CryptHOL

Andreas Lochbihler

August 16, 2018

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].

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
begin

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

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*⟩

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* ⟹ *transp* (λ*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 ∃*f*. *bij-betw f A B* ∧ (∀ *x*∈*A*. *R x* (*f x*))
⟨*proof*⟩

definition *restrict-relp* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a* ⇒ *bool*) ⇒ ('*b* ⇒ *bool*) ⇒ '*a* ⇒ '*b* ⇒ *bool*
(- ↾ (- ⊗ -) [*53*, *54*, *54*] *53*)
where *restrict-relp R P Q* = (λ*x y*. *R x y* ∧ *P x* ∧ *Q y*)

lemma *restrict-relp-apply* [*simp*]: $(R \restriction P \otimes Q)\ x\ y \longleftrightarrow R\ x\ y \wedge P\ x \wedge Q\ y$
⟨*proof*⟩

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

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

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

lemma *restrict-relp-restrict-relp* [*simp*]: $R \restriction P \otimes Q \restriction P' \otimes Q' = R \restriction inf\ P\ P' \otimes inf\ Q\ Q'$
⟨*proof*⟩

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

lemma *restrict-relp-cong-simp*:
⟦ $P = P'$; $Q = Q'$; $\bigwedge x\ y.\ P\ x$ =simp=> $Q\ y$ =simp=> $R\ x\ y = R'\ x\ y$ ⟧ ⟹ $R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$
⟨*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*: ⟦ $R \leq R'$; $P \leq P'$; $Q \leq Q'$ ⟧ ⟹ $R \upharpoonright P \otimes Q \leq R' \upharpoonright P' \otimes Q'$
⟨*proof*⟩

lemma *restrict-relp-mono'*:
⟦ $(R \upharpoonright P \otimes Q)\ x\ y$; ⟦ $R\ x\ y$; $P\ x$; $Q\ y$ ⟧ ⟹ $R'\ x\ y$ &&& $P'\ x$ &&& $Q'\ y$ ⟧
⟹ $(R' \upharpoonright P' \otimes Q')\ x\ y$
⟨*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* $(\lambda\text{-}.\ True) \otimes$ *pred-prod I2* $(\lambda\text{-}.\ True)$
⟨*proof*⟩

lemma *restrict-rel-prod2*:
rel-prod R (*S* ↾ *I1* ⊗ *I2*) = *rel-prod R S* ↾ *pred-prod* (λ-. *True*) *I1* ⊗ *pred-prod* (λ-. *True*) *I2*
⟨*proof*⟩

1.3 Pairs

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

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

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

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

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*⟩

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*; ⋀*a*. ⟦ *a* ∈ *set-option x*; *P a* ⟧ ⟹ *P′ a* ⟧ ⟹ *pred-option P′ x*
⟨*proof*⟩

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

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

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

lemma *pred-option-conj* [*simp*]:
pred-option (λ*x*. *P x* ∧ *Q x*) = (λ*x*. *pred-option P x* ∧ *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* ⟧ ⟹ *rel-option* (*R* ↾ *P* ⊗ *Q*) *x y*
⟨*proof*⟩

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

lemma *option-rel-map-restrict-relp*:
shows *option-rel-map-restrict-relp1*:
rel-option (*R* ↾ *P* ⊗ *Q*) (*map-option f x*) = *rel-option* (*R* ∘ *f* ↾ *P* ∘ *f* ⊗ *Q*) *x*
and *option-rel-map-restrict-relp2*:
rel-option (*R* ↾ *P* ⊗ *Q*) *x* (*map-option g y*) = *rel-option* ((λ*x*. *R x* ∘ *g*) ↾ *P* ⊗ *Q* ∘ *g*) *x 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 Maps

lemma *map-add-apply*: $(m1 \mathbin{++} m2)\ x = \mathit{sup\text{-}option}\ (m1\ x)\ (m2\ x)$
⟨*proof*⟩

lemma *map-le-map-upd2*: $[\![\ f \subseteq_m g;\ \bigwedge y'.\ f\ x = \mathit{Some}\ y' \Longrightarrow y' = y\ ]\!] \Longrightarrow f \subseteq_m g(x \mapsto y)$
⟨*proof*⟩

lemma *eq-None-iff-not-dom*: $f\ x = \mathit{None} \longleftrightarrow x \notin \mathit{dom}\ f$
⟨*proof*⟩

lemma *card-ran-le-dom*: $\mathit{finite}\ (\mathit{dom}\ m) \Longrightarrow \mathit{card}\ (\mathit{ran}\ m) \leq \mathit{card}\ (\mathit{dom}\ m)$
⟨*proof*⟩

lemma *dom-subset-ran-iff*:
assumes $\mathit{finite}\ (\mathit{ran}\ m)$
shows $\mathit{dom}\ m \subseteq \mathit{ran}\ m \longleftrightarrow \mathit{dom}\ m = \mathit{ran}\ m$
⟨*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*]: $\mathit{Map\text{-}empty} \equiv \mathit{Map.empty}$

lemma *map-le-Some1D*: $[\![\ m \subseteq_m m';\ m\ x = \mathit{Some}\ y\ ]\!] \Longrightarrow m'\ x = \mathit{Some}\ y$
⟨*proof*⟩

lemma *map-le-fun-upd2*: $[\![\ f \subseteq_m g;\ x \notin \mathit{dom}\ f\ ]\!] \Longrightarrow f \subseteq_m g(x := y)$
⟨*proof*⟩

lemma *map-eqI*: $\forall x \in \mathit{dom}\ m \cup \mathit{dom}\ m'.\ m\ x = m'\ x \Longrightarrow m = m'$
⟨*proof*⟩

1.6 Countable

lemma *countable-lfp*:
assumes *step*: $\bigwedge Y.\ \mathit{countable}\ Y \Longrightarrow \mathit{countable}\ (F\ Y)$
and *cont*: *Order-Continuity.sup-continuous F*
shows $\mathit{countable}\ (\mathit{lfp}\ F)$
⟨*proof*⟩

lemma *countable-lfp-apply*:
assumes *step*: $\bigwedge Y\ x.\ (\bigwedge x.\ \mathit{countable}\ (Y\ x)) \Longrightarrow \mathit{countable}\ (F\ Y\ x)$
and *cont*: *Order-Continuity.sup-continuous F*
shows $\mathit{countable}\ (\mathit{lfp}\ F\ x)$
⟨*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*: $A \neq \{\} \Longrightarrow eSuc\ (SUPREMUM\ A\ f) = (SUP\ x{:}A.\ eSuc\ (f\ x))$
⟨*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*]: $(SUP\ x{:}A.\ f\ x) = (0 :: enat) \longleftrightarrow (\forall x{\in}A.\ f\ x = 0)$
⟨*proof*⟩

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

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

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

lemma *SUP-iadd-le-iff*: $(SUP\ x{:}A.\ f\ x :: enat) + n \leq y \longleftrightarrow (if\ A = \{\}\ then\ n \leq y\ else\ \forall x{\in}A.\ f\ x + n \leq y)$
⟨*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-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-fun* (*BNF-Def.Grp A Abs*) (*BNF-Def.Grp B g*) = *BNF-Def.Grp* $\{f.\ f` A \subseteq B\}$ (*Rep* −−−> *g*)
⟨*proof*⟩

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

lemma *rel-set-comp-Grp*:
  rel-set R = (*BNF-Def.Grp* $\{x.\ x \subseteq \{(x,\ y).\ R\ x\ y\}\}$ ((`) *fst*))$^{-1-1}$ *OO BNF-Def.Grp* $\{x.\ x \subseteq \{(x,\ y).\ R\ x\ y\}\}$ ((`) *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*) ⟷ *A* = *UNIV*
⟨*proof*⟩

lemma *right-total-Grp* [*iff*]:
right-total (*BNF-Def.Grp A f*) ⟷ *f* ‘ *A* = *UNIV*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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* $n \leq 1 \longleftrightarrow n \leq 1$
⟨*proof*⟩

lemma (**in** *linordered-semidom*) *mult-right-le*: $c \leq 1 \Longrightarrow 0 \leq a \Longrightarrow c * a \leq a$
⟨*proof*⟩

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

lemma *fun-ordD*: *fun-ord ord f g* $\Longrightarrow$ *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*: $\bigwedge x\ y.$ ⟦ *orda x* (*ccpo.fixp luba orda f*); *ordb y* (*ccpo.fixp lubb ordb g*); *P x y* ⟧ $\Longrightarrow$ *P* (*f x*) (*g y*)
shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
⟨*proof*⟩

lemma *parallel-fixp-induct-strong-uc*:
assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: $\bigwedge x.$ *monotone* (*fun-ord orda*) *orda* (λ*f*. *U1* (*F* (*C1 f*)) *x*)
and *G*: $\bigwedge y.$ *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*)) (λ*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 - - - - λ*x*. *x* - λ*x*. *x* λ*x*. *x* - λ*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*=λ*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*: $f \equiv C\ (ccpo.fixp\ (fun\text{-}lub\ (flat\text{-}lub\ None))\ (fun\text{-}ord\ option\text{-}ord)\ (\lambda f.\ U\ (F\ (C\ f))))$
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*]: $(\lambda\text{-}.\ bot :: 'a :: order\text{-}bot) \leq x$
⟨*proof*⟩

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

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

lemma *finite-chainsD*: $[\![finite\text{-}chains\ ord;\ Complete\text{-}Partial\text{-}Order.chain\ ord\ Y\ ]\!] \Longrightarrow finite\ Y$
⟨*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**

$(A ===> B ===> C)\ f\ g \longleftrightarrow (\mathit{rel\text{-}prod}\ A\ B ===> C)\ (\mathit{case\text{-}prod}\ f)\ (\mathit{case\text{-}prod}\ g)$
⟨*proof*⟩

lemma (**in** *ccpo*) *Sup-image-mono*:
assumes *ccpo*: *class.ccpo luba orda lessa*
and *mono*: *monotone orda* $(\leq)$ *f*
and *chain*: *Complete-Partial-Order.chain orda A*
and $A \neq \{\}$
shows $\mathit{Sup}\ (f\ `\ A) \leq (f\ (\mathit{luba}\ A))$
⟨*proof*⟩

lemma (**in** *ccpo*) *admissible-le-mono*:
assumes *monotone* $(\leq)$ $(\leq)$ *f*
shows $\mathit{ccpo.admissible}\ \mathit{Sup}\ (\leq)\ (\lambda x.\ x \leq f\ x)$
⟨*proof*⟩

lemma (**in** *ccpo*) *fixp-induct-strong2*:
assumes *adm*: *ccpo.admissible Sup* $(\leq)$ *P*
and *mono*: *monotone* $(\leq)$ $(\leq)$ *f*
and *bot*: $P\ (\bigsqcup \{\})$
and *step*: $\bigwedge x.\ [\![\ x \leq \mathit{ccpo\text{-}class.fixp}\ f;\ x \leq f\ x;\ P\ x\ ]\!] \Longrightarrow P\ (f\ x)$
shows $P\ (\mathit{ccpo\text{-}class.fixp}\ f)$
⟨*proof*⟩

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

lemma *fixp-induct-strong2-uc*:
fixes $F :: 'c \Rightarrow 'c$
and $U :: 'c \Rightarrow 'b \Rightarrow 'a$
and $C :: ('b \Rightarrow 'a) \Rightarrow 'c$
and $P :: ('b \Rightarrow 'a) \Rightarrow \mathit{bool}$
assumes *mono*: $\bigwedge x.\ \mathit{mono\text{-}body}\ (\lambda f.\ U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C\ (\mathit{fixp\text{-}fun}\ (\lambda f.\ U\ (F\ (C\ f))))$
and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
and *adm*: *ccpo.admissible lub-fun le-fun P*
and *bot*: $P\ (\lambda\text{-}.\ \mathit{lub}\ \{\})$
and *step*: $\bigwedge f'.\ [\![\ \mathit{le\text{-}fun}\ (U\ f')\ (U\ f);\ \mathit{le\text{-}fun}\ (U\ f')\ (U\ (F\ f'));\ P\ (U\ f')\ ]\!] \Longrightarrow P\ (U\ (F\ f'))$
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 $P{=}\lambda f\ g.\ P\ (\mathit{curry}\ f)\ (\mathit{curry}\ (\mathit{curry}\ (\mathit{curry}\ g)))$,
unfolded case-prod-curry curry-case-prod curry-K,

OF - - - - - - refl refl]
for *P*

lemma (**in** *ccpo*) *fixp-greatest*:
assumes *f*: *monotone* (≤) (≤) *f*
and *ge*: ⋀*y*. *f y* ≤ *y* ⟹ *x* ≤ *y*
shows *x* ≤ *ccpo.fixp Sup* (≤) *f*
⟨*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* (λ*x*. *g* (*f x*)) = *g* (*ccpo.fixp lub2 leq2* (λ*x*. *f* (*g x*)))
⟨*proof*⟩

lemma *fixp-lfp-parametric-eq*:
includes *lifting-syntax*
assumes *f*: ⋀*x*. *lfp.mono-body* (λ*f*. *F f x*)
and *g*: ⋀*x*. *lfp.mono-body* (λ*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*: $\bigwedge x\ s\ A'.\ [\![\ x \in A';\ I\ A'\ s;\ A' \subseteq A\ ]\!] \Longrightarrow I\ (A' - \{x\})\ (f\ x\ s)$
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*: $\bigwedge x\ s\ A'.\ [\![\ I\ A'\ s;\ x \notin A';\ x \in A;\ A' \subseteq A\ ]\!] \Longrightarrow I\ (\mathit{insert}\ x\ A')\ (f\ x\ s)$
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* $(A \cup B)$ = *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*⟩

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* ∈ *nlists A n* ⟷ (∃ *n'*. *n* = *Suc n'* ∧ *x* ∈ *A* ∧ *xs* ∈ *nlists A n'*)
⟨*proof*⟩

lemma *in-nlists-Suc-iff*: *xs* ∈ *nlists A* (*Suc n*) ⟷ (∃ *x xs'*. *xs* = *x* # *xs'* ∧ *x* ∈ *A* ∧ *xs'* ∈ *nlists A n*)
⟨*proof*⟩

lemma *nlists-Suc*: *nlists A* (*Suc n*) = (⋃*x*∈*A*. (#) *x* ‘ *nlists A n*)
⟨*proof*⟩

lemma *replicate-in-nlists* [*simp*, *intro*]: *x* ∈ *A* ⟹ *replicate n x* ∈ *nlists A n*
⟨*proof*⟩

lemma *nlists-eq-empty-iff* [*simp*]: *nlists A n* = {} ⟷ *n* > *0* ∧ *A* = {}
⟨*proof*⟩

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

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

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

lemma *card-nlists*: *card* (*nlists A n*) = *card A* ^ *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* $\wedge$ *sprefix xs* (*stl ys*)

lemma *sprefix-append*: *sprefix* (*xs* @ *ys*) *zs* $\longleftrightarrow$ *sprefix xs zs* $\wedge$ *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 $\leq$ *length ys* $\Longrightarrow$ *sprefix xs* (*ys* @− *zs*) $\longleftrightarrow$ *prefix xs ys*
⟨*proof*⟩

lemma *prefixeq-stake2* [*simp*]: *prefix xs* (*stake n ys*) $\longleftrightarrow$ *length xs* $\leq$ *n* $\wedge$ *sprefix xs ys*
⟨*proof*⟩

lemma *tlength-eq-infinity-iff*: *tlength xs* = ∞ $\longleftrightarrow$ $\neg$ *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* ∘ *f*)
⟨*proof*⟩

end

lemma *bind-option-set* [*simp*]: *monad.bind-option fail-set* = (λ*x*. *UNION* (*set-option x*))
⟨*proof*⟩

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

lemma *run-bind-option-envT* [*simp*]:
⋀*more*. *run-env* (*monad.bind-option* (*fail-env fail*) *x f*) *s* =
monad.bind-option fail x (λ*y*. *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*]:
$A \subseteq B \Longrightarrow snd \in measurable\ (M1 \bigotimes_M count\text{-}space\ A)\ (count\text{-}space\ B)$
⟨*proof*⟩

1.19 Sequence space

lemma (**in** *sequence-space*) *nn-integral-split*:
assumes *f*[*measurable*]: $f \in borel\text{-}measurable\ S$
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*:
$measure\text{-}pmf\ (map\text{-}pmf\ f\ p) = distr\ (measure\text{-}pmf\ p)\ (count\text{-}space\ UNIV)\ f$
⟨*proof*⟩

abbreviation *coin-pmf* :: *bool pmf* **where** $coin\text{-}pmf \equiv pmf\text{-}of\text{-}set\ UNIV$

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

notepad begin
⟨*proof*⟩
end

lemma *pred-rel-pmf*:
$[\![\ pred\text{-}pmf\ P\ p;\ rel\text{-}pmf\ R\ p\ q\ ]\!] \Longrightarrow pred\text{-}pmf\ (Imagep\ R\ P)\ q$
⟨*proof*⟩

lemma *pmf-rel-mono'*: $[\![\ rel\text{-}pmf\ P\ x\ y;\ P \le Q\ ]\!] \Longrightarrow rel\text{-}pmf\ Q\ x\ y$
⟨*proof*⟩

lemma *rel-pmf-eqI* [*simp*]: $rel\text{-}pmf\ (=)\ x\ x$
⟨*proof*⟩

lemma *rel-pmf-bind-reflI*:
$(\bigwedge x.\ x \in set\text{-}pmf\ p \Longrightarrow rel\text{-}pmf\ R\ (f\ x)\ (g\ x))$

$\Longrightarrow$ *rel-pmf R* (*bind-pmf p f*) (*bind-pmf p g*)
⟨*proof*⟩

lemma *pmf-pred-mono-strong*:
⟦ *pred-pmf P p*; $\bigwedge$*a*. ⟦ *a* ∈ *set-pmf p*; *P a* ⟧ $\Longrightarrow$ *P' a* ⟧ $\Longrightarrow$ *pred-pmf P' p*
⟨*proof*⟩

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

lemma *rel-pmf-restrict-relpE* [*elim?*]:
assumes *rel-pmf* (*R* ↾ *P* ⊗ *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* ⟧ $\Longrightarrow$ *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* ≠ {} ⟧ $\Longrightarrow$ *pred-pmf P* (*pmf-of-set A*) = *Ball A P*
⟨*proof*⟩

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

lemma *pred-pmf-cond* [*simp*]:
set-pmf p ∩ *A* ≠ {} $\Longrightarrow$ *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*⟩

1.21 Subprobability mass functions

lemma *ord-spmf-return-spmf1*: *ord-spmf R* (*return-spmf x*) *p* ⟷ *lossless-spmf p* ∧ (∀ *y*∈*set-spmf p*. *R x y*)
⟨*proof*⟩

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

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

lemma *ord-spmf-eqD-measure*: *ord-spmf* (=) *p q* ⟹ *measure* (*measure-spmf p*) *A* ≤ *measure* (*measure-spmf q*) *A*
⟨*proof*⟩

lemma *ord-spmf-measureD*:
assumes *ord-spmf R p q*
shows *measure* (*measure-spmf p*) *A* ≤ *measure* (*measure-spmf q*) {*y*. ∃ *x*∈*A*. *R x y*}
(**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

lemma *ord-spmf-bind-pmfI1*:
(⋀*x*. *x* ∈ *set-pmf p* ⟹ *ord-spmf R* (*f x*) *q*) ⟹ *ord-spmf R* (*bind-pmf p f*) *q*
⟨*proof*⟩

lemma *ord-spmf-bind-spmfI1*:
(⋀*x*. *x* ∈ *set-spmf p* ⟹ *ord-spmf R* (*f x*) *q*) ⟹ *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*: ⋀*X*. *X* ⊆ *A* ⟹ *card B* ∗ *card X* ≤ *card A* ∗ *card* {*y*∈*B*. ∃ *x*∈*X*. *R x y*}
and *eq*: (*finite A* ∧ *A* ≠ {}) ⟷ (*finite B* ∧ *B* ≠ {})
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* ∈ *borel-measurable* (*count-space UNIV*)
shows *nn-integral* (*measure-spmf p*) *f* = *nn-integral* (*restrict-space* (*measure-pmf p*) (*range Some*)) (*f* ∘ *the*)
⟨*proof*⟩

lemma *nn-integral-spmf-neq-infinity*: (∫⁺ *x*. *spmf p x* ∂*count-space UNIV*) ≠ ∞
⟨*proof*⟩

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

lemma *rel-spmf-pos-distr*: *rel-spmf A OO rel-spmf B* $\leq$ *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* $\longleftrightarrow$ *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* $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *B x y* ⟧ $\Longrightarrow$ *rel-spmf B f g*
⟨*proof*⟩

lemma *set-spmf-eq-empty*: *set-spmf p* = {} $\longleftrightarrow$ *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* $\Longrightarrow$ *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* ≠ {} ⟧ $\Longrightarrow$ *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* $(R \upharpoonright P \otimes Q)$ 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* $(\lambda x.$ *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 ∧ (¬ lossless-spmf p ⟶ pred-spmf P q))
⟨*proof*⟩

lemma *pred-spmf-cond* [*simp*]:
pred-spmf P (cond-spmf p A) = pred-spmf (λx. x ∈ A ⟶ 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 ((λx. R x ∘ g) ↾ P ⊗ Q ∘ g) p q*
⟨*proof*⟩

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

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*) (*λx. 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 (λ*y*. *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* (λ*f*. *Set.bind x* (λ*x*. {*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*!]:
$[\![x \in f \diamond y;\ \bigwedge f'\ y'.\ [\![x = f'\ y';\ f' \in f;\ y' \in y]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow \mathit{thesis}$
⟨*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

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* (λ*f x y*. *f y x*) ⋄ *f* ⋄ *x* ⋄ *y* = *f* ⋄ *y* ⋄ *x*
⟨*proof*⟩

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

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

lemma *bind-ap-spmf*: *bind-spmf* (*p* ⋄ *x*) *f* = *bind-spmf p* (λ*p*. *x* ⤜ (λ*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* ∘ *map-option f*)
⟨*proof*⟩

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

end

end

1.24 Exclusive or on lists

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

definition *xor* :: *'a* ⇒ *'a* ⇒ *'a* :: {*uminus*,*inf*,*sup*} (**infixr** ⊕ *67*)
where *x* ⊕ *y* = *inf* (*sup x y*) (− (*inf x y*))

lemma *xor-bool-def* [*iff*]: **fixes** *x y* :: *bool* **shows** *x* ⊕ *y* ⟷ *x* ≠ *y*
⟨*proof*⟩

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

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

lemma *xor-left-commute*:

fixes $x\ y :: 'a :: boolean\text{-}algebra$
shows $x \oplus (y \oplus z) = y \oplus (x \oplus z)$
⟨*proof*⟩

lemma [*simp*]:
fixes $x :: 'a :: boolean\text{-}algebra$
shows *xor-bot*: $x \oplus bot = x$
and *bot-xor*: $bot \oplus x = x$
and *xor-top*: $x \oplus top = - x$
and *top-xor*: $top \oplus x = - x$
⟨*proof*⟩

lemma *xor-inverse* [*simp*]:
fixes $x :: 'a :: boolean\text{-}algebra$
shows $x \oplus x = bot$
⟨*proof*⟩

lemma *xor-left-inverse* [*simp*]:
fixes $x :: 'a :: boolean\text{-}algebra$
shows $x \oplus x \oplus y = y$
⟨*proof*⟩

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

definition *xor-list* :: $'a :: \{uminus, inf, sup\}\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ (**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* :: $'a :: \{semilattice\text{-}sup, semilattice\text{-}inf, uminus\}$ *list*
shows *xs* [⊕] *ys* = *ys* [⊕] *xs*
⟨*proof*⟩

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

lemma *xor-list-left-commute*:
fixes *xs ys zs* :: $'a :: boolean\text{-}algebra\ list$
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*]: $(f \diamond x)\ i = f\ i\ (x\ i)$
⟨*proof*⟩

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

lemma *all-envirI* [*Pure.intro*!, *intro*!]: $(\bigwedge x.\ p\ x) \Longrightarrow all\text{-}envir\ p$
⟨*proof*⟩

lemma *all-envirE* [*Pure.elim 2*, *elim*]: $all\text{-}envir\ p \Longrightarrow (p\ x \Longrightarrow thesis) \Longrightarrow thesis$
⟨*proof*⟩

lemma *all-envirD*: $all\text{-}envir\ p \Longrightarrow p\ x$
⟨*proof*⟩

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

lemma *pred-envir-conv*: $pred\text{-}envir\ p\ f \longleftrightarrow (\forall x.\ p\ (f\ x))$
⟨*proof*⟩

lemma *pred-envirI* [*Pure.intro*!, *intro*!]: $(\bigwedge x.\ p\ (f\ x)) \Longrightarrow pred\text{-}envir\ p\ f$
⟨*proof*⟩

lemma *pred-envirD*: $pred\text{-}envir\ p\ f \Longrightarrow p\ (f\ x)$
⟨*proof*⟩

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

lemma *pred-envir-mono*: $[\![\ pred\text{-}envir\ p\ f;\ \bigwedge x.\ p\ (f\ x) \Longrightarrow q\ (g\ x)\ ]\!] \Longrightarrow pred\text{-}envir\ q\ g$
⟨*proof*⟩

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

lemma *rel-envir-conv*: $rel\text{-}envir\ p\ f\ g \longleftrightarrow (\forall x.\ p\ (f\ x)\ (g\ x))$
⟨*proof*⟩

lemma *rel-envir-conv-rel-fun*: $rel\text{-}envir = rel\text{-}fun\ (=)$
⟨*proof*⟩

lemma *rel-envirI* [*Pure.intro*!, *intro*!]: $(\bigwedge x.\ p\ (f\ x)\ (g\ x)) \Longrightarrow rel\text{-}envir\ p\ f\ g$
⟨*proof*⟩

lemma *rel-envirD*: *rel-envir p f g* $\Longrightarrow$ *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*: $\bigwedge x.\ mono\text{-}set\ (\lambda f.\ U\ (F\ (C\ f))\ x)$
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*: $\bigwedge f\ x\ y.\ [\![\ \bigwedge x\ y.\ y \in U\ f\ x \Longrightarrow P\ x\ y;\ y \in U\ (F\ f)\ x\ ]\!] \Longrightarrow P\ x\ y$
and *enforce-variable-ordering*: $x = x$
and *elem*: $y \in U\ f\ x$
shows $P\ x\ y$
⟨*proof*⟩

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

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

lemma *monotone-le-le*: $monotone\ (\leq)\ (\leq) = mono$
⟨*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-set A* $\Longrightarrow$ *mono-set* ($\lambda f.\ g$ ‘ *A f*)
and *vimage-mono*: *mono-set A* $\Longrightarrow$ *mono-set* ($\lambda f.\ g$ −‘ *A f*)
⟨*proof*⟩

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

interpretation *coset*: *partial-function-definitions* (⊇) *Inter*
⟨*proof*⟩

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

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

lemma *coset-admissible*: *coset.admissible* (λf :: *'a* ⇒ *'b set*. $\forall x\ y.\ P\ x\ y \longrightarrow y \in f\ x$)
⟨*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*: $\bigwedge x.$ *mono-coset* ($\lambda f.\ U\ (F\ (C\ f))\ x$)
and *eq*: *f* ≡ *C* (*ccpo.fixp* (*fun-lub Inter*) (*fun-ord* (≥)) ($\lambda f.\ U\ (F\ (C\ f))$))
and *inverse2*: $\bigwedge f.\ U\ (C\ f) = f$

and *step*: $\bigwedge f'\ x\ y.\ [\![\ \bigwedge x.\ U\ f'\ x = U\ f'\ x;\ \neg\ P\ x\ y\ ]\!] \Longrightarrow y \in U\ (F\ (C\ (sup\ (\lambda x.\ \{y.\ \neg\ P\ x\ y\})\ (U\ f))))\ x$
— partial_function requires a quantifier over f', so let's have a fake one
and *elem*: $y \notin U\ f\ x$
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′* (λ*f*. ⋃*y*∈*B f*. *C y f*)
and *set-bind-mono′*: ⟦*mono-set′ B*; ⋀*y*. *mono-set′* (λ*f*. *C y f*)⟧ ⟹ *mono-set′* (λ*f*. *Set.bind* (*B f*) (λ*y*. *C y f*))
and *Un-mono′*: ⟦ *mono-set′ A*; *mono-set′ B* ⟧ ⟹ *mono-set′* (λ*f*. *A f* ∪ *B f*)
and *Int-mono′*: ⟦ *mono-set′ A*; *mono-set′ B* ⟧ ⟹ *mono-set′* (λ*f*. *A f* ∩ *B f*)
⟨*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 ∗: ⋀*x y*. *P x y* ⟹ *y* = *x* ∨ (*P* (*Suc x*) *y* ∨ *y* ∈ *test2* (*Suc x*))
shows *y* ∈ *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* ⟷ (∀ *c*>*0*. *f* ∈ *o*(λ*x*. *inverse* (*x powr c*)))

lemma *negligibleI* [*intro?*]:
(⋀*c*. *c* > *0* ⟹ *f* ∈ *o*(λ*x*. *inverse* (*x powr c*))) ⟹ *negligible f*
⟨*proof*⟩

lemma *negligibleD*:
⟦ *negligible f*; *c* > *0* ⟧ ⟹ *f* ∈ *o*(λ*x*. *inverse* (*x powr c*))
⟨*proof*⟩

lemma *negligibleD-real*:
assumes *negligible f*
shows *f* ∈ *o*(λ*x*. *inverse* (*x powr c*))

⟨*proof*⟩

lemma *negligible-mono*: ⟦ *negligible g*; *f* ∈ *O*(*g*) ⟧ ⟹ *negligible f*
⟨*proof*⟩

lemma *negligible-le*: ⟦ *negligible g*; ⋀η. |*f* η| ≤ *g* η ⟧ ⟹ *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* ⟧ ⟹ *negligible* (λη. *f* η + *g* η)
⟨*proof*⟩

lemma *negligible-uminus* [*simp*]: *negligible* (λη. − *f* η) ⟷ *negligible f*
⟨*proof*⟩

lemma *negligible-uminusI* [*negligible-intros*]: *negligible f* ⟹ *negligible* (λη. − *f* η)
⟨*proof*⟩

lemma *negligible-minus* [*negligible-intros*]:
⟦ *negligible f*; *negligible g* ⟧ ⟹ *negligible* (λη. *f* η − *g* η)
⟨*proof*⟩

lemma *negligible-cmult*: *negligible* (λη. *c* ∗ *f* η) ⟷ *negligible f* ∨ *c* = *0*
⟨*proof*⟩

lemma *negligible-cmultI* [*negligible-intros*]:
(*c* ≠ *0* ⟹ *negligible f*) ⟹ *negligible* (λη. *c* ∗ *f* η)
⟨*proof*⟩

lemma *negligible-multc*: *negligible* (λη. *f* η ∗ *c*) ⟷ *negligible f* ∨ *c* = *0*
⟨*proof*⟩

lemma *negligible-multcI* [*negligible-intros*]:
(*c* ≠ *0* ⟹ *negligible f*) ⟹ *negligible* (λη. *f* η ∗ *c*)
⟨*proof*⟩

lemma *negligible-times* [*negligible-intros*]:
  assumes *f*: *negligible f*
  and *g*: *negligible g*
  shows *negligible* (λη. *f* η ∗ *g* η :: *real*)
⟨*proof*⟩

lemma *negligible-power* [*negligible-intros*]:
  assumes *negligible f*
  and *n* > *0*
  shows *negligible* (λη. *f* η ^ *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*|) ⟷ *negligible f*
⟨*proof*⟩

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

lemma *negligible-powrI* [*negligible-intros*]:
  assumes *0* ≤ *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* ⟹ *negligible* (λη. *1* / *k* ^ η)
⟨*proof*⟩

inductive *polynomial* :: (*nat* ⇒ *real*) ⇒ *bool*
  for *f*
where *f* ∈ *O*(λx. *x powr n*) ⟹ *polynomial f*

lemma *negligible-times-poly*:
  assumes *f*: *negligible f*
  and *g*: *g* ∈ *O*(λx. *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 ⟧ ⟹ *negligible* $(\lambda x.\ f\ x * g\ x)$
⟨*proof*⟩

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

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

lemma *negligible-divide-poly1*:
⟦ $f \in O(\lambda x.\ x\ powr\ n)$; *negligible* $(\lambda \eta.\ 1\ /\ g\ \eta)$ ⟧ ⟹ *negligible* $(\lambda \eta.\ real\ (f\ \eta)\ /\ g\ \eta)$
⟨*proof*⟩

lemma *negligible-divide-polynomial1* [*negligible-intros*]:
⟦ *polynomial* f; *negligible* $(\lambda \eta.\ 1\ /\ g\ \eta)$ ⟧ ⟹ *negligible* $(\lambda \eta.\ real\ (f\ \eta)\ /\ g\ \eta)$
⟨*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*: $'in \Rightarrow (('a, 'out, 'in)\ resumption)$
where
resume (*Done a*) = $(\lambda inp.\ Done\ None)$

code-datatype *Done Pause*

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

x))) *inp else bind-resumption* (*resume x inp*) *f*)

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

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

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

lemma *result-bind-resumption* [*simp*]:
is-Done (*x* ⤜ *f*) ⟹ *result* (*x* ⤜ *f*) = *result x* ⤜ *result* ∘ *f*
⟨*proof*⟩

lemma *output-bind-resumption* [*simp*]:
¬ *is-Done* (*x* ⤜ *f*) ⟹ *output* (*x* ⤜ *f*) = (*if is-Done x then output* (*f* (*the* (*result x*))) *else output x*)
⟨*proof*⟩

lemma *resume-bind-resumption* [*simp*]:
¬ *is-Done* (*x* ⤜ *f*) ⟹
resume (*x* ⤜ *f*) =
(*if is-Done x then resume* (*f* (*the* (*result x*)))
else (λ*inp*. *resume x inp* ⤜ *f*))
⟨*proof*⟩

definition *DONE* :: '*a* ⇒ ('*a*, '*out*, '*in*) *resumption*
where *DONE* = *Done* ∘ *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* ⟷ *a* = *b*
and *DONE-neq-ABORT*: *DONE a* ≠ *ABORT*
and *ABORT-neq-DONE*: *ABORT* ≠ *DONE a*
and *ABORT-eq-Done*: ⋀*a*. *ABORT* = *Done a* ⟷ *a* = *None*
and *Done-eq-ABORT*: ⋀*a*. *Done a* = *ABORT* ⟷ *a* = *None*
and *DONE-eq-Done*: ⋀*b*. *DONE a* = *Done b* ⟷ *b* = *Some a*
and *Done-eq-DONE*: ⋀*b*. *Done b* = *DONE a* ⟷ *b* = *Some a*
and *DONE-neq-Pause*: *DONE a* ≠ *Pause out c*
and *Pause-neq-DONE*: *Pause out c* ≠ *DONE a*
and *ABORT-neq-Pause*: *ABORT* ≠ *Pause out c*
and *Pause-neq-ABORT*: *Pause out c* ≠ *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* $\gg\!=$ *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*:
⟦ *rel-resumption A B r1 r2*; *is-Done r1* ⟧ $\Longrightarrow$ *rel-option A* (*result r1*) (*result r2*)
⟨*proof*⟩

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

lemma *rel-resumption-outputD1*:
⟦ *rel-resumption A B r1 r2*; $\neg$ *is-Done r1* ⟧ $\Longrightarrow$ *B* (*output r1*) (*output r2*)
⟨*proof*⟩

lemma *rel-resumption-outputD2*:
⟦ *rel-resumption A B r1 r2*; $\neg$ *is-Done r2* ⟧ $\Longrightarrow$ *B* (*output r1*) (*output r2*)
⟨*proof*⟩

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

lemma *rel-resumption-resumeD2*:
⟦ *rel-resumption A B r1 r2*; $\neg$ *is-Done r2* ⟧
$\Longrightarrow$ *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*) $\wedge$ (*is-Done r1* $\longrightarrow$ *is-Done r2* $\longrightarrow$ *rel-option A* (*result r1*) (*result r2*))
and *Pause*: $\bigwedge$*r1 r2*. ⟦ *X r1 r2*; $\neg$ *is-Done r1*; $\neg$ *is-Done r2* ⟧ $\Longrightarrow$ *B* (*output r1*) (*output r2*) $\wedge$ ($\forall$ *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'* ⟹ *resumption-ord* (*Done a*) (*Done a'*)
| *Done-Pause*: *resumption-ord ABORT* (*Pause out c*)
| *Pause-Pause*: ((=) ===> *resumption-ord*) *c c'* ⟹ *resumption-ord* (*Pause out c*) (*Pause out c'*)

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

lemma *resumption-ord-is-DoneD*:
⟦ *resumption-ord r r'*; *is-Done r'* ⟧ ⟹ *is-Done r*
⟨*proof*⟩

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

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

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

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

lemma *resumption-ord-trans*:
⟦ *resumption-ord r r'*; *resumption-ord r' r''* ⟧
⟹ *resumption-ord r r''*
⟨*proof*⟩

primcorec *resumption-lub* :: ('*a*, '*out*, '*in*) *resumption set* ⇒ ('*a*, '*out*, '*in*) *resumption*
where
∀ *r* ∈ *R*. *is-Done r* ⟹ *is-Done* (*resumption-lub R*)
| *result* (*resumption-lub R*) = *flat-lub None* (*result* ‘ *R*)
| *output* (*resumption-lub R*) = (*THE out*. *out* ∈ *output* ‘ (*R* ∩ {*r*. ¬ *is-Done r*}))
| *resume* (*resumption-lub R*) = (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*) ⟷ (∀ *r* ∈ *R*. *is-Done r*)
⟨*proof*⟩

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

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

lemma *resume-resumption-lub* [*simp*]:
∃ *r*∈*R*. ¬ *is-Done r*
⟹ *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 $h \equiv \lambda r.$ *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*: $\bigwedge$*out. mcont* (*fun-lub resumption-lub*) (*fun-ord resumption-ord*) *lub ord* (λ*c. g out c* (*Pause out c*))
and *ccpo*: *class.ccpo lub ord* (*mk-less ord*)
and *bot*: $\bigwedge x.$ *ord* (*f None*) *x*
shows *mcont resumption-lub resumption-ord lub ord* ($\lambda r.$ *case r of Done x* $\Rightarrow$ *f x* | *Pause out c* $\Rightarrow$ *g out c r*)
(**is** *mcont ?lub ?ord - - ?f*)
⟨*proof*⟩

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

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

lemma fixes *f F*
defines $F \equiv \lambda$*outputs xs. case xs of resumption.Done x* $\Rightarrow$ {} | *resumption.Pause out c* $\Rightarrow$ *insert out* ($\bigcup$*input. outputs* (*c input*))
shows *outputs-conv-fixp*: *outputs* $\equiv$ *ccpo.fixp* (*fun-lub Union*) (*fun-ord* ($\subseteq$)) *F* (**is** - $\equiv$ *?fixp*)
and *outputs-mono*: $\bigwedge x.$ *monotone* (*fun-ord* ($\subseteq$)) ($\subseteq$) ($\lambda f.$ *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* ($\subseteq$) *outputs*
⟨*proof*⟩

lemma *mono2mono-outputs*[*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-outputs*: *monotone resumption-ord* ($\subseteq$) *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*: $out = out' \Longrightarrow IO\ out\ c = IO\ out'\ c$ ⟨*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* ⟹ *result* (*map-generat f g h x*) = *f* (*result x*)
⟨*proof*⟩

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

lemma *continuation-map-generat* [*simp*]: ¬ *is-Pure x* ⟹ *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* ⟷ (∃ *x'*. *generat* = *Pure x'* ∧ *x* = *f x'*)
  and *Pure-eq-map-generat*:
  Pure x = *map-generat f g h generat* ⟷ (∃ *x'*. *generat* = *Pure x'* ∧ *x* = *f x'*)
⟨*proof*⟩

lemma [*simp*]:
  shows *map-generat-eq-IO*:
  map-generat f g h generat = *IO out c* ⟷ (∃ *out' c'*. *generat* = *IO out' c'* ∧ *out* = *g out'* ∧ *c* = *h c'*)
  and *IO-eq-map-generat*:
  IO out c = *map-generat f g h generat* ⟷ (∃ *out' c'*. *generat* = *IO out' c'* ∧ *out* = *g out'* ∧ *c* = *h c'*)
⟨*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* ⟷ *is-Pure y*;
    ⟦ *is-Pure x*; *is-Pure y* ⟧ ⟹ *A* (*result x*) (*result y*);

$[\![\neg \mathit{is\text{-}Pure}\ x;\ \neg \mathit{is\text{-}Pure}\ y]\!] \Longrightarrow \mathit{Out}\ (\mathit{output}\ x)\ (\mathit{output}\ y) \wedge R\ (\mathit{continuation}\ x)\ (\mathit{continuation}\ y)]\!]$
$\Longrightarrow \mathit{rel\text{-}generat}\ A\ \mathit{Out}\ R\ x\ y$
⟨*proof*⟩

lemma *rel-generatD'*:
$\mathit{rel\text{-}generat}\ A\ \mathit{Out}\ R\ x\ y$
$\Longrightarrow (\mathit{is\text{-}Pure}\ x \longleftrightarrow \mathit{is\text{-}Pure}\ y) \wedge$
$(\mathit{is\text{-}Pure}\ x \longrightarrow \mathit{is\text{-}Pure}\ y \longrightarrow A\ (\mathit{result}\ x)\ (\mathit{result}\ y)) \wedge$
$(\neg \mathit{is\text{-}Pure}\ x \longrightarrow \neg \mathit{is\text{-}Pure}\ y \longrightarrow \mathit{Out}\ (\mathit{output}\ x)\ (\mathit{output}\ y) \wedge R\ (\mathit{continuation}\ x)\ (\mathit{continuation}\ y))$
⟨*proof*⟩

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

lemma *rel-generat-mono*:
$[\![\mathit{rel\text{-}generat}\ A\ B\ C\ x\ y;\ \bigwedge x\ y.\ A\ x\ y \Longrightarrow A'\ x\ y;\ \bigwedge x\ y.\ B\ x\ y \Longrightarrow B'\ x\ y;\ \bigwedge x\ y.\ C\ x\ y \Longrightarrow C'\ x\ y]\!]$
$\Longrightarrow \mathit{rel\text{-}generat}\ A'\ B'\ C'\ x\ y$
⟨*proof*⟩

lemma *rel-generat-mono'* [*mono*]:
$[\![\bigwedge x\ y.\ A\ x\ y \longrightarrow A'\ x\ y;\ \bigwedge x\ y.\ B\ x\ y \longrightarrow B'\ x\ y;\ \bigwedge x\ y.\ C\ x\ y \longrightarrow C'\ x\ y]\!]$
$\Longrightarrow \mathit{rel\text{-}generat}\ A\ B\ C\ x\ y \longrightarrow \mathit{rel\text{-}generat}\ A'\ B'\ C'\ x\ y$
⟨*proof*⟩

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

lemma *rel-generat-reflI*:
$[\![\bigwedge y.\ y \in \mathit{generat\text{-}pures}\ x \Longrightarrow A\ y\ y;$
$\bigwedge \mathit{out}.\ \mathit{out} \in \mathit{generat\text{-}outs}\ x \Longrightarrow B\ \mathit{out}\ \mathit{out};$
$\bigwedge \mathit{cont}.\ \mathit{cont} \in \mathit{generat\text{-}conts}\ x \Longrightarrow C\ \mathit{cont}\ \mathit{cont}]\!]$
$\Longrightarrow \mathit{rel\text{-}generat}\ A\ B\ C\ x\ x$
⟨*proof*⟩

lemma *reflp-rel-generat* [*simp*]: $\mathit{reflp}\ (\mathit{rel\text{-}generat}\ A\ B\ C) \longleftrightarrow \mathit{reflp}\ A \wedge \mathit{reflp}\ B$

$\wedge$ *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*) = (λ*r*. ∃ *y*. *r* = *Pure y* ∧ *A x y*)
⟨*proof*⟩

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

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* ∈ *results'-gpv gpv*
obtains (*Pure*) *generat* **where** *generat* ∈ *set-spmf* (*the-gpv gpv*) *x* ∈ *generat-pures generat*
| (*Cont*) *generat c input* **where** *generat* ∈ *set-spmf* (*the-gpv gpv*) *c* ∈ *generat-conts generat x* ∈ *results'-gpv* (*c input*)
⟨*proof*⟩

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

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

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

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

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

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

lemma *bi-total-rel-gpv* [*transfer-rule*]: ⟦ *bi-total A*; *bi-total B* ⟧ ⟹ *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*; ⋀*a*. ⟦ *a* ∈ *results'-gpv x*; *P a* ⟧ ⟹ *P' a*; ⋀*b*. ⟦ *b* ∈ *outs'-gpv x*; *Q b* ⟧ ⟹ *Q' b* ⟧ ⟹ *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* ⟧ ⟹ *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* ≤ *A'* *C* ≤ *C'* *R'* ≤ *R*
  shows *rel-gpv''* *A* *C* *R* ≤ *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'* ≤ *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''-neg-distr*:
  assumes *R*: *left-unique* *R* *right-total* *R*

and R': *right-unique* R' *left-total* R'
shows *rel-gpv''* $(A\ OO\ A')\ (C\ OO\ C')\ (R\ OO\ R') \leq$ *rel-gpv''* $A\ C\ R\ OO$ *rel-gpv''* $A'\ C'\ R'$
⟨*proof*⟩

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* $\{x.\ (x \circ f)\ `\ UNIV \subseteq B\}$ (*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 *rel-gpv''-Grp*:
rel-gpv'' (*BNF-Def.Grp* $A\ f$) (*BNF-Def.Grp* $B\ g$) $(BNF\text{-}Def.Grp\ UNIV\ h)^{-1-1}$ =
BNF-Def.Grp $\{x.$ *results'-gpv* $x \subseteq A \wedge$ *outs'-gpv* $x \subseteq B\}$ (*map-gpv'* $f\ g\ h$)
(**is** *?lhs* = *?rhs*)
⟨*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* ⇒ (*'a*, *'out*, *'in*) *gpv*
where *the-gpv* (*Done a*) = *return-spmf* (*Pure a*)

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

primcorec *lift-spmf* :: *'a spmf* ⇒ (*'a*, *'out*, *'in*) *gpv*
where *the-gpv* (*lift-spmf p*) = *map-spmf Pure p*

definition *Fail* :: (*'a*, *'out*, *'in*) *gpv*
where *Fail* = *GPV* (*return-pmf None*)

definition *React* :: (*'in* ⇒ *'out* × (*'a*, *'out*, *'in*) *rpv*) ⇒ (*'a*, *'out*, *'in*) *rpv*
where *React f input* = *case-prod Pause* (*f input*)

definition *rFail* :: (*'a*, *'out*, *'in*) *rpv*
where *rFail* = (λ-. *Fail*)

lemma *Done-inject* [*simp*]: *Done x* = *Done y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *Pause-inject* [*simp*]: *Pause out c* = *Pause out' c'* ⟷ *out* = *out'* ∧ *c* = *c'*
⟨*proof*⟩

lemma [*simp*]:
shows *Done-neq-Pause*: *Done x* ≠ *Pause out c*
and *Pause-neq-Done*: *Pause out c* ≠ *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* (⋃ *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* ∧ *All* (*pred-gpv P Q* ∘ *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*: $f = f' \Longrightarrow$ *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*
(λ*f*. *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*: $\bigwedge f\ g.\ X\ f\ g \Longrightarrow$ *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 $\Longrightarrow$ *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* ⟧ $\Longrightarrow$ *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

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 $*$: $\bigwedge$*gpv* n *out* c *input*. ⟦ X *gpv* n; *IO out* $c \in$ *set-spmf* (*the-gpv gpv*) ⟧
$\Longrightarrow \exists n'.$ (X (c *input*) $n' \vee$ *interaction-any-bound* (c *input*) $\leq n'$) $\wedge$ *eSuc* $n' \leq n$
shows *interaction-any-bound gpv* $\leq n$
⟨*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* $\leq n$ ⟧ $\Longrightarrow$ *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*; $n \leq m$ ⟧ $\Longrightarrow$ *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 ⋀*out c*. ⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *consider out* ⟧ ⟹ *n* ≠ *0* ∧ (∀ *input*. *interaction-bounded-by consider* (*c input*) (*n* − *1*))
and ⋀*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*)

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*) *I* = *UNIV* :: ('*call* ⇒ '*ret set*) *set* ⟨*proof*⟩

setup-lifting *type-definition-I*

lemma *outs-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-I* :: ('*call*, '*ret*) *I* ⇒ '*call set* **is** λ*resps*. {*out*. *resps out* ≠ {}}
parametric *outs-I-tparametric* ⟨*proof*⟩
lift-definition *responses-I* :: ('*call*, '*ret*) *I* ⇒ '*call* ⇒ '*ret set* **is** λ*x*. *x* **parametric** *id-transfer*[*unfolded id-def*] ⟨*proof*⟩

lift-definition *rel-I* :: ('*call* ⇒ '*call'* ⇒ *bool*) ⇒ ('*ret* ⇒ '*ret'* ⇒ *bool*) ⇒ ('*call*, '*ret*) *I* ⇒ ('*call'*, '*ret'*) *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-II* [*intro?*]:
  ⟦ *rel-set C* (*outs-I I1*) (*outs-I I2*); ⋀*x y*. *C x y* ⟹ *rel-set R* (*responses-I I1 x*) (*responses-I I2 y*) ⟧
  ⟹ *rel-I C R I1 I2*
⟨*proof*⟩

lemma *rel-I-eq* [*relator-eq*]: *rel-I* (=) (=) = (=)
⟨*proof*⟩

lemma *rel-I-conversep* [*simp*]: *rel-I* C^{-1-1} R^{-1-1} = (*rel-I C R*)$^{-1-1}$

⟨*proof*⟩

lemma *rel-ℐ-conversep1-eq* [*simp*]: *rel-ℐ* C^{-1-1} (=) = (*rel-ℐ* *C* (=))$^{-1-1}$
⟨*proof*⟩

lemma *rel-ℐ-conversep2-eq* [*simp*]: *rel-ℐ* (=) R^{-1-1} = (*rel-ℐ* (=) *R*)$^{-1-1}$
⟨*proof*⟩

lemma *responses-ℐ-empty-iff*: *responses-ℐ* ℐ *out* = {} ⟷ *out* ∉ *outs-ℐ* ℐ
including *ℐ.lifting* ⟨*proof*⟩

lemma *in-outs-ℐ-iff-responses-ℐ*: *out* ∈ *outs-ℐ* ℐ ⟷ *responses-ℐ* ℐ *out* ≠ {}
⟨*proof*⟩

lift-definition *ℐ-full* :: ('*call*, '*ret*) *ℐ* **is** λ-. *UNIV* ⟨*proof*⟩

lemma *ℐ-full-sel* [*simp*]:
  shows *outs-ℐ-full*: *outs-ℐ* *ℐ-full* = *UNIV*
  and *responses-ℐ-full*: *responses-ℐ* *ℐ-full* *x* = *UNIV*
⟨*proof*⟩

context includes *lifting-syntax* **begin**
lemma *outs-ℐ-parametric* [*transfer-rule*]: (*rel-ℐ* *C* *R* ===> *rel-set* *C*) *outs-ℐ* *outs-ℐ*
⟨*proof*⟩

lemma *responses-ℐ-parametric* [*transfer-rule*]:
  (*rel-ℐ* *C* *R* ===> *C* ===> *rel-set* *R*) *responses-ℐ* *responses-ℐ*
⟨*proof*⟩

end

definition *ℐ-trivial* :: ('*out*, '*in*) *ℐ* ⇒ *bool*
where *ℐ-trivial* ℐ ⟷ *outs-ℐ* ℐ = *UNIV*

lemma *ℐ-trivialI* [*intro?*]: (⋀*x*. *x* ∈ *outs-ℐ* ℐ) ⟹ *ℐ-trivial* ℐ
⟨*proof*⟩

lemma *ℐ-trivialD*: *ℐ-trivial* ℐ ⟹ *outs-ℐ* ℐ = *UNIV*
⟨*proof*⟩

lemma *ℐ-trivial-ℐ-full* [*simp*]: *ℐ-trivial* *ℐ-full*
⟨*proof*⟩

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

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-*$\mathcal{I}$ 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-*$\mathcal{I}$*-full*: *results-gpv* $\mathcal{I}$*-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* (λ*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*) = (⋃ *y*∈*set-option x*. *results'-gpv* (*f y*))
⟨*proof*⟩

lemma *bind-gpv-bind-option-assoc*:
  bind-gpv (*monad.bind-option Fail x f*) *g* = *monad.bind-option Fail x* (λ*x*. *bind-gpv* (*f x*) *g*)
⟨*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
  (⋀*out c*. *IO out c* ∈ *set-spmf gpv* ⟹ *out* ∈ *outs-*$\mathcal{I}$ Γ ∧ (∀ *input*∈*responses-*$\mathcal{I}$ Γ *out*. Γ ⊢*g c input* √))
  ⟹ 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*: ⋀*gpv out c*.
    ⟦ *X gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟧
    ⟹ *out* ∈ *outs-*$\mathcal{I}$ Γ ∧ (∀ *input* ∈ *responses-*$\mathcal{I}$ Γ *out*. *X* (*c input*) ∨ Γ ⊢*g c input*

$\surd$)
shows $\Gamma \vdash g$ *gpv* $\surd$
⟨*proof*⟩

lemma *WT-gpv-simps*:
$\Gamma \vdash g$ *GPV gpv* $\surd \longleftrightarrow$
($\forall$ *out c*. *IO out c* ∈ *set-spmf gpv* $\longrightarrow$ *out* ∈ *outs-*$\mathcal{I}$ $\Gamma \wedge (\forall$ *input*∈*responses-*$\mathcal{I}$ Γ *out*. $\Gamma \vdash g$ *c input* $\surd$))
⟨*proof*⟩

lemma *WT-gpvI*:
($\bigwedge$*out c*. *IO out c* ∈ *set-spmf* (*the-gpv gpv*) $\Longrightarrow$ *out* ∈ *outs-*$\mathcal{I}$ $\Gamma \wedge (\forall$ *input*∈*responses-*$\mathcal{I}$ Γ *out*. $\Gamma \vdash g$ *c input* $\surd$))
$\Longrightarrow \Gamma \vdash g$ *gpv* $\surd$
⟨*proof*⟩

lemma *WT-gpvD*:
assumes $\Gamma \vdash g$ *gpv* $\surd$
shows *WT-gpv-OutD*: *IO out c* ∈ *set-spmf* (*the-gpv gpv*) $\Longrightarrow$ *out* ∈ *outs-*$\mathcal{I}$ Γ
and *WT-gpv-ContD*: ⟦ *IO out c* ∈ *set-spmf* (*the-gpv gpv*); *input* ∈ *responses-*$\mathcal{I}$ Γ *out* ⟧ $\Longrightarrow \Gamma \vdash g$ *c input* $\surd$
⟨*proof*⟩

lemma *WT-gpv-mono*:
assumes *WT*: $\mathcal{I}1 \vdash g$ *gpv* $\surd$
and *outs*: *outs-*$\mathcal{I}$ $\mathcal{I}1$ ⊆ *outs-*$\mathcal{I}$ $\mathcal{I}2$
and *responses*: $\bigwedge x$. *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}1 \Longrightarrow$ *responses-*$\mathcal{I}$ $\mathcal{I}2$ *x* ⊆ *responses-*$\mathcal{I}$ $\mathcal{I}1$ *x*
shows $\mathcal{I}2 \vdash g$ *gpv* $\surd$
⟨*proof*⟩

lemma *WT-gpv-Done* [*iff*]: $\Gamma \vdash g$ *Done x* $\surd$
⟨*proof*⟩

lemma *WT-gpv-Fail* [*iff*]: $\Gamma \vdash g$ *Fail* $\surd$
⟨*proof*⟩

lemma *WT-gpv-PauseI*:
⟦ *out* ∈ *outs-*$\mathcal{I}$ Γ; $\bigwedge$*input*. *input* ∈ *responses-*$\mathcal{I}$ Γ *out* $\Longrightarrow \Gamma \vdash g$ *c input* $\surd$ ⟧
$\Longrightarrow \Gamma \vdash g$ *Pause out c* $\surd$
⟨*proof*⟩

lemma *WT-gpv-Pause* [*iff*]:
$\Gamma \vdash g$ *Pause out c* $\surd \longleftrightarrow$ *out* ∈ *outs-*$\mathcal{I}$ $\Gamma \wedge (\forall$ *input* ∈ *responses-*$\mathcal{I}$ Γ *out*. $\Gamma \vdash g$ *c input* $\surd$)
⟨*proof*⟩

lemma *WT-gpv-bindI*:
⟦ $\Gamma \vdash g$ *gpv* $\surd$; $\bigwedge x$. *x* ∈ *results-gpv* Γ *gpv* $\Longrightarrow \Gamma \vdash g$ *f x* $\surd$ ⟧
$\Longrightarrow \Gamma \vdash g$ *gpv* $\gg\!=$ *f* $\surd$

⟨*proof*⟩

lemma *WT-gpv-bindD2*:
assumes *WT*: Γ ⊢*g gpv* ⤜ *f* √
and *x*: *x* ∈ *results-gpv* Γ *gpv*
shows Γ ⊢*g f x* √
⟨*proof*⟩

lemma *WT-gpv-bindD1*: Γ ⊢*g gpv* ⤜ *f* √ ⟹ Γ ⊢*g gpv* √
⟨*proof*⟩

lemma *WT-gpv-bind* [*simp*]: Γ ⊢*g gpv* ⤜ *f* √ ⟷ Γ ⊢*g gpv* √ ∧ (∀ *x*∈*results-gpv* Γ *gpv*. Γ ⊢*g f x* √)
⟨*proof*⟩

lemma *WT-gpv-full* [*simp*, *intro*!]: *I-full* ⊢*g gpv* √
⟨*proof*⟩

lemma *WT-gpv-lift-spmf* [*simp*, *intro*!]: *I* ⊢*g lift-spmf p* √
⟨*proof*⟩

lemma *WT-gpv-coinduct-bind* [*consumes 1*, *case-names WT-gpv*, *case-conclusion WT-gpv out cont*]:
assumes ∗: *X gpv*
and *step*: ⋀*gpv out c*. ⟦ *X gpv*; *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟧
⟹ *out* ∈ *outs-I I* ∧ (∀ *input*∈*responses-I I out*.
X (*c input*) ∨
I ⊢*g c input* √ ∨
(∃ (*gpv'* :: (*'b*, *'call*, *'ret*) *gpv*) *f*. *c input* = *gpv'* ⤜ *f* ∧ *I* ⊢*g gpv'* √ ∧ (∀ *x*∈*results-gpv I gpv'*. *X* (*f x*))))
shows *I* ⊢*g gpv* √
⟨*proof*⟩

lemma *I-trivial-WT-gpvD* [*simp*]: *I-trivial I* ⟹ *I* ⊢*g gpv* √
⟨*proof*⟩

lemma *I-trivial-WT-gpvI*:
assumes ⋀*gpv* :: (*'a*, *'out*, *'in*) *gpv*. *I* ⊢*g gpv* √
shows *I-trivial I*
⟨*proof*⟩

4.13 Sub-gpvs

context begin
qualified inductive *sub-gpvsp* :: (*'out*, *'in*) *I* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ (*'a*, *'out*, *'in*) *gpv* ⇒ *bool*
for *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 $\mathcal{I}$ ⊢*g gpv* √ **and** *gpv'* ∈ *sub-gpvs* $\mathcal{I}$ *gpv*
shows $\mathcal{I}$ ⊢*g gpv'* √
⟨*proof*⟩

lemma *WT-sub-gpvsI*:
⟦ ⋀*out c*. *IO out c* ∈ *set-spmf* (*the-gpv gpv*) ⟹ *out* ∈ *outs-*$\mathcal{I}$ Γ;
⋀*gpv'*. *gpv'* ∈ *sub-gpvs* Γ *gpv* ⟹ Γ ⊢*g gpv'* √ ⟧
⟹ Γ ⊢*g gpv* √
⟨*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* ≡ λ*gen-lossless-gpv gpv*. ∃ *pa*. *gpv* = *GPV pa* ∧
lossless-spmf pa ∧ (∀ *out c input*. *IO out c* ∈ *set-spmf pa* ⟶ *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* ⟶ *gen-lossless-gpv* (*c input*))
and *co'* ≡ *co* — We use a copy of *co* such that we can do case distinctions on *co'* without the simplifier rewriting the *co* in the local abbreviations for the constants.
begin

lemma *gen-lossless-gpv-mono*: *mono F*
⟨*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-*$\mathcal{I}$ $\mathcal{I}$ *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-*$\mathcal{I}$ $\mathcal{I}$ *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-*$\mathcal{I}$ $\mathcal{I}$ *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 ℐ gpv* ⋀*x*. *x* ∈ *results-gpv ℐ gpv* ⟹ *gen-lossless-gpv co ℐ* (*f x*)
shows *gen-lossless-gpv co ℐ* (*gpv* ⤜ *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 ℐ* (*gpv* ⤜ *f*)
shows *gen-lossless-gpv co 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 ℐ* (*gpv* ⤜ *f*)
and *x* ∈ *results-gpv ℐ gpv*
shows *gen-lossless-gpv co 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 𝓘' gpv'*; *rel-𝓘 C (=) 𝓘 𝓘'* ⟧
⟹ *colossless-gpv 𝓘 gpv*
⟨*proof*⟩

lemma *gen-lossless-gpv-parametric'*:
((=) ===> *rel-𝓘 C R* ===> *rel-gpv'' A C R* ===> (=))
gen-lossless-gpv gen-lossless-gpv
⟨*proof*⟩

lemma *gen-lossless-gpv-parametric* [*transfer-rule*]:
((=) ===> *rel-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 𝓘-full (*map-gpv f g gpv*) = *gen-lossless-gpv b 𝓘-full gpv*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *gen-lossless-gpv-map-id* [*simp*]:
gen-lossless-gpv b 𝓘 (*map-gpv f id gpv*) = *gen-lossless-gpv b 𝓘 gpv*
⟨*proof*⟩

lemma *results-gpv-try-gpv* [*simp*]:
results-gpv 𝓘 (*TRY gpv ELSE gpv'*) =
results-gpv 𝓘 gpv ∪ (*if colossless-gpv 𝓘 gpv then* {} *else results-gpv 𝓘 gpv'*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *results'-gpv-try-gpv* [*simp*]:
results'-gpv (*TRY gpv ELSE gpv'*) =
results'-gpv gpv ∪ (*if colossless-gpv 𝓘-full gpv then* {} *else results'-gpv gpv'*)
⟨*proof*⟩

lemma *outs'-gpv-try-gpv* [*simp*]:
outs'-gpv (*TRY gpv ELSE gpv'*) =
outs'-gpv gpv ∪ (*if colossless-gpv 𝓘-full gpv then* {} *else outs'-gpv gpv'*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *pred-gpv-try* [*simp*]:
pred-gpv P Q (*try-gpv gpv gpv'*) = (*pred-gpv P Q gpv* ∧ (¬ *colossless-gpv 𝓘-full gpv* ⟶ *pred-gpv P Q gpv'*))
⟨*proof*⟩

lemma *lossless-WT-gpv-induct* [*consumes 2*, *case-names lossless-gpv*]:

assumes *lossless*: *lossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *step*: $\bigwedge p.$ ⟦
lossless-spmf p;
$\bigwedge$*out c. IO out c* ∈ *set-spmf p* ⟹ *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$;
$\bigwedge$*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*);
$\bigwedge$*out c input.* ⟦*IO out c* ∈ *set-spmf p*; *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟹ *input* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out*⟧ ⟹ $\mathcal{I} \vdash g$ *c input* $\surd$;
$\bigwedge$*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*: $\mathcal{I} \vdash g$ *gpv* $\surd$
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*) ⟹ $\mathcal{I} \vdash g$ *gpv* $\surd$;
$\bigwedge$*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 $\Longrightarrow$ (*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 $\Longrightarrow$ *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''* $\Longrightarrow$ *gpv'* = *gpv'''* ⟧
$\Longrightarrow$ *try-gpv gpv gpv'* = *try-gpv gpv'' 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* ℐ*-full* (*catch-gpv gpv*)
⟨*proof*⟩

lemma *colosless-gpv-catch-gpv-conv-map*:
colossless-gpv 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 ℐ (*catch-gpv gpv*) = *Some* ‘ *results-gpv* ℐ *gpv* ∪ (*if colossless-gpv* ℐ *gpv then* {} *else* {*None*})
⟨*proof*⟩

lemma *Some-in-results-gpv-catch-gpv* [*simp*]:
Some x ∈ *results-gpv* ℐ (*catch-gpv gpv*) ⟷ *x* ∈ *results-gpv* 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*) ⟹ *f x* = *f' x* ⟧

$\Longrightarrow$ *bind-gpv′ gpv f* = *bind-gpv′ gpv′ f′*
⟨*proof*⟩

lemma *bind-gpv′-cong2*:
⟦ *gpv* = *gpv′*; ⋀*x*. *x* ∈ *results′-gpv gpv′* $\Longrightarrow$ *f* (*Some x*) = *f′* (*Some x*); ¬ *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*: ⋀*gpv gpv′*. *R gpv gpv′*
$\Longrightarrow$ *rel-spmf* (*rel-generat* (=) (=) (*rel-fun* (=) (λ*gpv gpv′*. *R gpv gpv′* ∨ *gpv* = *gpv′* ∨
(∃ *gpv2* :: (*′b*, *′call*, *′ret*) *gpv*. ∃ *gpv2′* :: (*′c*, *′call*, *′ret*) *gpv*. ∃*f f′*. *gpv* = *bind-gpv gpv2 f* ∧ *gpv′* = *bind-gpv gpv2′ f′* ∧
rel-gpv (λ*x y*. *R* (*f x*) (*f′ y*)) (=) *gpv2 gpv2′*))))
(*the-gpv gpv*) (*the-gpv gpv′*)
shows *gpv* = *gpv′*
⟨*proof*⟩

Inlining one gpv into another. This may throw out arbitrarily many interactions between the two gpvs if the inlined one does not call its callee. So we define it as the coiteration of a least-fixpoint search operator.

context
fixes *callee* :: *′s* ⇒ *′call* ⇒ (*′ret* × *′s*, *′call′*, *′ret′*) *gpv*
notes [[*function-internals*]]
begin

partial-function (*spmf*) *inline1*
:: (*′a*, *′call*, *′ret*) *gpv* ⇒ *′s*
⇒ (*′a* × *′s* + *′call′* × (*′ret* × *′s*, *′call′*, *′ret′*) *rpv* × (*′a*, *′call*, *′ret*) *rpv*) *spmf*
where
inline1 gpv s =
the-gpv gpv ⤜
case-generat (λ*x*. *return-spmf* (*Inl* (*x*, *s*)))
(λ*out rpv*. *the-gpv* (*callee s out*) ⤜
case-generat (λ(*x*, *y*). *inline1* (*rpv x*) *y*)
(λ*out rpv′*. *return-spmf* (*Inr* (*out*, *rpv′*, *rpv*))))

lemma *inline1-unfold*:
inline1 gpv s =
the-gpv gpv ⤜
case-generat (λ*x*. *return-spmf* (*Inl* (*x*, *s*)))
(λ*out rpv*. *the-gpv* (*callee s out*) ⤜
case-generat (λ(*x*, *y*). *inline1* (*rpv x*) *y*)

(λout 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) (λlr. 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″, λr1. 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, λr. 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 (λs x. lift-spmf (p s x)) callee gpv s s′)* ⊆ *range Inl*
⟨*proof*⟩

lemma *in-set-inline2-lift-spmf1*: *y* ∈ *set-spmf (inline2 (λs 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 (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*, *out′*, *rpv′*, *rpv*))))

private lemma *inline1′-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*inline1′*. *P* (λ*gpv s*. *inline1′* (*gpv*, *s*)))
and *P* (λ- -. *return-pmf None*)
and ⋀*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*, *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*: $\bigwedge s\ x.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow$ *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*: $\mathcal{I} \vdash g$ *gpv* $\surd$
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 $\mathcal{I} \vdash g$ *gpv* $\surd$
shows *rpv' x* ∈ *sub-gpvs* $\mathcal{I}$ *gpv*
⟨*proof*⟩

context
assumes *WT*: $\bigwedge x\ s.\ x \in$ *outs-*$\mathcal{I}$ $\mathcal{I} \Longrightarrow \mathcal{I}' \vdash g$ *callee s x* $\surd$
begin

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

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

end

context
fixes *gpv* :: ('*a*, '*call*, '*ret*) *gpv*
assumes *gpv*: *lossless-gpv* $\mathcal{I}$ *gpv* $\mathcal{I} \vdash g$ *gpv* $\surd$
begin

lemma *lossless-spmf-inline1*:

assumes *lossless*: $\bigwedge s\ x.\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow \mathit{lossless}\text{-}\mathit{spmf}\ (\mathit{the}\text{-}\mathit{gpv}\ (\mathit{callee}\ s\ x))$
shows *lossless-spmf* (*inline1 callee gpv s*)
⟨*proof*⟩

lemma *lossless-gpv-inline1*:
assumes ∗: $\mathit{Inr}\ (\mathit{out}, \mathit{rpv}, \mathit{rpv}') \in \mathit{set}\text{-}\mathit{spmf}\ (\mathit{inline1}\ \mathit{callee}\ \mathit{gpv}\ s)$
and ∗∗: $\mathit{input} \in \mathit{responses}\text{-}\mathcal{I}\ \mathcal{I}'\ \mathit{out}$
and *lossless*: $\bigwedge s\ x.\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow \mathit{lossless}\text{-}\mathit{gpv}\ \mathcal{I}'\ (\mathit{callee}\ s\ x)$
shows *lossless-gpv* $\mathcal{I}'$ (*rpv input*)
⟨*proof*⟩

lemma *lossless-results-inline1*:
assumes $\mathit{Inr}\ (\mathit{out}, \mathit{rpv}, \mathit{rpv}') \in \mathit{set}\text{-}\mathit{spmf}\ (\mathit{inline1}\ \mathit{callee}\ \mathit{gpv}\ s)$
and $(x, s') \in \mathit{results}\text{-}\mathit{gpv}\ \mathcal{I}'\ (\mathit{rpv}\ \mathit{input})$
and $\mathit{input} \in \mathit{responses}\text{-}\mathcal{I}\ \mathcal{I}'\ \mathit{out}$
shows *lossless-gpv* $\mathcal{I}$ ($\mathit{rpv}'\ x$)
⟨*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-gpv* $\mathcal{I}$ *gpv* $\mathcal{I} \vdash g$ *gpv* $\surd$
and *lossless*: $\bigwedge s\ x.\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow \mathit{lossless}\text{-}\mathit{gpv}\ \mathcal{I}'\ (\mathit{callee}\ s\ x)$
shows *lossless-gpv* $\mathcal{I}'$ (*inline callee gpv s*)
⟨*proof*⟩

end

definition *id-oracle* :: ′*s* ⇒ ′*call* ⇒ (′*ret* × ′*s*, ′*call*, ′*ret*) *gpv*
where *id-oracle s x* = *Pause x* (λ*x*. *Done* (*x*, *s*))

lemma *inline1-id-oracle*:
inline1 id-oracle gpv s =
map-spmf (λ*generat*. *case generat of Pure x* ⇒ *Inl* (*x*, *s*) | *IO out c* ⇒ *Inr* (*out*, λ*x*. *Done* (*x*, *s*), *c*)) (*the-gpv gpv*)
⟨*proof*⟩

lemma *inline-id-oracle* [*simp*]: *inline id-oracle gpv s* = *map-gpv* (λ*x*. (*x*, *s*)) *id gpv*
⟨*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 ⤜ f) s = exec-gpv callee c s ⤜ (λ(x, s′) ⇒ 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 ⤜ (λ(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*′ ⇒ ′*c*′ ⇒ (′*r*′ × ′*s*′, ′*c*, ′*r*) *gpv*
shows *exec-gpv callee (inline gpv c′ s′) s =*
map-spmf (λ(x, s′, s). ((x, s′), s)) (exec-gpv (λ(s′, s) y. map-spmf (λ((x, s′), s). (x, s′, s)) (exec-gpv callee (gpv s′ y) s)) c′ (s′, s))
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *ord-spmf-exec-gpv*:
assumes *callee*: ⋀*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

$(\lambda a.\ (a,\ s))\ x)$
| *resumption.Pause out c* ⇒ *bind-spmf* (*callee s out*) (λ(*input*, *s′*). *execp-resumption* (*c input*) *s′*))

simps-of-case *execp-resumption-simps* [*simp*]: *execp-resumption.simps*

lemma *execp-resumption-ABORT* [*simp*]: *execp-resumption ABORT s* = *return-pmf None*
⟨*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* (=)) (λ*r*. *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* ⟧
⟹ *I* ⊢*c callee* √

lemmas *WT-calleeI* = *WT-callee*
hide-fact *WT-callee*

lemma *WT-calleeD*: $[\![\mathcal{I} \vdash c\ callee\ \surd; (ret, s) \in set\text{-}spmf\ (callee\ out); out \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow ret \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out$
⟨*proof*⟩

lemma *WT-callee-full* [*intro*!, *simp*]: $\mathcal{I}\text{-}full \vdash c\ callee\ \surd$
⟨*proof*⟩

lemma *WT-callee-parametric* [*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique R*
shows $(rel\text{-}\mathcal{I}\ C\ R ===> (C ===> rel\text{-}spmf\ (rel\text{-}prod\ R\ S)) ===> (=))$ *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*: $\bigwedge s\ x\ y\ s'. [\![(y, s') \in set\text{-}spmf\ (callee\ s\ x); I\ s; x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow I\ s'$
and *WT-callee*: $\bigwedge s. I\ s \Longrightarrow \mathcal{I} \vdash c\ callee\ s\ \surd$
begin

lemma *callee-invariant'*: $[\![(y, s') \in set\text{-}spmf\ (callee\ s\ x); I\ s; x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow I\ s' \wedge y \in responses\text{-}\mathcal{I}\ \mathcal{I}\ x$
⟨*proof*⟩

lemma *exec-gpv-invariant'*:
$[\![I\ s; \mathcal{I} \vdash g\ gpv\ \surd\ ]\!] \Longrightarrow set\text{-}spmf\ (exec\text{-}gpv\ callee\ gpv\ s) \subseteq \{(x, s').\ I\ s'\}$
⟨*proof*⟩

lemma *exec-gpv-invariant*:
$[\![(x, s') \in set\text{-}spmf\ (exec\text{-}gpv\ callee\ gpv\ s); I\ s; \mathcal{I} \vdash g\ gpv\ \surd\ ]\!] \Longrightarrow I\ s'$
⟨*proof*⟩

lemma *interaction-bounded-by-exec-gpv-count'*:
fixes *count*
assumes *bound*: *interaction-bounded-by consider gpv n*
and *count*: $\bigwedge s\ x\ y\ s'. [\![(y, s') \in set\text{-}spmf\ (callee\ s\ x); I\ s; consider\ x; x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow count\ s' \leq eSuc\ (count\ s)$
and *ignore*: $\bigwedge s\ x\ y\ s'. [\![(y, s') \in set\text{-}spmf\ (callee\ s\ x); I\ s; \neg\ consider\ x; x \in outs\text{-}\mathcal{I}\ \mathcal{I}\ ]\!] \Longrightarrow count\ s' \leq count\ s$

and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
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-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *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-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *count s'* $\leq$ *count s*
and *WT*: $\mathcal{I} \vdash g$ *gpv* $\surd$
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-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *count s'* $\leq$ *Suc* (*count s*)
and *ignore*: $\bigwedge$*s x y s'*. ⟦ (*y*, *s'*) $\in$ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *count s'* $\leq$ *count s*
and *outs*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *I*: *I s*
shows *count s'* $\leq$ *n* + *count s*
⟨*proof*⟩

lemma *pred-spmf-calleeI*: ⟦ *I s*; *x* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *pred-spmf* (*pred-prod* (λ-. *True*) *I*) (*callee s x*)
⟨*proof*⟩

lemma *lossless-exec-gpv*:
assumes *gpv*: *lossless-gpv* $\mathcal{I}$ *gpv*
and *callee*: $\bigwedge$*s out*. ⟦ *out* $\in$ *outs-*$\mathcal{I}$ $\mathcal{I}$; *I s* ⟧ $\Longrightarrow$ *lossless-spmf* (*callee s out*)
and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
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* ℐ.
(∀ *s* ∈ *Collect I*. ∀ *x* ∈ *outs-ℐ* ℐ. ∀ (*y*, *s'*) ∈ *set-spmf* (*callee s x*). *I s'*) ∧
(∀ *s* ∈ *Collect I*. ℐ ⊢*c callee s* √))
⟨*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-ℐ C R* ===> (=))
callee-invariant-on callee-invariant-on
⟨*proof*⟩

lemma *callee-invariant-on-cong*:
⟦ *I* = *I'*; *outs-ℐ* ℐ = *outs-ℐ* ℐ';
⋀*s x*. ⟦ *I' s*; *x* ∈ *outs-ℐ* ℐ' ⟧ ⟹ *set-spmf* (*callee s x*) ⊆ *responses-ℐ* ℐ *x* × *Collect I'* ⟷ *set-spmf* (*callee' s x*) ⊆ *responses-ℐ* ℐ' *x* × *Collect I'* ⟧
⟹ *callee-invariant-on callee I* ℐ = *callee-invariant-on callee' I'* ℐ'
⟨*proof*⟩

abbreviation *callee-invariant* :: ('*s* ⇒ '*a* ⇒ ('*b* × '*s*) *spmf*) ⇒ ('*s* ⇒ *bool*) ⇒ *bool*
where *callee-invariant callee I* ≡ *callee-invariant-on callee I ℐ-full*

interpretation *oi-True*: *callee-invariant-on callee* λ-. *True ℐ-full* **for** *callee*
⟨*proof*⟩

lemma *callee-invariant-on-return-spmf* [*simp*]:
callee-invariant-on (λ*s x*. *return-spmf* (*f s x*)) *I* ℐ ⟷ (∀ *s*. ∀ *x*∈*outs-ℐ* ℐ. *I s* ⟶ *I* (*snd* (*f s x*)) ∧ *fst* (*f s x*) ∈ *responses-ℐ* ℐ *x*)
⟨*proof*⟩

lemma *callee-invariant-return-spmf* [*simp*]:
callee-invariant (λ*s x*. *return-spmf* (*f s x*)) *I* ⟷ (∀ *s x*. *I s* ⟶ *I* (*snd* (*f s x*)))
⟨*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*) ℐ

$\longleftrightarrow$ $(\forall\, s.\ \mathcal{I} \vdash c\ callee\ s\ \surd)$
⟨*proof*⟩

lemma *lossless-exec-gpv*:
⟦ *lossless-gpv* $\mathcal{I}$ *gpv*; $\bigwedge s\ out.\ out \in outs\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow lossless\text{-}spmf\ (callee\ s\ out)$;
$\mathcal{I} \vdash g\ gpv\ \surd$; $\bigwedge s.\ \mathcal{I} \vdash c\ callee\ s\ \surd$ ⟧
$\Longrightarrow$ *lossless-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

lemma *in-set-spmf-exec-gpv-into-results'-gpv*:
assumes ∗: $(x,\ s') \in set\text{-}spmf\ (exec\text{-}gpv\ callee\ gpv\ s)$
shows $x \in results'\text{-}gpv\ gpv$
⟨*proof*⟩

context fixes $\mathcal{I} :: ('out,\ 'in)\ \mathcal{I}$ **begin**

primcorec *restrict-gpv* :: $('a,\ 'out,\ 'in)\ gpv \Rightarrow ('a,\ 'out,\ 'in)\ gpv$
where
restrict-gpv gpv = *GPV* (
map-pmf (*case-option None* (*case-generat* (*Some* ∘ *Pure*)
(λ*out c. if out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ *then Some* (*IO out* (λ*input. 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* (λ*input. 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*) (λ*x. restrict-gpv* (*f x*))
⟨*proof*⟩

lemma *WT-restrict-gpv* [*simp*]: $\mathcal{I} \vdash g$ *restrict-gpv gpv* $\surd$
⟨*proof*⟩

lemma *exec-gpv-restrict-gpv*:
assumes $\mathcal{I} \vdash g$ *gpv* $\surd$ **and** *WT-callee*: $\bigwedge s.\ \mathcal{I} \vdash c$ *callee s* $\surd$
shows *exec-gpv callee* (*restrict-gpv gpv*) *s* = *exec-gpv callee gpv s*
⟨*proof*⟩

lemma *in-outs'-restrict-gpvD*: $x \in$ *outs'-gpv* (*restrict-gpv gpv*) $\Longrightarrow x \in$ *outs-*$\mathcal{I}$ $\mathcal{I}$
⟨*proof*⟩

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

lemma *lossless-restrict-gpvI*: ⟦ *lossless-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ $\Longrightarrow$ *lossless-gpv* $\mathcal{I}$ (*restrict-gpv gpv*)
⟨*proof*⟩

lemma *lossless-restrict-gpvD*: ⟦ *lossless-gpv* $\mathcal{I}$ (*restrict-gpv gpv*); $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ $\Longrightarrow$ *lossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *colossless-restrict-gpvD*:
⟦ *colossless-gpv* $\mathcal{I}$ (*restrict-gpv gpv*); $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ $\Longrightarrow$ *colossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *colossless-restrict-gpvI*:
⟦ *colossless-gpv* $\mathcal{I}$ *gpv*; $\mathcal{I} \vdash g$ *gpv* $\surd$ ⟧ $\Longrightarrow$ *colossless-gpv* $\mathcal{I}$ (*restrict-gpv gpv*)
⟨*proof*⟩

lemma *gen-colossless-restrict-gpv* [*simp*]:
$\mathcal{I} \vdash g$ *gpv* $\surd$ $\Longrightarrow$ *gen-lossless-gpv b* $\mathcal{I}$ (*restrict-gpv gpv*) $\longleftrightarrow$ *gen-lossless-gpv b* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *interaction-bound-restrict-gpv*:
interaction-bound consider (*restrict-gpv gpv*) $\leq$ *interaction-bound consider gpv*
⟨*proof*⟩

lemma *interaction-bounded-by-restrict-gpvI* [*interaction-bound*, *simp*]:
interaction-bounded-by consider gpv n $\Longrightarrow$ *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-*$\mathcal{I}$ *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 $\Longrightarrow$ (*rel-*$\mathcal{I}$ *C* (=) ===> *rel-gpv A C* ===> *rel-gpv A C*) *restrict-gpv restrict-gpv*
⟨*proof*⟩

lemma *map-restrict-gpv*: *map-gpv f id* (*restrict-gpv* $\mathcal{I}$ *gpv*) = *restrict-gpv* $\mathcal{I}$ (*map-gpv f id gpv*)
  for *gpv* :: ('*a*, '*out*, '*ret*) *gpv*
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *exec-gpv-restrict-gpv-invariant*:
  assumes $\mathcal{I}$ ⊢*g gpv* √ **and** *I s*
  shows *exec-gpv callee* (*restrict-gpv* $\mathcal{I}$ *gpv*) *s* = *exec-gpv callee gpv s*
⟨*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* ⤜ *f*) ⟷ *finite-gpv gpv* ∧ (∀ *x*∈*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* (=) $\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *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* $\mathcal{I}'$ $\mathcal{I}$ ⟧ ⟹ *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* (=) $\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *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'*) ⟷ *finite-gpv* $\mathcal{I}$ *gpv* ∧ (*colossless-gpv* $\mathcal{I}$ *gpv* ∨ *finite-gpv* $\mathcal{I}$ *gpv'*)
(**is** *?lhs* = -)
⟨*proof*⟩

lemma *lossless-gpv-conv-finite*:
lossless-gpv $\mathcal{I}$ *gpv* ⟷ *finite-gpv* $\mathcal{I}$ *gpv* ∧ *colossless-gpv* $\mathcal{I}$ *gpv*
(**is** *?loss* ⟷ *?fin* ∧ *?co*)
⟨*proof*⟩

lemma *colossless-gpv-try* [*simp*]:
colossless-gpv $\mathcal{I}$ (*TRY* *gpv* *ELSE* *gpv'*) ⟷ *colossless-gpv* $\mathcal{I}$ *gpv* ∨ *colossless-gpv* $\mathcal{I}$ *gpv'*
(**is** *?lhs* ⟷ *?gpv* ∨ *?gpv'*)
⟨*proof*⟩

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*: ¬ *bad s*
  and *count*: $\bigwedge$*s x y s'*. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *count s'* ≤ *Suc* (*count s*)
  and *ignore*: $\bigwedge$*s x y s'*. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); ¬ *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *count s'* ≤ *count s*
  and *bad*: $\bigwedge$*s' x*. ⟦ ¬ *bad s'*; *count s'* < *n* + *count s*; *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s' x*)) *True* ≤ *k*
  and *consider*: $\bigwedge$*s x y s'*. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); ¬ *bad s*; *bad s'*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *consider x*
  and *k-nonneg*: *k* ≥ *0*
  and *WT-gpv*: $\mathcal{I}$ ⊢*g gpv* $\surd$
  and *WT-callee*: $\bigwedge$*s*. $\mathcal{I}$ ⊢*c callee s* $\surd$
  shows *spmf* (*map-spmf* (*bad* ∘ *snd*) (*exec-gpv callee gpv s*)) *True* ≤ *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-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *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-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *count s'* ≤ *count s*
  and *bad*: $\bigwedge$*s' x*. ⟦ *I s'*; ¬ *bad s'*; *count s'* < *n* + *count s*; *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *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-*$\mathcal{I}$ $\mathcal{I}$ ⟧ $\Longrightarrow$ *consider x*

and *k-nonneg*: $k \geq 0$
and *WT-gpv*: $\mathcal{I} \vdash g\ gpv\ \surd$
shows *spmf* (*map-spmf* (*bad* ∘ *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*: ⋀*s x y s'*. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *I s*; *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *count s'* ≤ *Suc* (*count s*)
and *ignore*: ⋀*s x y s'*. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *count s'* ≤ *count s*
and *bad*: ⋀*s' x*. ⟦ *I s'*; ¬ *bad s'*; *count s'* < *n* + *count s*; *consider x*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s' x*)) *True* ≤ *k*
and *consider*: ⋀*s x y s'*. ⟦ (*y*, *s'*) ∈ *set-spmf* (*callee s x*); *I s*; ¬ *bad s*; *bad s'*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *consider x*
and *k-nonneg*: $k \geq 0$
and *WT-gpv*: $\mathcal{I} \vdash g\ gpv\ \surd$
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*: ⋀*s x*. ⟦ *I s*; ¬ *bad s*; *x* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$ ⟧ ⟹ *spmf* (*map-spmf* (*bad* ∘ *snd*) (*callee s x*)) *True* ≤ *k*
and *k-nonneg*: 0 ≤ *k*
and *WT-gpv*: $\mathcal{I} \vdash g\ gpv\ \surd$
shows *spmf* (*map-spmf* (*bad* ∘ *snd*) (*exec-gpv callee gpv s*)) *True* ≤ *k* ∗ *n*
⟨*proof*⟩

end

end

4.18 Expectation transformer semantics

theory *GPV-Expectation* **imports**
Generative-Probabilistic-Value
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*: $[\![y > 0 \Longrightarrow x \leq 1]\!] \Longrightarrow x * y \leq y$ **for** $x\ y ::$ *ennreal*
⟨*proof*⟩

lemma *ennreal-leI*: $x \leq$ *enn2real* $y \Longrightarrow$ *ennreal* $x \leq y$
⟨*proof*⟩

lemma *enn2real-INF*: $[\![A \neq \{\}; \forall x{\in}A.\ f\ x < \top]\!] \Longrightarrow$ *enn2real* $(INF\ x{:}A.\ f\ x) = (INF\ x{:}A.$ *enn2real* $(f\ x))$
⟨*proof*⟩

lemma *monotone-times-ennreal1*: *monotone* $(\leq)$ $(\leq)$ $(\lambda x.\ x * y ::$ *ennreal*$)$
⟨*proof*⟩

lemma *monotone-times-ennreal2*: *monotone* $(\leq)$ $(\leq)$ $(\lambda x.\ y * x ::$ *ennreal*$)$
⟨*proof*⟩

lemma *mono2mono-times-ennreal*[*THEN lfp.mono2mono2*, *cont-intro*, *simp*]:
shows *monotone-times-ennreal*: *monotone* (*rel-prod* $(\leq)$ $(\leq)$) $(\leq)$ $(\lambda(x,\ y).\ x * y ::$ *ennreal*$)$
⟨*proof*⟩

lemma *mcont-times-ennreal1*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda y.\ x * y ::$ *ennreal*$)$
⟨*proof*⟩

lemma *mcont-times-ennreal2*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda y.\ y * x ::$ *ennreal*$)$
⟨*proof*⟩

lemma *mcont2mcont-times-ennreal* [*cont-intro*, *simp*]:
$[\![$ *mcont lub ord Sup* $(\leq)$ $(\lambda x.\ f\ x)$;
mcont lub ord Sup $(\leq)$ $(\lambda x.\ g\ x)$ $]\!]$
$\Longrightarrow$ *mcont lub ord Sup* $(\leq)$ $(\lambda x.\ f\ x * g\ x ::$ *ennreal*$)$
⟨*proof*⟩

lemma *ereal-INF-cmult*: $0 < c \Longrightarrow (INF\ i{:}I.\ c * f\ i) =$ *ereal* $c * (INF\ i{:}I.\ f\ i)$
⟨*proof*⟩

lemma *ereal-INF-multc*: $0 < c \Longrightarrow (INF\ i{:}I.\ f\ i * c) = (INF\ i{:}I.\ f\ i) *$ *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 $c * (INF\ i{:}I.\ f\ i) = (INF\ i{:}I.\ c * f\ i ::$*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* : ⋃*x*∈*A*. *B x*. *f z*) = (*INF x*:*A*. *INF z*:*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 (λf. F f x)) ⟹ lfp.mono-body (λf. nn-integral M (F f))
⟨proof⟩

lemma *INF-mono-lfp* [*partial-function-mono*]:
(⋀x. lfp.mono-body (λf. F f x)) ⟹ lfp.mono-body (λf. INF x:M. F f x)
⟨proof⟩

lemmas *parallel-fixp-induct-1-2 = parallel-fixp-induct-uc*[
of - - - - λx. x - λx. x case-prod - curry,
where *P=λf g. P f (curry g)*,
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - - refl refl]
for *P*

lemma *monotone-ennreal-add1*: *monotone* (≤) (≤) (λx. x + y :: *ennreal*)
⟨*proof*⟩

lemma *monotone-ennreal-add2*: *monotone* (≤) (≤) (λy. 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* (≤)) (≤) (λx. *f x* + *g x* :: *ennreal*)
⟨*proof*⟩

context
fixes *fail* :: *ennreal*
and *I* :: (*'out*, *'ret*) *I*
and *f* :: *'a* ⇒ *ennreal*
notes [[*function-internals*]]
begin

partial-function (*lfp-strong*) *expectation-gpv* :: (*'a*, *'out*, *'ret*) *gpv* ⇒ *ennreal*
where
expectation-gpv gpv =
(∫⁺ *generat*. (*case generat of Pure x* ⇒ *f x*
| *IO out c* ⇒ *INF r:responses-I I out*. *expectation-gpv* (*c r*))
∂*measure-spmf* (*the-gpv gpv*))
+ *fail* ∗ *pmf* (*the-gpv gpv*) *None*

lemma *expectation-gpv-fixp-induct* [*case-names adm bottom step*]:
assumes *lfp.admissible P*
and *P* (λ-. *0*)

and $\bigwedge$*expectation-gpv'*. ⟦ $\bigwedge$*gpv*. *expectation-gpv' gpv* ≤ *expectation-gpv gpv*; *P expectation-gpv'* ⟧ ⟹
P (λ*gpv*. (∫⁺ *generat*. (*case generat of Pure x* ⇒ *f x* | *IO out c* ⇒ *INF r*:*responses-ℐ ℐ out*. *expectation-gpv'* (*c r*)) ∂*measure-spmf* (*the-gpv gpv*)) + *fail* ∗ *pmf* (*the-gpv gpv*) *None*)
shows *P expectation-gpv*
⟨*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-ℐ 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-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-ℐ C* (=) ===> (*A* ===> (=)) ===> *rel-gpv A C* ===> (=)) *expectation-gpv expectation-gpv*
⟨*proof*⟩

lemma *expectation-gpv-cong*:
fixes *fail fail'*
assumes *fail*: *fail* = *fail'*
and ℐ: ℐ = ℐ'
and *gpv*: *gpv* = *gpv'*
and *f*: $\bigwedge$*x*. *x* ∈ *results-gpv ℐ' gpv'* ⟹ *f x* = *g x*
shows *expectation-gpv fail ℐ f gpv* = *expectation-gpv fail' 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* ≤ *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*:
  $\mathcal{I}$ ⊢*g gpv* √ $\Longrightarrow$ *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*: $\mathcal{I}$ ⊢*g gpv* √ $\Longrightarrow$ *expectation-gpv fail* $\mathcal{I}$ (λ-. *c*) *gpv* ≤ *max c fail* **for** *fail*
⟨*proof*⟩

lemma *expectation-gpv-no-results*:
  ⟦ *results-gpv* $\mathcal{I}$ *gpv* = {}; $\mathcal{I}$ ⊢*g gpv* √ ⟧ $\Longrightarrow$ *expectation-gpv 0* $\mathcal{I}$ *f gpv* = *0*
⟨*proof*⟩

lemma *expectation-gpv-cmult*:
  fixes *fail*
  assumes *0* < *c* **and** *c* ≠ ⊤

shows $c * \mathit{expectation\text{-}gpv}\ \mathit{fail}\ \mathcal{I}\ f\ \mathit{gpv} = \mathit{expectation\text{-}gpv}\ (c * \mathit{fail})\ \mathcal{I}\ (\lambda x.\ c * f\ x)\ \mathit{gpv}$
⟨*proof*⟩

lemma *expectation-gpv-le-exec-gpv*:
assumes *callee*: $\bigwedge s\ x.\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow \mathit{lossless\text{-}spmf}\ (\mathit{callee}\ s\ x)$
and *WT-gpv*: $\mathcal{I} \vdash g\ \mathit{gpv}\ \surd$
and *WT-callee*: $\bigwedge s.\ \mathcal{I} \vdash c\ \mathit{callee}\ s\ \surd$
shows $\mathit{expectation\text{-}gpv}\ 0\ \mathcal{I}\ f\ \mathit{gpv} \le \int^+ (x, s).\ f\ x\ \partial\mathit{measure\text{-}spmf}\ (\mathit{exec\text{-}gpv}\ \mathit{callee}\ \mathit{gpv}\ s)$
⟨*proof*⟩

definition *weight-gpv* :: $('\mathit{out}, '\mathit{ret})\ \mathcal{I} \Rightarrow ('a, '\mathit{out}, '\mathit{ret})\ \mathit{gpv} \Rightarrow \mathit{real}$
where $\mathit{weight\text{-}gpv}\ \mathcal{I}\ \mathit{gpv} = \mathit{enn2real}\ (\mathit{expectation\text{-}gpv}\ 0\ \mathcal{I}\ (\lambda\text{-}.\ 1)\ \mathit{gpv})$

lemma *weight-gpv-Done* [*simp*]: $\mathit{weight\text{-}gpv}\ \mathcal{I}\ (\mathit{Done}\ x) = 1$
⟨*proof*⟩

lemma *weight-gpv-Fail* [*simp*]: $\mathit{weight\text{-}gpv}\ \mathcal{I}\ \mathit{Fail} = 0$
⟨*proof*⟩

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

lemma *weight-gpv-Pause* [*simp*]:
$(\bigwedge r.\ r \in \mathit{responses}\text{-}\mathcal{I}\ \mathcal{I}\ \mathit{out} \Longrightarrow \mathcal{I} \vdash g\ c\ r\ \surd)$
$\Longrightarrow \mathit{weight\text{-}gpv}\ \mathcal{I}\ (\mathit{Pause}\ \mathit{out}\ c) = (\mathit{if}\ \mathit{out} \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I}\ \mathit{then}\ \mathit{INF}\ r{:}\mathit{responses}\text{-}\mathcal{I}\ \mathcal{I}\ \mathit{out}.\ \mathit{weight\text{-}gpv}\ \mathcal{I}\ (c\ r)\ \mathit{else}\ 0)$
⟨*proof*⟩

lemma *weight-gpv-nonneg*: $0 \le \mathit{weight\text{-}gpv}\ \mathcal{I}\ \mathit{gpv}$
⟨*proof*⟩

lemma *weight-gpv-le-1*: $\mathcal{I} \vdash g\ \mathit{gpv}\ \surd \Longrightarrow \mathit{weight\text{-}gpv}\ \mathcal{I}\ \mathit{gpv} \le 1$
⟨*proof*⟩

theorem *weight-exec-gpv*:
assumes *callee*: $\bigwedge s\ x.\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I} \Longrightarrow \mathit{lossless\text{-}spmf}\ (\mathit{callee}\ s\ x)$
and *WT-gpv*: $\mathcal{I} \vdash g\ \mathit{gpv}\ \surd$
and *WT-callee*: $\bigwedge s.\ \mathcal{I} \vdash c\ \mathit{callee}\ s\ \surd$
shows $\mathit{weight\text{-}gpv}\ \mathcal{I}\ \mathit{gpv} \le \mathit{weight\text{-}spmf}\ (\mathit{exec\text{-}gpv}\ \mathit{callee}\ \mathit{gpv}\ s)$
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *weight-exec-gpv*:
assumes *callee*: $\bigwedge s\ x.\ ⟦\ x \in \mathit{outs}\text{-}\mathcal{I}\ \mathcal{I};\ I\ s\ ⟧ \Longrightarrow \mathit{lossless\text{-}spmf}\ (\mathit{callee}\ s\ x)$
and *WT-gpv*: $\mathcal{I} \vdash g\ \mathit{gpv}\ \surd$
and *I*: $I\ s$
shows $\mathit{weight\text{-}gpv}\ \mathcal{I}\ \mathit{gpv} \le \mathit{weight\text{-}spmf}\ (\mathit{exec\text{-}gpv}\ \mathit{callee}\ \mathit{gpv}\ s)$
including *lifting-syntax*

⟨*proof*⟩

4.19 Probabilistic termination

definition *pgen-lossless-gpv* :: *ennreal* ⇒ ('*c*, '*r*) $\mathcal{I}$ ⇒ ('*a*, '*c*, '*r*) *gpv* ⇒ *bool*
where *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* = (*expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1*) **for** *fail*

abbreviation *plossless-gpv* :: ('*c*, '*r*) $\mathcal{I}$ ⇒ ('*a*, '*c*, '*r*) *gpv* ⇒ *bool*
where *plossless-gpv* ≡ *pgen-lossless-gpv 0*

abbreviation *pfinite-gpv* :: ('*c*, '*r*) $\mathcal{I}$ ⇒ ('*a*, '*c*, '*r*) *gpv* ⇒ *bool*
where *pfinite-gpv* ≡ *pgen-lossless-gpv 1*

lemma *pgen-lossless-gpvI* [*intro?*]: *expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1* ⟹ *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* **for** *fail*
⟨*proof*⟩

lemma *pgen-lossless-gpvD*: *pgen-lossless-gpv fail* $\mathcal{I}$ *gpv* ⟹ *expectation-gpv fail* $\mathcal{I}$ (λ-. *1*) *gpv* = *1* **for** *fail*
⟨*proof*⟩

lemma *lossless-imp-plossless-gpv*:
assumes *lossless-gpv* $\mathcal{I}$ *gpv* $\mathcal{I}$ ⊢*g* *gpv* √
shows *plossless-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *finite-imp-pfinite-gpv*:
assumes *finite-gpv* $\mathcal{I}$ *gpv* $\mathcal{I}$ ⊢*g* *gpv* √
shows *pfinite-gpv* $\mathcal{I}$ *gpv*
⟨*proof*⟩

lemma *plossless-gpv-lossless-spmfD*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}$ *gpv*
and *WT*: $\mathcal{I}$ ⊢*g* *gpv* √
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*; $\mathcal{I}$ ⊢*g* *gpv* √ ⟧
⟹ *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*; $\mathcal{I}$ ⊢*g* *gpv* √ ⟧
⟹ *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* $\longleftrightarrow$ *colossless-gpv* $\mathcal{I}$ *gpv* $\wedge$ *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* $\longleftrightarrow$ *fail* = *1* **for** *fail*
⟨*proof*⟩

lemma *pgen-lossless-gpv-PauseI* [*simp*, *intro*!]:
⟦ *out* ∈ *outs-*$\mathcal{I}$ $\mathcal{I}$; $\bigwedge r$. *r* ∈ *responses-*$\mathcal{I}$ $\mathcal{I}$ *out* $\Longrightarrow$ *pgen-lossless-gpv fail* $\mathcal{I}$ (*c r*) ⟧
$\Longrightarrow$ *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*; $\bigwedge x$. *x* ∈ *results-gpv* $\mathcal{I}$ *gpv* $\Longrightarrow$ *pgen-lossless-gpv fail* $\mathcal{I}$ (*f x*) ⟧
$\Longrightarrow$ *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*) $\longleftrightarrow$ *lossless-spmf p* $\vee$ *fail* = *1* **for** *fail*
⟨*proof*⟩

lemma *expectation-gpv-top-pfinite*:
assumes *pfinite-gpv* $\mathcal{I}$ *gpv*
shows *expectation-gpv* $\top$ $\mathcal{I}$ (λ-. $\top$) *gpv* = $\top$
⟨*proof*⟩

lemma *pfinite-INF-le-expectation-gpv*:
fixes *fail* $\mathcal{I}$ *gpv f*
defines *c* ≡ *min* (*INF x*:*results-gpv* $\mathcal{I}$ *gpv*. *f x*) *fail*
assumes *fin*: *pfinite-gpv* $\mathcal{I}$ *gpv*
shows *c* ≤ *expectation-gpv fail* $\mathcal{I}$ *f gpv* (**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

lemma *plossless-INF-le-expectation-gpv*:
fixes *fail*
assumes *plossless-gpv* $\mathcal{I}$ *gpv* **and** $\mathcal{I} \vdash g$ *gpv* $\surd$
shows (*INF x*:*results-gpv* $\mathcal{I}$ *gpv*. *f x*) ≤ *expectation-gpv fail* $\mathcal{I}$ *f gpv* (**is** *?lhs* ≤ *?rhs*)
⟨*proof*⟩

lemma *expectation-gpv-le-inline*:
fixes $\mathcal{I}'$

defines *expectation-gpv2* ≡ *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* × *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* ≤ *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* $\surd$
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* × *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* × *UNIV*
shows *lossless-spmf* (*exec-gpv callee gpv s*)
⟨*proof*⟩

end

theory *GPV-Bisim* **imports**
GPV-Expectation
begin

4.20 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* $\Longrightarrow$ *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*: $\bigwedge s1\ s2\ x.\ X\ s1\ s2 \Longrightarrow$ *rel-spmf* $(\lambda(a, s1')\ (b, s2').\ a = b \wedge X\ s1'\ s2')$ $(oracle1\ s1\ x)$ $(oracle2\ s2\ x)$
and *R*: $\bigwedge x\ s1'\ s2'.\ [\![X\ s1'\ s2';\ (x, s1') \in$ *set-spmf* $(exec\text{-}gpv\ oracle1\ gpv\ s1);$ $(x, s2') \in$ *set-spmf* $(exec\text{-}gpv\ oracle2\ gpv\ s2)\ ]\!] \Longrightarrow R\ (x, s1')\ (x, s2')$
shows *rel-spmf R* $(exec\text{-}gpv\ oracle1\ gpv\ s1)$ $(exec\text{-}gpv\ oracle2\ gpv\ s2)$
⟨*proof*⟩

lemma *run-gpv-oracle-bisim*:
assumes *X s1 s2*
and $\bigwedge s1\ s2\ x.\ X\ s1\ s2 \Longrightarrow$ *rel-spmf* $(\lambda(a, s1')\ (b, s2').\ a = b \wedge X\ s1'\ s2')$ $(oracle1\ s1\ x)$ $(oracle2\ s2\ x)$
shows *run-gpv oracle1 gpv s1* = *run-gpv oracle2 gpv s2*
⟨*proof*⟩

context
fixes *joint-oracle* :: $('s1 \times 's2) \Rightarrow 'a \Rightarrow (('b \times 's1) \times ('b \times 's2))$ *spmf*
and *oracle1* :: $'s1 \Rightarrow 'a \Rightarrow ('b \times 's1)$ *spmf*
and *bad1* :: $'s1 \Rightarrow$ *bool*
and *oracle2* :: $'s2 \Rightarrow 'a \Rightarrow ('b \times 's2)$ *spmf*
and *bad2* :: $'s2 \Rightarrow$ *bool*
begin

partial-function (*spmf*) *exec-until-bad* :: $('x, 'a, 'b)$ *gpv* $\Rightarrow 's1 \Rightarrow 's2 \Rightarrow (('x \times 's1) \times ('x \times 's2))$ *spmf*
where
exec-until-bad gpv s1 s2 =
(*if bad1 s1* $\vee$ *bad2 s2 then pair-spmf* $(exec\text{-}gpv\ oracle1\ gpv\ s1)$ $(exec\text{-}gpv\ oracle2\ gpv\ s2)$
else bind-spmf $(the\text{-}gpv\ gpv)$ $(\lambda generat.$
case generat of Pure x $\Rightarrow$ *return-spmf* $((x, s1), (x, s2))$
| *IO out f* $\Rightarrow$ *bind-spmf* $(joint\text{-}oracle\ (s1, s2)\ out)$ $(\lambda((x, s1'), (y, s2')).$
if bad1 s1' $\vee$ *bad2 s2' then pair-spmf* $(exec\text{-}gpv\ oracle1\ (f\ x)\ s1')$ $(exec\text{-}gpv\ oracle2\ (f\ y)\ s2')$
else exec-until-bad $(f\ x)\ s1'\ s2')))$

lemma *exec-until-bad-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* $(fun\text{-}lub\ lub\text{-}spmf)$ $(fun\text{-}ord\ (ord\text{-}spmf\ (=)))$ $(\lambda f.\ P\ (\lambda gpv\ s1\ s2.\ f\ ((gpv, s1), s2)))$
and $P\ (\lambda\text{-}\ \text{-}.\ return\text{-}pmf\ None)$
and $\bigwedge exec\text{-}until\text{-}bad'.\ P\ exec\text{-}until\text{-}bad' \Longrightarrow$
$P\ (\lambda gpv\ s1\ s2.\ if\ bad1\ s1 \vee bad2\ s2\ then\ pair\text{-}spmf\ (exec\text{-}gpv\ oracle1\ gpv\ s1)\ (exec\text{-}gpv\ oracle2\ gpv\ s2)$
else bind-spmf $(the\text{-}gpv\ gpv)$ $(\lambda generat.$
case generat of Pure x $\Rightarrow$ *return-spmf* $((x, s1), (x, s2))$
| *IO out f* $\Rightarrow$ *bind-spmf* $(joint\text{-}oracle\ (s1, s2)\ out)$ $(\lambda((x, s1'), (y, s2')).$
if bad1 s1' $\vee$ *bad2 s2' then pair-spmf* $(exec\text{-}gpv\ oracle1\ (f\ x)\ s1')$ $(exec\text{-}gpv\ oracle2\ (f\ y)\ s2')$
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* ⇒ *'s2* ⇒ *bool*
and *oracle1* :: *'s1* ⇒ *'a* ⇒ (*'b* × *'s1*) *spmf*
and *oracle2* :: *'s2* ⇒ *'a* ⇒ (*'b* × *'s2*) *spmf*
assumes ∗: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and *bad*: *bad1 s1 = bad2 s2*
and *bisim*: ⋀*s1 s2 x*. ⟦ *X s1 s2*; *x* ∈ *outs-*ℐ ℐ ⟧ ⟹ *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1' = bad2 s2'* ∧ (*if bad2 s2' then X-bad s1' s2' else a = b* ∧ *X s1' s2'*)) (*oracle1 s1 x*) (*oracle2 s2 x*)
and *bad-sticky1*: ⋀*s2*. *bad2 s2* ⟹ *callee-invariant-on oracle1* (λ*s1*. *bad1 s1* ∧ *X-bad s1 s2*) ℐ
and *bad-sticky2*: ⋀*s1*. *bad1 s1* ⟹ *callee-invariant-on oracle2* (λ*s2*. *bad2 s2* ∧ *X-bad s1 s2*) ℐ
and *lossless1*: ⋀*s1 x*. ⟦ *bad1 s1*; *x* ∈ *outs-*ℐ ℐ ⟧ ⟹ *lossless-spmf* (*oracle1 s1 x*)
and *lossless2*: ⋀*s2 x*. ⟦ *bad2 s2*; *x* ∈ *outs-*ℐ ℐ ⟧ ⟹ *lossless-spmf* (*oracle2 s2 x*)
and *lossless*: *plossless-gpv* ℐ *gpv*
and *WT-oracle1*: ⋀*s1*. ℐ ⊢*c oracle1 s1* √
and *WT-oracle2*: ⋀*s2*. ℐ ⊢*c oracle2 s2* √
and *WT-gpv*: ℐ ⊢*g gpv* √
shows *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1' = bad2 s2'* ∧ (*if bad2 s2' then X-bad s1' s2' else a = b* ∧ *X s1' s2'*)) (*exec-gpv oracle1 gpv s1*) (*exec-gpv oracle2 gpv s2*)
(**is** *rel-spmf ?R ?p ?q*)
⟨*proof*⟩

lemma *exec-gpv-oracle-bisim-bad'*:
fixes *s1* :: *'s1* **and** *s2* :: *'s2* **and** *X* :: *'s1* ⇒ *'s2* ⇒ *bool*
and *oracle1* :: *'s1* ⇒ *'a* ⇒ (*'b* × *'s1*) *spmf*
and *oracle2* :: *'s2* ⇒ *'a* ⇒ (*'b* × *'s2*) *spmf*
assumes ∗: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and *bad*: *bad1 s1 = bad2 s2*
and *bisim*: ⋀*s1 s2 x*. ⟦ *X s1 s2*; *x* ∈ *outs-*ℐ ℐ ⟧ ⟹ *rel-spmf* (λ(*a*, *s1'*) (*b*, *s2'*). *bad1 s1' = bad2 s2'* ∧ (*if bad2 s2' then X-bad s1' s2' else a = b* ∧ *X s1' s2'*)) (*oracle1 s1 x*) (*oracle2 s2 x*)
and *bad-sticky1*: ⋀*s2*. *bad2 s2* ⟹ *callee-invariant-on oracle1* (λ*s1*. *bad1 s1* ∧ *X-bad s1 s2*) ℐ
and *bad-sticky2*: ⋀*s1*. *bad1 s1* ⟹ *callee-invariant-on oracle2* (λ*s2*. *bad2 s2* ∧ *X-bad s1 s2*) ℐ
and *lossless1*: ⋀*s1 x*. ⟦ *bad1 s1*; *x* ∈ *outs-*ℐ ℐ ⟧ ⟹ *lossless-spmf* (*oracle1 s1 x*)
and *lossless2*: ⋀*s2 x*. ⟦ *bad2 s2*; *x* ∈ *outs-*ℐ ℐ ⟧ ⟹ *lossless-spmf* (*oracle2 s2 x*)
and *lossless*: *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*: ⋀*s2*. *bad2 s2* ⟹ *callee-invariant-on oracle1* (λ*s1*. *bad1 s1* ∧ *X-bad s1 s2*) *ℐ*
and *bad-sticky2*: ⋀*s1*. *bad1 s1* ⟹ *callee-invariant-on oracle2* (λ*s2*. *bad2 s2* ∧ *X-bad s1 s2*) *ℐ*
and *lossless1*: ⋀*s1 x*. *bad1 s1* ⟹ *lossless-spmf* (*oracle1 s1 x*)
and *lossless2*: ⋀*s2 x*. *bad2 s2* ⟹ *lossless-spmf* (*oracle2 s2 x*)
and *lossless*: *lossless-gpv ℐ gpv*
and *WT-oracle1*: ⋀*s1*. *ℐ* ⊢*c oracle1 s1* √
and *WT-oracle2*: ⋀*s2*. *ℐ* ⊢*c oracle2 s2* √

and *WT-gpv*: $\mathcal{I} \vdash g$ *gpv* $\surd$
and *R*: $\bigwedge a\ s1\ b\ s2.\ [\![\ bad1\ s1 = bad2\ s2;\ \neg\ bad2\ s2 \Longrightarrow a = b \wedge X\ s1\ s2;\ bad2\ s2 \Longrightarrow X\text{-}bad\ s1\ s2\ ]\!] \Longrightarrow R\ (a,\ s1)\ (b,\ s2)$
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* $(\lambda(a,\ s1')\ (b,\ s2').\ bad1\ s1' = bad2\ s2' \wedge (\neg\ bad2\ s2' \longrightarrow a = b \wedge X\ s1'\ s2'))$ (*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* ($\lambda x.$ (*f x*, *bad x*)) *p*)) ($A \times \{False\}$) = *measure* (*measure-spmf* (*map-spmf* ($\lambda x.$ (*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*| ≤ *spmf* (*map-spmf bad p*) *True*
⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *exec-gpv-bind-materialize*:
fixes *f* :: *'s* ⇒ *'r spmf*
and *g* :: *'x* × *'s* ⇒ *'r* ⇒ *'y spmf*
and *s* :: *'s*
defines *exec-gpv2* ≡ *exec-gpv*
assumes *cond*: $\bigwedge s\ x\ y\ s'.\ [\![\ (y,\ s') \in$ *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*) ($\lambda as.$ *bind-spmf* (*f* (*snd as*)) (*g as*)) =
exec-gpv2 ($\lambda(r,\ s)\ x.$ *bind-spmf* (*callee s x*) ($\lambda(y,\ s').$ *if I s'* ∧ *r* = *None then map-spmf* ($\lambda r.$ (*y*, (*Some r*, *s'*))) (*f s'*) *else return-spmf* (*y*, (*r*, *s'*)))) *gpv* (*None*, *s*)
⟜ ($\lambda(a,\ r,\ s).$ *case r of None* ⇒ *bind-spmf* (*f s*) (*g* (*a*, *s*)) | *Some r'* ⇒ *g* (*a*, *s*) *r'*)
(**is** *?lhs* = *?rhs* **is** - = *bind-spmf* (*exec-gpv2 ?callee2* - -) -)
⟨*proof*⟩

primcorec *gpv-stop* :: (*'a*, *'c*, *'r*) *gpv* ⇒ (*'a option*, *'c*, *'r option*) *gpv*

where
the-gpv (*gpv-stop gpv*) =
map-spmf (*map-generat Some id* (λ*rpv input*. *case input of None* ⇒ *Done None* | *Some input'* ⇒ *gpv-stop* (*rpv input'*)))
(*the-gpv gpv*)

lemma *gpv-stop-Done* [*simp*]: *gpv-stop* (*Done x*) = *Done* (*Some x*)
⟨*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* ⇒ *Done None* | *Some input'* ⇒ *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* ⇒ *Done None* | *Some x'* ⇒ *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*)) (λ*x*. *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*) (λ*generat*. *case generat of*
Pure x ⇒ *return-spmf* (*x*, *s1*, *s2*)
| *IO out rpv* ⇒ *bind-pmf* (*joint-oracle s1 s2 out*) (λ(*a*, *b*).
case a of None ⇒ *return-pmf None*
| *Some* (*r1*, *s1'*) ⇒ (*case b of None* ⇒ *undefined* | *Some* (*r2*, *s2'*) ⇒
(*case* (*r1*, *r2*) *of* (*None*, *None*) ⇒ *exec-until-stop* (*Done None*) *s1' s2' True*
| (*Some r1'*, *Some r2'*) ⇒ *exec-until-stop* (*rpv r1'*) *s1' s2' True*
| (*None*, *Some r2'*) ⇒ *exec-until-stop* (*Done None*) *s1' s2' True*
| (*Some r1'*, *None*) ⇒ *exec-until-stop* (*rpv r1'*) *s1' s2' False*))))
else
bind-spmf (*the-gpv gpv*) (λ*generat*. *case generat of*
Pure x ⇒ *return-spmf* (*None*, *s1*, *s2*)
| *IO out rpv* ⇒ *bind-spmf* (*callee1 s1 out*) (λ(*r1*, *s1'*).
case r1 of None ⇒ *exec-until-stop* (*Done None*) *s1' s2 False*
| *Some r1'* ⇒ *exec-until-stop* (*rpv r1'*) *s1' s2 False*)))

end

lemma *ord-spmf-exec-gpv-stop*:
fixes *callee1* :: ('*c*, '*r option*, '*s*) *callee*
and *callee2* :: ('*c*, '*r option*, '*s*) *callee*
and *S* :: '*s* ⇒ '*s* ⇒ *bool*
and *gpv* :: ('*a*, '*c*, '*r*) *gpv*
assumes *bisim*:

$\bigwedge s1\ s2\ x.\ [\![\ S\ s1\ s2;\ \neg\ stop\ s2\ ]\!] \Longrightarrow$
ord-spmf $(\lambda(r1, s1')\ (r2, s2').$ *le-option r2 r1* $\wedge$ *S s1' s2'* $\wedge$ $(r2 = None \wedge r1 \neq None \longleftrightarrow stop\ s2'))$
(*callee1 s1 x*) (*callee2 s2 x*)
and *init*: *S s1 s2*
and *go*: $\neg$ *stop s2*
and *sticking*: $\bigwedge s1\ s2\ x\ y\ s1'.\ [\![\ (y, s1') \in$ *set-spmf* (*callee1 s1 x*); *S s1 s2*; *stop s2* $]\!] \Longrightarrow$ *S s1' s2*
shows *ord-spmf* (*rel-prod* (*ord-option* $\top)^{-1-1}$ *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

4.21 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* $(\lambda f'.$ *bind-gpv x* $(\lambda x'.$ *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* $(\lambda x.\ x) \diamond x = x$
⟨*proof*⟩

lemma *ap-gpv-comp*: *pure-gpv* $(\circ) \diamond u \diamond v \diamond w = u \diamond (v \diamond w)$
⟨*proof*⟩

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

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

lemma *exec-gpv-ap*:
exec-gpv callee (*f* ⋄ *x*) *σ* =
exec-gpv callee f σ ⤜ (λ(*f'*, *σ'*). *pure-spmf* (λ(*x'*, *σ''*). (*f' x'*, *σ''*)) ⋄ *exec-gpv callee x σ'*)
⟨*proof*⟩

lemma *exec-gpv-ap-pure* [*simp*]:
exec-gpv callee (*pure-gpv f* ⋄ *x*) *σ* = *pure-spmf* (*apfst f*) ⋄ *exec-gpv callee x σ*
⟨*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 lifting-syntax* **begin**

lift-definition *plus-ℐ* :: ('*out*, '*ret*) *ℐ* ⇒ ('*out'*, '*ret'*) *ℐ* ⇒ ('*out* + '*out'*, '*ret* + '*ret'*) *ℐ* (**infix** ⊕$_{ℐ}$ *500*)
is λ*resp1 resp2*. λ*out*. *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*]: $\mathcal{I}$*-trivial* $(\mathcal{I}_1 \oplus_{\mathcal{I}} \mathcal{I}_2) \longleftrightarrow \mathcal{I}$*-trivial* $\mathcal{I}_1 \wedge \mathcal{I}$*-trivial* $\mathcal{I}_2$
⟨*proof*⟩

end

context
  fixes *left* :: $('s, 'a, 'b)$ *oracle'*
  and *right* :: $('s, 'c, 'd)$ *oracle'*
  and $s :: 's$
begin

primrec *plus-oracle* :: $'a + 'c \Rightarrow (('b + 'd) \times 's)$ *spmf*
where
  plus-oracle (*Inl a*) = *map-spmf* (*apfst Inl*) (*left s a*)
| *plus-oracle* (*Inr b*) = *map-spmf* (*apfst Inr*) (*right s b*)

lemma *lossless-plus-oracleI* [*intro*, *simp*]:
  ⟦ $\bigwedge a.$ *x* = *Inl a* ⟹ *lossless-spmf* (*left s a*);
    $\bigwedge b.$ *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*!]:
  $[\![\mathcal{I}l \vdash_c left\ s\ \surd; \mathcal{I}r \vdash_c right\ s\ \surd]\!] \Longrightarrow \mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash_c (left \oplus_O right)\ s\ \surd$
⟨*proof*⟩

lemma *WT-plus-oracleD1*:
  assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash_c (left \oplus_O right)\ s\ \surd$ (**is** $?\mathcal{I} \vdash_c ?callee\ s\ \surd$)
  shows $\mathcal{I}l \vdash_c left\ s\ \surd$
⟨*proof*⟩

lemma *WT-plus-oracleD2*:
  assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash_c (left \oplus_O right)\ s\ \surd$ (**is** $?\mathcal{I} \vdash_c ?callee\ s\ \surd$)
  shows $\mathcal{I}r \vdash_c right\ s\ \surd$
⟨*proof*⟩

lemma *WT-plus-oracle-iff* [*simp*]: $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash_c (left \oplus_O right)\ s\ \surd \longleftrightarrow \mathcal{I}l \vdash_c left\ s\ \surd \wedge \mathcal{I}r \vdash_c right\ s\ \surd$
⟨*proof*⟩

lemma *callee-invariant-on-plus-oracle* [*simp*]:
  callee-invariant-on (*left* $\oplus_O$ *right*) *I* ($\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r$) $\longleftrightarrow$
   callee-invariant-on left I $\mathcal{I}l \wedge$ *callee-invariant-on right I* $\mathcal{I}r$
   (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *callee-invariant-plus-oracle* [*simp*]:
  callee-invariant (*left* $\oplus_O$ *right*) *I* $\longleftrightarrow$
   callee-invariant left I $\wedge$ *callee-invariant right I*
  (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

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′*);
⋀*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* ⊕*O right1*) *s1 q1*) ((*left2* ⊕*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*]:
⟦ ⋀*a*. *x* = *Inl a* ⟹ *lossless-spmf* (*left s a*);
⋀*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** ⊕*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*]:
⟦ *𝓘l* ⊢*c left sl* √; *𝓘r* ⊢*c right sr* √ ⟧ ⟹ *plus-𝓘 𝓘l 𝓘r* ⊢*c parallel-oracle* (*sl*, *sr*) √
⟨*proof*⟩

lemma *callee-invariant-parallel-oracleI* [*simp*, *intro*]:
assumes *callee-invariant-on left Il 𝓘l callee-invariant-on right Ir 𝓘r*
shows *callee-invariant-on parallel-oracle* (*pred-prod Il Ir*) (*𝓘l* ⊕*𝓘* *𝓘r*)
⟨*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* ⇒ ('*s*, '*a*, '*b*) *oracle*') ⇒ ('*i* ⇒ '*s*, '*i* × '*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* ⇒ ('*call*, '*ret*, '*s*' × '*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

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*⟩

end

7 Cyclic groups

theory *Cyclic-Group* **imports**
HOL−Algebra.Coset
begin

record *'a cyclic-group* = *'a monoid* +
generator :: *'a* (**g**ı)

locale *cyclic-group* = *group G*
for *G* :: (*'a*, *'b*) *cyclic-group-scheme* (**structure**)
+
assumes *generator-closed* [*intro*, *simp*]: *generator G* ∈ *carrier G*
and *generator*: *carrier G* ⊆ *range* (λ*n* :: *nat*. *generator G* [^]$_G$ *n*)

begin

lemma *generatorE* [*elim?*]:
assumes $x \in$ *carrier* G
obtains n :: *nat* **where** $x =$ *generator* G [^] n
⟨*proof*⟩

lemma *inj-on-generator*: *inj-on* (([^]) **g**) {..<*order* G}
⟨*proof*⟩

lemma *carrier-conv-generator*:
finite (*carrier* G) ⟹ *carrier* $G = (\lambda n.$ **g** [^] $n)$ ' {..<*order* G}
⟨*proof*⟩

lemma *bij-betw-generator-carrier*:
finite (*carrier* G) ⟹ *bij-betw* $(\lambda n$ :: *nat*. **g** [^] $n)$ {..<*order* G} (*carrier* G)
⟨*proof*⟩

end

lemma (**in** *monoid*) *order-in-range-Suc*: *order* $G \in$ *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* ⟹ *set-spmf* (*sample-uniform n*) = {..<*n*}
⟨*proof*⟩

lemma (**in** *cyclic-group*) *sample-uniform-one-time-pad*:
  assumes [*simp*]: *c* ∈ *carrier G*
  shows
  map-spmf (λ*x*. **g** [^] *x* ⊗ *c*) (*sample-uniform* (*order G*)) =
   map-spmf (λ*x*. **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.