*; *rel-*$\mathcal{I}$ *C R* $\mathcal{I}'$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *finite-gpv* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

lemma *finite-gpv-relD2*: ⟦ *rel-gpv A C gpv gpv'*; *finite-gpv* $\mathcal{I}$ *gpv'*; *rel-*$\mathcal{I}$ *C* (=) $\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *finite-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *finite-gpv-parametric'*: (*rel-*$\mathcal{I}$ *C R* ===> *rel-gpv'' A C R* ===> (=)) *finite-gpv finite-gpv*
⟨*proof*⟩

lemma *finite-gpv-parametric* [*transfer-rule*]: (*rel-*$\mathcal{I}$ *C* (=) ===> *rel-gpv A C* ===> (=)) *finite-gpv finite-gpv*
⟨*proof*⟩

end

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

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

lemma *finite-gpv-try* [*simp*]:
finite-gpv $\mathcal{I}$ (*TRY gpv ELSE gpv'*) $\longleftrightarrow$ *finite-gpv* $\mathcal{I}$ *gpv* $\wedge$ (*colossless-gpv* $\mathcal{I}$ *gpv* $\vee$ *finite-gpv* $\mathcal{I}$ *gpv'*)
(**is** *?lhs* = -)
⟨*proof*⟩

lemma *lossless-gpv-conv-finite*:
lossless-gpv $\mathcal{I}$ *gpv* $\longleftrightarrow$ *finite-gpv* $\mathcal{I}$ *gpv* $\wedge$ *colossless-gpv* $\mathcal{I}$ *gpv*
(**is** *?loss* $\longleftrightarrow$ *?fin* $\wedge$ *?co*)
⟨*proof*⟩

lemma *colossless-gpv-try* [*simp*]:
colossless-gpv $\mathcal{I}$ (*TRY gpv ELSE gpv′*) $\longleftrightarrow$ *colossless-gpv* $\mathcal{I}$ *gpv* $\vee$ *colossless-gpv* $\mathcal{I}$ *gpv′*
(**is** *?lhs* $\longleftrightarrow$ *?gpv* $\vee$ *?gpv′*)
⟨*proof*⟩

lemma *lossless-gpv-try* [*simp*]:
lossless-gpv $\mathcal{I}$ (*TRY gpv ELSE gpv′*) $\longleftrightarrow$
finite-gpv $\mathcal{I}$ *gpv* $\wedge$ (*lossless-gpv* $\mathcal{I}$ *gpv* $\vee$ *lossless-gpv* $\mathcal{I}$ *gpv′*)
⟨*proof*⟩

lemma *interaction-any-bounded-by-imp-finite*:
assumes *interaction-any-bounded-by gpv* (*enat n*)
shows *finite-gpv* $\mathcal{I}$*-full gpv*
⟨*proof*⟩

lemma *finite-restrict-gpvI* [*simp*]: *finite-gpv* $\mathcal{I}'$ *gpv* $\Longrightarrow$ *finite-gpv* $\mathcal{I}'$ (*restrict-gpv* $\mathcal{I}$ *gpv*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*: ¬ *bad s*
and *count*: $\bigwedge s\ x\ y\ s'$. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *I s*; *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s'* ≤ *Suc* (*count s*)
and *ignore*: $\bigwedge s\ x\ y\ s'$. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s'* ≤ *count s*
and *bad*: $\bigwedge s'\ x$. ⟦ *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*: $\bigwedge s\ x\ y\ s'$. ⟦ (*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* ≤ *ennreal k* ∗ *n*
⟨*proof*⟩

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*: $\bigwedge s\ x\ y\ s'$. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *I s*; *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s'* ≤ *Suc* (*count s*)
and *ignore*: $\bigwedge s\ x\ y\ s'$. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* ∈ *outs-ℐ ℐ* ⟧ ⟹ *count s'* ≤ *count s*
and *bad*: $\bigwedge s'\ x$. ⟦ *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*: $\bigwedge s\ x\ y\ s'$. ⟦ (*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*
⟨*proof*⟩

lemma *interaction-bounded-by-exec-gpv-bad*:
assumes *interaction-any-bounded-by gpv n*
and *I s* ¬ *bad s*
and *bad*: $\bigwedge s\ x$. ⟦ *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*
⟨*proof*⟩

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

⟨*ML*⟩
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** ‹$\oplus_{\mathcal{I}}$› *500*)
is *λresp1 resp2. λout. case out of Inl out' ⇒ Inl ‘ resp1 out' | Inr out' ⇒ Inr ‘ resp2 out'* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *vimage-Inl-Plus* [*simp*]: *Inl* −‘ (*A* <+> *B*) = *A*
and *vimage-Inr-Plus* [*simp*]: *Inr* −‘ (*A* <+> *B*) = *B*
⟨*proof*⟩

lemma *vimage-Inl-image-Inr*: *Inl* −‘ *Inr* ‘ *A* = {}
and *vimage-Inr-image-Inl*: *Inr* −‘ *Inl* ‘ *A* = {}
⟨*proof*⟩

lemma *plus-ℐ-parametric* [*transfer-rule*]:
(*rel-ℐ C R* ===> *rel-ℐ C' R'* ===> *rel-ℐ* (*rel-sum C C'*) (*rel-sum R R'*)) *plus-ℐ plus-ℐ*
⟨*proof*⟩

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

lemma *ℐ-trivial-plus-ℐ* [*simp*]: *ℐ-trivial* ($\mathcal{I}_1 \oplus_{\mathcal{I}} \mathcal{I}_2$) ⟷ *ℐ-trivial* $\mathcal{I}_1$ ∧ *ℐ-trivial* $\mathcal{I}_2$

⟨*proof*⟩

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*⟩

lemma *le-plus-$\mathcal{I}$-iff* [*simp*]:
$\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \leq \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \longleftrightarrow \mathcal{I}1 \leq \mathcal{I}1' \wedge \mathcal{I}2 \leq \mathcal{I}2'$
⟨*proof*⟩

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* ≤ $\mathcal{I}2$
⟨*proof*⟩

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

context
fixes *left* :: (*'s*, *'a*, *'b*) *oracle'*
and *right* :: (*'s*,*'c*, *'d*) *oracle'*
and *s* :: *'s*
begin

primrec *plus-oracle* :: *'a* + *'c* ⇒ ((*'b* + *'d*) × *'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*]:
⟦ ⋀*a*. *x* = *Inl a* ⟹ *lossless-spmf* (*left s a*);
⋀*b*. *x* = *Inr b* ⟹ *lossless-spmf* (*right s b*) ⟧
⟹ *lossless-spmf* (*plus-oracle x*)
⟨*proof*⟩

lemma *plus-oracle-split*:
P (*plus-oracle lr*) ⟷
(∀ *x*. *lr* = *Inl x* ⟶ *P* (*map-spmf* (*apfst Inl*) (*left s x*))) ∧
(∀ *y*. *lr* = *Inr y* ⟶ *P* (*map-spmf* (*apfst Inr*) (*right s y*)))
⟨*proof*⟩

lemma *plus-oracle-split-asm*:
P (*plus-oracle lr*) ⟷
¬ ((∃ *x*. *lr* = *Inl x* ∧ ¬ *P* (*map-spmf* (*apfst Inl*) (*left s x*))) ∨
(∃ *y*. *lr* = *Inr y* ∧ ¬ *P* (*map-spmf* (*apfst Inr*) (*right s y*))))
⟨*proof*⟩

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*!]:
  ⟦ $\mathcal{I}l$ ⊢*c left s* √; $\mathcal{I}r$ ⊢*c right s* √ ⟧ ⟹ $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$ ⊢*c* (*left* $\oplus_O$ *right*) *s* √
⟨*proof*⟩

lemma *WT-plus-oracleD1*:
  assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$ ⊢*c* (*left* $\oplus_O$ *right*) *s* √ (**is** *?*$\mathcal{I}$ ⊢*c ?callee s* √)
  shows $\mathcal{I}l$ ⊢*c left s* √
⟨*proof*⟩

lemma *WT-plus-oracleD2*:
  assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$ ⊢*c* (*left* $\oplus_O$ *right*) *s* √ (**is** *?*$\mathcal{I}$ ⊢*c ?callee s* √)
  shows $\mathcal{I}r$ ⊢*c right s* √
⟨*proof*⟩

lemma *WT-plus-oracle-iff* [*simp*]: $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$ ⊢*c* (*left* $\oplus_O$ *right*) *s* √ ⟷ $\mathcal{I}l$ ⊢*c left s* √ ∧ $\mathcal{I}r$ ⊢*c right s* √
⟨*proof*⟩

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*⟩

lemma *callee-invariant-plus-oracle* [*simp*]:
  callee-invariant (*left* $\oplus_O$ *right*) *I* ⟷
   callee-invariant left I ∧ *callee-invariant right I*
  (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *rel-spmf-plus-oracle*:
  ⟦ ⋀*q1' q2'*. ⟦ *q1* = *Inl q1'*; *q2* = *Inl q2'* ⟧ ⟹ *rel-spmf* (*rel-prod B S*) (*left1 s1 q1'*) (*left2 s2 q2'*);

$\bigwedge$*q1′ q2′*. ⟦ *q1* = *Inr q1′*; *q2* = *Inr q2′* ⟧ ⟹ *rel-spmf* (*rel-prod D S*) (*right1 s1 q1′*) (*right2 s2 q2′*);
S s1 s2; *rel-sum A C q1 q2* ⟧
⟹ *rel-spmf* (*rel-prod* (*rel-sum B D*) *S*) ((*left1* $\oplus_O$ *right1*) *s1 q1*) ((*left2* $\oplus_O$ *right2*) *s2 q2*)
⟨*proof*⟩

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*]:
⟦ $\bigwedge$*a*. *x* = *Inl a* ⟹ *lossless-spmf* (*left s a*);
$\bigwedge$*b*. *x* = *Inr b* ⟹ *lossless-spmf* (*right s b*) ⟧
⟹ *lossless-spmf* (*plus-oracle-stop x*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*) √
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lossless-family-oracle*:
lossless-spmf (*family-oracle f s ix*) ⟷ *lossless-spmf* (*f* (*fst ix*) (*s* (*fst ix*)) (*snd ix*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩
end

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

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

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

end

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

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

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

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

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

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

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

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

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*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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

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

end

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
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

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* (λ*rpv 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 ≡ λ*input*. *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*
⟨*proof*⟩

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

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

end

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

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

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

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

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

⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *map-gpv′-map-gpv-swap*:
map-gpv′ f g h (*map-gpv f′ id gpv*) = *map-gpv* ($f \circ f'$) *id* (*map-gpv′ id g h gpv*)
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

s1)) *id* (*right s2 b*))

end

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*
⟨*proof*⟩

lemma *enn2real-leD*: ⟦ *enn2real x* < *y*; *x* ≠ ⊤ ⟧ ⟹ *x* < *ennreal y*
⟨*proof*⟩

lemma *ennreal-mult-le-self2I*: ⟦ *y* > *0* ⟹ *x* ≤ *1* ⟧ ⟹ *x* ∗ *y* ≤ *y* **for** *x y* :: *ennreal*
⟨*proof*⟩

lemma *ennreal-leI*: *x* ≤ *enn2real y* ⟹ *ennreal x* ≤ *y*
⟨*proof*⟩

lemma *enn2real-INF*: ⟦ *A* ≠ {}; ∀ *x*∈*A*. *f x* < ⊤ ⟧ ⟹ *enn2real* (*INF x*∈*A*. *f x*) = (*INF x*∈*A*. *enn2real* (*f x*))
⟨*proof*⟩

lemma *monotone-times-ennreal1*: *monotone* (≤) (≤) (λ*x*. *x* ∗ *y* :: *ennreal*)
⟨*proof*⟩

lemma *monotone-times-ennreal2*: *monotone* (≤) (≤) (λ*x*. *y* ∗ *x* :: *ennreal*)
⟨*proof*⟩

lemma *mono2mono-times-ennreal*[*THEN lfp.mono2mono2*, *cont-intro*, *simp*]:
shows *monotone-times-ennreal*: *monotone* (*rel-prod* (≤) (≤)) (≤) (λ(*x*, *y*). *x* ∗ *y* :: *ennreal*)
⟨*proof*⟩

lemma *mcont-times-ennreal1*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *x* ∗ *y* :: *ennreal*)
⟨*proof*⟩

lemma *mcont-times-ennreal2*: *mcont Sup* (≤) *Sup* (≤) (λ*y*. *y* ∗ *x* :: *ennreal*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *ereal-INF-cmult*: $0 < c \Longrightarrow (INF\ i{\in}I.\ c * f\ i) = ereal\ c * (INF\ i{\in}I.\ f\ i)$
⟨*proof*⟩

lemma *ereal-INF-multc*: $0 < c \Longrightarrow (INF\ i{\in}I.\ f\ i * c) = (INF\ i{\in}I.\ f\ i) * ereal\ c$
⟨*proof*⟩

lemma *INF-mult-left-ennreal*:
assumes $I = \{\} \Longrightarrow c \neq 0$
and $[\![\ c = \top;\ \exists\, i{\in}I.\ f\ i > 0\ ]\!] \Longrightarrow \exists\, p{>}0.\ \forall\, i{\in}I.\ f\ i \geq p$
shows $c * (INF\ i{\in}I.\ f\ i) = (INF\ i{\in}I.\ c * f\ i :: ennreal)$
⟨*proof*⟩ **including** *ennreal.lifting*
⟨*proof*⟩

lemma *pmf-map-spmf-None*: *pmf* (*map-spmf f p*) *None* = *pmf p None*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *INF-UNION*: $(INF\ z \in \bigcup x{\in}A.\ B\ x.\ f\ z) = (INF\ x{\in}A.\ INF\ z{\in}B\ x.\ f\ z)$
for *f* :: - ⇒ *'b*::*complete-lattice*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *cont-nn-integral-spmf*:
cont lub-spmf (*ord-spmf* (=)) *Sup* (≤) (λ*p* :: *'a spmf*. *nn-integral* (*measure-spmf p*) *f*)

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *nn-integral-mono2mono*:
assumes ⋀*x*. *x* ∈ *space M* ⟹ *monotone ord* (≤) (λ*f*. *F f x*)
shows *monotone ord* (≤) (λ*f*. *nn-integral M* (*F f*))
⟨*proof*⟩

lemma *nn-integral-mono-lfp* [*partial-function-mono*]:
— `Partial_Function.mono_tac` does not like conditional assumptions (more precisely the case splitter)
(⋀*x*. *lfp.mono-body* (λ*f*. *F f x*)) ⟹ *lfp.mono-body* (λ*f*. *nn-integral M* (*F f*))
⟨*proof*⟩

lemma *INF-mono-lfp* [*partial-function-mono*]:
(⋀*x*. *lfp.mono-body* (λ*f*. *F f x*)) ⟹ *lfp.mono-body* (λ*f*. *INF x*∈*M*. *F f x*)
⟨*proof*⟩

lemmas *parallel-fixp-induct-1-2* = *parallel-fixp-induct-uc*[
of - - - - λ*x*. *x* - λ*x*. *x case-prod* - *curry*,
where *P*=λ*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* (≤) (≤) (λ*x*. *x* + *y* :: *ennreal*)
⟨*proof*⟩

lemma *monotone-ennreal-add2*: *monotone* (≤) (≤) (λ*y*. *x* + *y* :: *ennreal*)
⟨*proof*⟩

lemma *mono2mono-ennreal-add*[*THEN lfp.mono2mono2*, *cont-intro*, *simp*]:
shows *monotone-eadd*: *monotone* (*rel-prod* (≤) (≤)) (≤) (λ(*x*, *y*). *x* + *y* :: *ennreal*)
⟨*proof*⟩

lemma *ennreal-add-partial-function-mono* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord* (≤)) (≤) *f*; *monotone* (*fun-ord* (≤)) (≤) *g* ⟧
⟹ *monotone* (*fun-ord* (≤)) (≤) (λ*x*. *f x* + *g x* :: *ennreal*)
⟨*proof*⟩

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*-$\mathcal{I}$ $\mathcal{I}$ *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*-$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv'* (*c r*)) ∂*measure-spmf* (*the-gpv gpv*)) + *fail* ∗ *pmf* (*the-gpv gpv*) *None*)
shows *P expectation-gpv*
⟨*proof*⟩

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

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

lemma *expectation-gpv-lift-spmf* [*simp*]:
expectation-gpv (*lift-spmf p*) = (∫⁺ *x*. *f x* ∂*measure-spmf p*) + *fail* ∗ *pmf p None*
⟨*proof*⟩

lemma *expectation-gpv-Pause* [*simp*]:
expectation-gpv (*Pause out c*) = (*INF r*∈*responses*-$\mathcal{I}$ $\mathcal{I}$ *out*. *expectation-gpv* (*c r*))
⟨*proof*⟩

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*-$\mathcal{I}$ *C R* ===> (*A* ===> (=)) ===> *rel-gpv''* *A C R* ===> (=)) *expectation-gpv expectation-gpv*
⟨*proof*⟩
end

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

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'*
⟨*proof*⟩

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*⟩

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*⟩

lemma *expectation-gpv-mono-strong*:
fixes *fail fail'*
assumes *fail*: ¬ *colossless-gpv* $\mathcal{I}$ *gpv* $\Longrightarrow$ *fail* ≤ *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* ≤ *expectation-gpv fail'* $\mathcal{I}$ *g gpv*
⟨*proof*⟩

lemma *expectation-gpv-bind* [*simp*]:
fixes $\mathcal{I}$ *f g fail*
defines *expectation-gpv1* ≡ *expectation-gpv fail* $\mathcal{I}$ *f*
and *expectation-gpv2* ≡ *expectation-gpv fail* $\mathcal{I}$ (*expectation-gpv fail* $\mathcal{I}$ *f* ∘ *g*)
shows *expectation-gpv1* (*bind-gpv gpv g*) = *expectation-gpv2 gpv* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

lemma *expectation-gpv-restrict-gpv*:
$\mathcal{I} \vdash g$ *gpv* $\surd \Longrightarrow$ *expectation-gpv fail* $\mathcal{I}$ *f* (*restrict-gpv* $\mathcal{I}$ *gpv*) = *expectation-gpv fail* $\mathcal{I}$ *f gpv* **for** *fail*
⟨*proof*⟩

lemma *expectation-gpv-const-le*: $\mathcal{I} \vdash g$ *gpv* $\surd \Longrightarrow$ *expectation-gpv fail* $\mathcal{I}$ (λ-. *c*) *gpv* $\leq$ *max c fail* **for** *fail*
⟨*proof*⟩

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*⟩

lemma *expectation-gpv-cmult*:
fixes *fail*
assumes $0 < c$ **and** $c \neq \top$
shows *c* ∗ *expectation-gpv fail* $\mathcal{I}$ *f gpv* = *expectation-gpv* (*c* ∗ *fail*) $\mathcal{I}$ (λx. *c* ∗ *f x*) *gpv*
⟨*proof*⟩

lemma *expectation-gpv-le-exec-gpv*:
assumes *callee*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ $\Longrightarrow$ *lossless-spmf* (*callee s x*)
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *WT-callee*: $\bigwedge s.\ \mathcal{I} \vdash c$ *callee s* $\surd$
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* $\leq \int^{+}$ (*x*, *s*). *f x* ∂*measure-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *weight-gpv-Fail* [*simp*]: *weight-gpv* $\mathcal{I}$ *Fail* = *0*
⟨*proof*⟩

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

lemma *weight-gpv-Pause* [*simp*]:
($\bigwedge r.\ r \in$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* $\Longrightarrow \mathcal{I} \vdash g$ *c r* $\surd$)
$\Longrightarrow$ *weight-gpv* $\mathcal{I}$ (*Pause out c*) = (*if out* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ *then INF r*$\in$*responses-*$\mathcal{I}$ $\mathcal{I}$ *out*. *weight-gpv* $\mathcal{I}$ (*c r*) *else 0*)
⟨*proof*⟩

lemma *weight-gpv-nonneg*: *0* $\leq$ *weight-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *weight-gpv-le-1*: $\mathcal{I} \vdash g$ *gpv* $\surd \Longrightarrow$ *weight-gpv* $\mathcal{I}$ *gpv* ≤ 1
⟨*proof*⟩

theorem *weight-exec-gpv*:
assumes *callee*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *lossless-spmf* (*callee s x*)
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *WT-callee*: $\bigwedge s.\ \mathcal{I} \vdash c$ *callee s* $\surd$
shows *weight-gpv* $\mathcal{I}$ *gpv* $\leq$ *weight-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

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

6.5 Probabilistic termination

definition *pgen-lossless-gpv* :: *ennreal* $\Rightarrow$ ('*c*, '*r*) $\mathcal{I} \Rightarrow$ ('*a*, '*c*, '*r*) *gpv* $\Rightarrow$ *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*) $\mathcal{I} \Rightarrow$ ('*a*, '*c*, '*r*) *gpv* $\Rightarrow$ *bool*
where *plossless-gpv* $\equiv$ *pgen-lossless-gpv 0*

abbreviation *pfinite-gpv* :: ('*c*, '*r*) $\mathcal{I} \Rightarrow$ ('*a*, '*c*, '*r*) *gpv* $\Rightarrow$ *bool*
where *pfinite-gpv* $\equiv$ *pgen-lossless-gpv 1*

lemma *pgen-lossless-gpvI* [*intro?*]: *expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1* $\Longrightarrow$ *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* **for** *fail*
⟨*proof*⟩

lemma *pgen-lossless-gpvD*: *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* $\Longrightarrow$ *expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1* **for** *fail*
⟨*proof*⟩

lemma *lossless-imp-plossless-gpv*:
assumes *lossless-gpv* $\mathcal{I}$ *gpv* $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *plossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *finite-imp-pfinite-gpv*:
assumes *finite-gpv* $\mathcal{I}$ *gpv* $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *pfinite-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *plossless-gpv-lossless-spmfD*:

assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *lossless-spmf* (*the-gpv gpv*)
⟨*proof*⟩

lemma
shows *plossless-gpv-ContD*:
⟦ *plossless-gpv* $\mathcal{I}$ *gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧
⟹ *plossless-gpv* $\mathcal{I}$ (*c input*)
and *pfinite-gpv-ContD*:
⟦ *pfinite-gpv* $\mathcal{I}$ *gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧
⟹ *pfinite-gpv* $\mathcal{I}$ (*c input*)
⟨*proof*⟩

lemma *plossless-iff-colossless-pfinite*:
assumes *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *plossless-gpv* $\mathcal{I}$ *gpv* ⟷ *colossless-gpv* $\mathcal{I}$ *gpv* ∧ *pfinite-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *pgen-lossless-gpv-Done* [*simp*]: *pgen-lossless-gpv fail* $\mathcal{I}$ (*Done x*) **for** *fail*
⟨*proof*⟩

lemma *pgen-lossless-gpv-Fail* [*simp*]: *pgen-lossless-gpv fail* $\mathcal{I}$ *Fail* ⟷ *fail* = *1* **for** *fail*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pgen-lossless-gpv-lift-spmf* [*simp*]:
pgen-lossless-gpv fail $\mathcal{I}$ (*lift-spmf p*) ⟷ *lossless-spmf p* ∨ *fail* = *1* **for** *fail*
⟨*proof*⟩

lemma *expectation-gpv-top-pfinite*:
assumes *pfinite-gpv* $\mathcal{I}$ *gpv*
shows *expectation-gpv* ⊤ $\mathcal{I}$ (λ-. ⊤) *gpv* = ⊤
⟨*proof*⟩

lemma *pfinite-INF-le-expectation-gpv*:

fixes *fail* $\mathcal{I}$ *gpv* *f*
defines $c \equiv min$ (*INF* $x \in$*results-gpv* $\mathcal{I}$ *gpv*. *f x*) *fail*
assumes *fin*: *pfinite-gpv* $\mathcal{I}$ *gpv*
shows $c \le$ *expectation-gpv fail* $\mathcal{I}$ *f gpv* (**is** *?lhs* $\le$ *?rhs*)
⟨*proof*⟩

lemma *plossless-INF-le-expectation-gpv*:
fixes *fail*
assumes *plossless-gpv* $\mathcal{I}$ *gpv* **and** $\mathcal{I} \vdash g$ *gpv* $\surd$
shows (*INF* $x \in$*results-gpv* $\mathcal{I}$ *gpv*. *f x*) $\le$ *expectation-gpv fail* $\mathcal{I}$ *f gpv* (**is** *?lhs* $\le$ *?rhs*)
⟨*proof*⟩

lemma *expectation-gpv-le-inline*:
fixes $\mathcal{I}'$
defines *expectation-gpv2* $\equiv$ *expectation-gpv 0* $\mathcal{I}'$
assumes *callee*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *callee'*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *results-gpv* $\mathcal{I}'$ (*callee s x*) $\subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* $\times$ *UNIV*
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *WT-callee*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow \mathcal{I}' \vdash g$ *callee s x* $\surd$
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* $\le$ *expectation-gpv2* ($\lambda(x, s)$. *f x*) (*inline callee gpv s*)
⟨*proof*⟩

lemma *plossless-inline*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *callee'*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *results-gpv* $\mathcal{I}'$ (*callee s x*) $\subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* $\times$ *UNIV*
and *WT-callee*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow \mathcal{I}' \vdash g$ *callee s x* $\surd$
shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
⟨*proof*⟩

lemma *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-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *lossless-spmf* (*callee s x*)
and *callee'*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *set-spmf* (*callee s x*) $\subseteq$ *responses-*$\mathcal{I}$ $\mathcal{I}$ *x* $\times$ *UNIV*
shows *lossless-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

lemma *expectation-gpv-*$\mathcal{I}$*-mono*:
defines *expectation-gpv'* $\equiv$ *expectation-gpv*
assumes *le*: $\mathcal{I} \le \mathcal{I}'$
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$

shows *expectation-gpv fail* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv′ fail* $\mathcal{I}'$ *f gpv*
⟨*proof*⟩

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

lemma *plossless-gpv-mono*:
⟦ *plossless-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g$ *gpv* √ ⟧ ⟹ *plossless-gpv* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

lemma *pfinite-gpv-mono*:
⟦ *pfinite-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \leq \mathcal{I}'$; $\mathcal{I} \vdash g$ *gpv* √ ⟧ ⟹ *pfinite-gpv* $\mathcal{I}'$ *gpv*
⟨*proof*⟩

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

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

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

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

lemma *expectation-gpv-le-inline*:
defines *expectation-gpv2* ≡ *expectation-gpv 0* $\mathcal{I}'$
assumes *callee*: ⋀*s x*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* √
and *I*: *I s*
shows *expectation-gpv 0* $\mathcal{I}$ *f gpv* ≤ *expectation-gpv2* (λ(*x*, *s*). *f x*) (*inline callee gpv s*)
⟨*proof*⟩

lemma *plossless-inline*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* √
and *callee*: ⋀*s x*. ⟦ *I s*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *plossless-gpv* $\mathcal{I}'$ (*callee s x*)
and *I*: *I s*

shows *plossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
⟨*proof*⟩

end

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

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

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

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

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

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

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

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

lemma (**in** *callee-invariant-on*) *plossless-exec-gpv*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *callee*: $\bigwedge$*s x*. ⟦ *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ ⟹ *lossless-spmf* (*callee s x*)
and *I*: *I s*

shows *lossless-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

end

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

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

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

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

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

end

theory *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*: $\bigwedge$*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*)
⟨*proof*⟩

lemma *exec-gpv-oracle-bisim*:
assumes ∗: *X s1 s2*
and *bisim*: $\bigwedge$*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*: $\bigwedge$*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*)
⟨*proof*⟩

lemma *run-gpv-oracle-bisim*:
assumes *X s1 s2*
and $\bigwedge$*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*
⟨*proof*⟩

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 $\bigwedge$*exec-until-bad'*. *P exec-until-bad'* $\Longrightarrow$
P (λ*gpv s1 s2*. *if bad1 s1* $\vee$ *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 $\Rightarrow$ *return-spmf* ((*x*, *s1*), (*x*, *s2*))
| *IO out f* $\Rightarrow$ *bind-spmf* (*joint-oracle* (*s1*, *s2*) *out*) (λ((*x*, *s1'*), (*y*, *s2'*)).
if bad1 s1' $\vee$ *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*
⟨*proof*⟩

end

lemma *exec-gpv-oracle-bisim-bad-plossless*:
fixes *s1* :: *'s1* **and** *s2* :: *'s2* **and** *X* :: *'s1* $\Rightarrow$ *'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-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-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1'* = *bad2 s2'* $\wedge$ (*if bad2 s2' then X-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-invariant-on oracle1* (λ*s1*. *bad1 s1* $\wedge$ *X-bad s1 s2*) $\mathcal{I}$
and *bad-sticky2*: $\bigwedge$*s1*. *bad1 s1* $\Longrightarrow$ *callee-invariant-on oracle2* (λ*s2*. *bad2 s2* $\wedge$ *X-bad s1 s2*) $\mathcal{I}$
and *lossless1*: $\bigwedge$*s1 x*. ⟦ *bad1 s1*; *x* $\in$ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *lossless-spmf* (*oracle1 s1 x*)
and *lossless2*: $\bigwedge$*s2 x*. ⟦ *bad2 s2*; *x* $\in$ *outs-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *lossless-spmf* (*oracle2 s2 x*)
and *lossless*: *plossless-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-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1'* = *bad2 s2'* $\wedge$ (*if bad2 s2' then X-bad s1' s2' else a* = *b* $\wedge$ *X s1' s2'*)) (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
(**is** *rel-spmf ?R ?p ?q*)
⟨*proof*⟩

lemma *exec-gpv-oracle-bisim-bad'*:
fixes *s1* :: *'s1* **and** *s2* :: *'s2* **and** *X* :: *'s1* $\Rightarrow$ *'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-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-$\mathcal{I}$ $\mathcal{I}$* ⟧ $\Longrightarrow$ *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1'* = *bad2 s2'* $\wedge$ (*if bad2 s2' then X-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-invariant-on oracle1* (λ*s1*. *bad1 s1* $\wedge$

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*: *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* √
  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*)
⟨*proof*⟩

lemma *exec-gpv-oracle-bisim-bad-invariant*:
  fixes *s1* :: *'s1* **and** *s2* :: *'s2* **and** *X* :: *'s1* ⇒ *'s2* ⇒ *bool* **and** *I1* :: *'s1* ⇒ *bool* **and** *I2* :: *'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*; *I1 s1*; *I2 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*; *I2 s2* ⟧ ⟹ *callee-invariant-on oracle1* (λ*s1*. *bad1 s1* ∧ *X-bad s1 s2*) 𝓘
  and *bad-sticky2*: ⋀*s1*. ⟦ *bad1 s1*; *I1 s1* ⟧ ⟹ *callee-invariant-on oracle2* (λ*s2*. *bad2 s2* ∧ *X-bad s1 s2*) 𝓘
  and *lossless1*: ⋀*s1 x*. ⟦ *bad1 s1*; *I1 s1*; *x* ∈ *outs-𝓘 𝓘* ⟧ ⟹ *lossless-spmf* (*oracle1 s1 x*)
  and *lossless2*: ⋀*s2 x*. ⟦ *bad2 s2*; *I2 s2*; *x* ∈ *outs-𝓘 𝓘* ⟧ ⟹ *lossless-spmf* (*oracle2 s2 x*)
  and *lossless*: *lossless-gpv 𝓘 gpv*
  and *WT-gpv*: 𝓘 ⊢g *gpv* √
  and *I1*: *callee-invariant-on oracle1 I1 𝓘*
  and *I2*: *callee-invariant-on oracle2 I2 𝓘*
  and *s1*: *I1 s1*
  and *s2*: *I2 s2*
  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*)
  including *lifting-syntax*
⟨*proof*⟩

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*: $\bigwedge$*s2*. *bad2 s2* $\Longrightarrow$ *callee-invariant-on oracle1* (λ*s1*. *bad1 s1* $\wedge$ *X-bad s1 s2*) $\mathcal{I}$
  and *bad-sticky2*: $\bigwedge$*s1*. *bad1 s1* $\Longrightarrow$ *callee-invariant-on oracle2* (λ*s2*. *bad2 s2* $\wedge$ *X-bad s1 s2*) $\mathcal{I}$
  and *lossless1*: $\bigwedge$*s1 x*. *bad1 s1* $\Longrightarrow$ *lossless-spmf* (*oracle1 s1 x*)
  and *lossless2*: $\bigwedge$*s2 x*. *bad2 s2* $\Longrightarrow$ *lossless-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* $\surd$
  and *R*: $\bigwedge$*a s1 b s2*. $[\![$ *bad1 s1* = *bad2 s2*; $\neg$ *bad2 s2* $\Longrightarrow$ *a* = *b* $\wedge$ *X s1 s2*; *bad2 s2* $\Longrightarrow$ *X-bad s1 s2* $]\!]$ $\Longrightarrow$ *R* (*a*, *s1*) (*b*, *s2*)
  shows *rel-spmf R* (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
⟨*proof*⟩

lemma *exec-gpv-oracle-bisim-bad-full*:
  assumes *X s1 s2*
  and *bad1 s1* = *bad2 s2*
  and $\bigwedge$*s1 s2 x*. *X s1 s2* $\Longrightarrow$ *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1'* = *bad2 s2'* $\wedge$ ($\neg$ *bad2 s2'* $\longrightarrow$ *a* = *b* $\wedge$ *X s1' s2'*)) (*oracle1 s1 x*) (*oracle2 s2 x*)
  and *callee-invariant oracle1 bad1*
  and *callee-invariant oracle2 bad2*
  and $\bigwedge$*s1 x*. *bad1 s1* $\Longrightarrow$ *lossless-spmf* (*oracle1 s1 x*)
  and $\bigwedge$*s2 x*. *bad2 s2* $\Longrightarrow$ *lossless-spmf* (*oracle2 s2 x*)
  and *lossless-gpv* $\mathcal{I}$*-full gpv*
  and *R*: $\bigwedge$*a s1 b s2*. $[\![$ *bad1 s1* = *bad2 s2*; $\neg$ *bad2 s2* $\Longrightarrow$ *a* = *b* $\wedge$ *X s1 s2* $]\!]$ $\Longrightarrow$ *R* (*a*, *s1*) (*b*, *s2*)
  shows *rel-spmf R* (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
⟨*proof*⟩

lemma *max-enn2ereal*: *max* (*enn2ereal x*) (*enn2ereal y*) = *enn2ereal* (*max x y*)
including *ennreal.lifting* ⟨*proof*⟩

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* $\times$ {*False*}) = *measure* (*measure-spmf* (*map-spmf* (λ*x*. (*f x*, *bad x*)) *q*)) (*A* $\times$ {*False*})
  shows |*measure* (*measure-spmf* (*map-spmf f p*)) *A* − *measure* (*measure-spmf* (*map-spmf f q*)) *A*| $\leq$ *spmf* (*map-spmf bad p*) *True*
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *exec-gpv-bind-materialize*:
  fixes *f* :: *'s* $\Rightarrow$ *'r spmf*
  and *g* :: *'x* $\times$ *'s* $\Rightarrow$ *'r* $\Rightarrow$ *'y spmf*
  and *s* :: *'s*
  defines *exec-gpv2* $\equiv$ *exec-gpv*
  assumes *cond*: $\bigwedge$*s x y s'*. $[\![$ (*y*, *s'*) $\in$ *set-spmf* (*callee s x*); *I s* $]\!]$ $\Longrightarrow$ *f s* = *f s'*
  and $\mathcal{I}$: $\mathcal{I}$ = $\mathcal{I}$*-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'* $\wedge$ *r* = *None then map-spmf* (λ*r*. (*y*, (*Some r*, *s'*))) (*f s'*) *else return-spmf* (*y*, (*r*, *s'*)))) *gpv* (*None*, *s*)
$\gg=$ (λ(*a*, *r*, *s*). *case r of None* $\Rightarrow$ *bind-spmf* (*f s*) (*g* (*a*, *s*)) | *Some r'* $\Rightarrow$ *g* (*a*, *s*) *r'*)
(**is** *?lhs* = *?rhs* **is** - = *bind-spmf* (*exec-gpv2 ?callee2* - -) -)
⟨*proof*⟩

primcorec *gpv-stop* :: (*'a*, *'c*, *'r*) *gpv* $\Rightarrow$ (*'a option*, *'c*, *'r option*) *gpv*
where
the-gpv (*gpv-stop gpv*) =
map-spmf (*map-generat Some id* (λ*rpv input. case input of None* $\Rightarrow$ *Done None* | *Some input'* $\Rightarrow$ *gpv-stop* (*rpv input'*)))
(*the-gpv gpv*)

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

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

lemma *gpv-stop-Pause* [*simp*]: *gpv-stop* (*Pause out rpv*) = *Pause out* (λ*input. case input of None* $\Rightarrow$ *Done None* | *Some input'* $\Rightarrow$ *gpv-stop* (*rpv input'*))
⟨*proof*⟩

lemma *gpv-stop-lift-spmf* [*simp*]: *gpv-stop* (*lift-spmf p*) = *lift-spmf* (*map-spmf Some p*)
⟨*proof*⟩

lemma *gpv-stop-bind* [*simp*]:
gpv-stop (*bind-gpv gpv f*) = *bind-gpv* (*gpv-stop gpv*) (λ*x. case x of None* $\Rightarrow$ *Done None* | *Some x'* $\Rightarrow$ *gpv-stop* (*f x'*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *gpv-stop-parametric* [*transfer-rule*]:
shows (*rel-gpv A C* ===> *rel-gpv* (*rel-option A*) *C*) *gpv-stop gpv-stop*
⟨*proof*⟩

lemma *gpv-stop-transfer*:

(*rel-gpv'' A B C ===> rel-gpv'' (pcr-Some A) B (pcr-Some C)*) (*λx. x*) *gpv-stop*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *interaction-bound-gpv-stop* [*simp*]:
interaction-bound consider (*gpv-stop gpv*) = *interaction-bound consider gpv*
⟨*proof*⟩

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*) (*λgenerat. 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*) (*λgenerat. 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*:
$\bigwedge$*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*: $\bigwedge$*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*⟩

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*
⟨*proof*⟩

lemma *ap-gpv-comp*: *pure-gpv* (∘) ⋄ *u* ⋄ *v* ⋄ *w* = *u* ⋄ (*v* ⋄ *w*)
⟨*proof*⟩

lemma *ap-gpv-homo*: *pure-gpv f* ⋄ *pure-gpv x* = *pure-gpv* (*f x*)
⟨*proof*⟩

lemma *ap-gpv-interchange*: $u \diamond$ *pure-gpv* x = *pure-gpv* $(\lambda f.\ f\ x) \diamond u$
⟨*proof*⟩

applicative *gpv*
for
pure: *pure-gpv*
ap: *ap-gpv*
⟨*proof*⟩

lemma *map-conv-ap-gpv*: *map-gpv* f $(\lambda x.\ x)$ *gpv* = *pure-gpv* $f \diamond$ *gpv*
⟨*proof*⟩

lemma *exec-gpv-ap*:
exec-gpv callee $(f \diamond x)\ \sigma$ =
exec-gpv callee $f\ \sigma \gg\!= (\lambda(f', \sigma'))$. *pure-spmf* $(\lambda(x', \sigma'').\ (f'\ x', \sigma'')) \diamond$ *exec-gpv callee* $x\ \sigma'$)
⟨*proof*⟩

lemma *exec-gpv-ap-pure* [*simp*]:
exec-gpv callee (*pure-gpv* $f \diamond x$) σ = *pure-spmf* (*apfst* f) $\diamond$ *exec-gpv callee* $x\ \sigma$
⟨*proof*⟩

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* $[\urcorner]_G$ *n*)
begin

lemma *generatorE* [*elim?*]:
assumes *x* ∈ *carrier G*
obtains *n* :: *nat* **where** *x* = *generator G* [⌉] *n*
⟨*proof*⟩

lemma *inj-on-generator*: *inj-on* (([⌉]) **g**) {..<*order G*}

⟨*proof*⟩

lemma *finite-carrier*: *finite* (*carrier G*)
⟨*proof*⟩

lemma *carrier-conv-generator*: *carrier G* = (*λn*. **g** [^] *n*) ‘ {..<*order G*}
⟨*proof*⟩

lemma *bij-betw-generator-carrier*:
bij-betw (*λn* :: *nat*. **g** [^] *n*) {..<*order G*} (*carrier G*)
⟨*proof*⟩

lemma *order-gt-0*: *order G* > *0*
⟨*proof*⟩

end

lemma (**in** *monoid*) *order-in-range-Suc*: *order G* ∈ *range Suc* ⟷ *finite* (*carrier G*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *weight-sample-uniform*: *weight-spmf* (*sample-uniform n*) = *indicator* (*range Suc*) *n*
⟨*proof*⟩

lemma *weight-sample-uniform-0* [*simp*]: *weight-spmf* (*sample-uniform 0*) = *0*
⟨*proof*⟩

lemma *weight-sample-uniform-gt-0* [*simp*]: *0* < *n* ⟹ *weight-spmf* (*sample-uniform n*) = *1*
⟨*proof*⟩

lemma *lossless-sample-uniform* [*simp*]: *lossless-spmf* (*sample-uniform n*) ⟷ *0* < *n*

⟨*proof*⟩

lemma *set-spmf-sample-uniform* [*simp*]: $0 < n \Longrightarrow$ *set-spmf* (*sample-uniform n*) = {..<*n*}
⟨*proof*⟩

lemma (**in** *cyclic-group*) *sample-uniform-one-time-pad*:
 assumes [*simp*]: $c \in$ *carrier G*
 shows
 map-spmf (λx. **g** [^] $x \otimes c$) (*sample-uniform* (*order G*)) =
 map-spmf (λx. **g** [^] x) (*sample-uniform* (*order G*))
 (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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.