

CryptHOL

Andreas Lochbihler

February 6, 2026

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 resulting proofs are trustworthy and comprehensible, and the framework is extensible and widely applicable.

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

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

Contents

1	Miscellaneous library additions	4
1.1	HOL	4
1.2	Relations	5
1.3	Pairs	7
1.4	Sums	8
1.5	Option	9
1.5.1	Predicator and relator	10
1.5.2	Orders on option	11
1.5.3	Filter for option	12
1.5.4	Assert for option	12
1.5.5	Join on options	13

1.5.6	Zip on options	13
1.5.7	Binary supremum on <i>'a option</i>	14
1.5.8	Restriction on <i>'a option</i>	15
1.5.9	Maps	16
1.6	Countable	16
1.7	Extended naturals	17
1.8	Extended non-negative reals	18
1.9	BNF material	18
1.10	Transfer and lifting material	21
1.11	Arithmetic	23
1.12	Chain-complete partial orders and <i>partial-function</i>	23
1.13	Folding over finite sets	27
1.14	Parametrisation of transfer rules	27
1.15	Lists	27
1.15.1	List of a given length	28
1.15.2	The type of lists of a given length	29
1.16	Streams and infinite lists	29
1.17	Monomorphic monads	30
1.18	Measures	31
1.19	Sequence space	32
1.20	Probability mass functions	32
1.21	Subprobability mass functions	35
1.21.1	Embedding of <i>'a option</i> into <i>'a spmf</i>	46
1.22	Applicative instance for <i>'a set</i>	48
1.23	Applicative instance for <i>'a spmf</i>	49
1.24	Exclusive or on lists	50
1.25	The environment functor	52
1.26	Setup for <i>partial-function</i> for sets	54
2	Negligibility	57
3	The resumption-error monad	60
3.1	Setup for <i>partial-function</i>	64
3.2	Setup for lifting and transfer	68
4	Generative probabilistic values	69
4.1	Single-step generative	69
4.2	Type definition	74
4.3	Generalised mapper and relator	78
4.4	Simple, derived operations	83
4.5	Monad structure	87
4.6	Embedding <i>'a spmf</i> as a monad	90
4.7	Embedding <i>'a option</i> as a monad	94
4.8	Embedding resumptions	95

4.9	Assertions	96
4.10	Order for $(\text{'a}, \text{'out}, \text{'in}) \text{ gpv}$	98
4.11	Bounds on interaction	99
4.12	Typing	106
	4.12.1 Interface between gpvs and rpvs / callees	106
	4.12.2 Type judgements	115
4.13	Sub-gpvs	118
4.14	Losslessness	119
4.15	Sequencing with failure handling included	126
4.16	Inlining	129
4.17	Running GPVs	141
5	Oracle combinators	155
5.1	Shared state	155
5.2	Shared state with aborts	158
5.3	Disjoint state	158
5.4	Indexed oracles	160
5.5	State extension	160
6	Combining GPVs	163
6.1	Shared state without interrupts	163
6.2	Shared state with interrupts	164
6.3	One-sided shifts	164
6.4	Expectation transformer semantics	169
6.5	Probabilistic termination	175
6.6	Bisimulation for oracles	180
6.7	Applicative instance for $(-, \text{'out}, \text{'in}) \text{ gpv}$	187
7	Cyclic groups	188

1 Miscellaneous library additions

```
theory Misc-CryptHOL imports  
  Probabilistic-While.While-SPMF  
  HOL-Library.Rewrite  
  HOL-Library.Simps-Case-Conv  
  HOL-Library.Type-Length  
  HOL-Eisbach.Eisbach  
  Coinductive.TLList  
  Monad-Normalisation.Monad-Normalisation  
  Monomorphic-Monad.Monomorphic-Monad  
  Applicative-Lifting.Applicative  
begin  
  
hide-const (open) Henstock-Kurzweil-Integration.negligible  
  
declare eq-on-def [simp del]
```

1.1 HOL

```
lemma asm-rl-conv: (PROP P  $\implies$  PROP P)  $\equiv$  Trueprop True  
<proof>
```

```
named-theorems if-distrib Distributivity theorems for If
```

```
lemma if-mono-cong:  $\llbracket b \implies x \leq x'; \neg b \implies y \leq y' \rrbracket \implies \text{If } b \ x \ y \leq \text{If } b \ x' \ y'$   
<proof>
```

```
lemma if-cong-then:  $\llbracket b = b'; b' \implies t = t'; e = e' \rrbracket \implies \text{If } b \ t \ e = \text{If } b' \ t' \ e'$   
<proof>
```

```
lemma if-False-eq:  $\llbracket b \implies \text{False}; e = e' \rrbracket \implies \text{If } b \ t \ e = e'$   
<proof>
```

```
lemma imp-OO-imp [simp]:  $(\longrightarrow) \text{ OO } (\longrightarrow) = (\longrightarrow)$   
<proof>
```

```
lemma inj-on-fun-updD:  $\llbracket \text{inj-on } (f(x := y)) \ A; x \notin A \rrbracket \implies \text{inj-on } f \ A$   
<proof>
```

```
lemma disjoint-notin1:  $\llbracket A \cap B = \{\}; x \in B \rrbracket \implies x \notin A$  <proof>
```

```
lemma Least-le-Least:  
  fixes x :: 'a :: wellorder  
  assumes Q x  
  and Q:  $\bigwedge x. Q \ x \implies \exists y \leq x. P \ y$   
  shows Least P  $\leq$  Least Q  
  <proof>
```

```
lemma is-empty-image [simp]: Set.is-empty (f ' A) = Set.is-empty A
```

$\langle proof \rangle$

1.2 Relations

inductive *Imagep* :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b \Rightarrow bool
for *R P*

where *ImageI*: $\llbracket P\ x; R\ x\ y \rrbracket \Longrightarrow Imagep\ R\ P\ y$

lemma *r-r-into-tranclp*: $\llbracket r\ x\ y; r\ y\ z \rrbracket \Longrightarrow r^{++}\ x\ z$
 $\langle proof \rangle$

lemma *transp-tranclp-id*:

assumes *transp R*

shows *tranclp R = R*

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *Domainp-conversep*: *Domainp R*⁻¹⁻¹ = *Rangep R*

$\langle proof \rangle$

lemma *bi-unique-rel-set-bij-betw*:

assumes *unique: bi-unique R*

and *rel: rel-set R A B*

shows $\exists f.\ bij\ betw\ f\ A\ B \wedge (\forall x \in A.\ R\ x\ (f\ x))$

$\langle proof \rangle$

definition *restrict-relp* :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool

($\langle - \mid (- \otimes -) \rangle$ [53, 54, 54] 53)

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

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

$\langle proof \rangle$

lemma *restrict-relpI* [*intro?*]: $\llbracket R\ x\ y; P\ x; Q\ y \rrbracket \Longrightarrow (R \mid P \otimes Q)\ x\ y$

$\langle proof \rangle$

lemma *restrict-relpE* [*elim?*, *cases pred*]:

assumes (*R* \mid *P* \otimes *Q*) *x y*

obtains (*restrict-relp*) *R x y P x Q y*

$\langle proof \rangle$

lemma *conversep-restrict-relp* [*simp*]: (*R* \mid *P* \otimes *Q*)⁻¹⁻¹ = *R*⁻¹⁻¹ \mid *Q* \otimes *P*

$\langle proof \rangle$

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

<proof>

lemma *restrict-relp-cong*:

$\llbracket P = P'; Q = Q'; \bigwedge x y. \llbracket P x; Q y \rrbracket \implies R x y = R' x y \rrbracket \implies R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$

<proof>

lemma *restrict-relp-cong-simp*:

$\llbracket P = P'; Q = Q'; \bigwedge x y. P x = \text{simp} \implies Q y = \text{simp} \implies R x y = R' x y \rrbracket \implies R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$

<proof>

lemma *restrict-relp-parametric* [*transfer-rule*]:

includes *lifting-syntax* **shows**

$((A \implies B \implies (=)) \implies (A \implies (=)) \implies (B \implies (=)) \implies A \implies B \implies (=))$ *restrict-relp restrict-relp*

<proof>

lemma *restrict-relp-mono*: $\llbracket R \leq R'; P \leq P'; Q \leq Q' \rrbracket \implies R \upharpoonright P \otimes Q \leq R' \upharpoonright P' \otimes Q'$

<proof>

lemma *restrict-relp-mono'*:

$\llbracket (R \upharpoonright P \otimes Q) x y; \llbracket R x y; P x; Q y \rrbracket \implies R' x y \ \&\&\& \ P' x \ \&\&\& \ Q' y \rrbracket \implies (R' \upharpoonright P' \otimes Q') x y$

<proof>

lemma *restrict-relp-DomainpD*: $\text{Domainp } (R \upharpoonright P \otimes Q) x \implies \text{Domainp } R x \wedge P x$

<proof>

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

<proof>

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

<proof>

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

<proof>

definition *rel-prod2* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow ('c \times 'b) \Rightarrow \text{bool}$

where *rel-prod2* $R a = (\lambda(c, b). R a b)$

lemma *rel-prod2-simps* [*simp*]: $\text{rel-prod2 } R a (c, b) \longleftrightarrow R a b$

<proof>

lemma *restrict-rel-prod*:

$\text{rel-prod } (R \upharpoonright I1 \otimes I2) (S \upharpoonright I1' \otimes I2') = \text{rel-prod } R S \upharpoonright \text{pred-prod } I1 I1' \otimes \text{pred-prod } I2 I2'$

<proof>

lemma *restrict-rel-prod1*:

$rel\text{-}prod (R \upharpoonright I1 \otimes I2) S = rel\text{-}prod R S \upharpoonright pred\text{-}prod I1 (\lambda\cdot. True) \otimes pred\text{-}prod I2 (\lambda\cdot. True)$

<proof>

lemma *restrict-rel-prod2*:

$rel\text{-}prod R (S \upharpoonright I1 \otimes I2) = rel\text{-}prod R S \upharpoonright pred\text{-}prod (\lambda\cdot. True) I1 \otimes pred\text{-}prod (\lambda\cdot. True) I2$

<proof>

consts *relcompp-witness* :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('b \Rightarrow 'c \Rightarrow bool) \Rightarrow 'a \times 'c \Rightarrow 'b

specification (*relcompp-witness*)

$relcompp\text{-}witness1: (A \text{ OO } B) (fst\ xy) (snd\ xy) \Longrightarrow A (fst\ xy) (relcompp\text{-}witness\ A\ B\ xy)$

$relcompp\text{-}witness2: (A \text{ OO } B) (fst\ xy) (snd\ xy) \Longrightarrow B (relcompp\text{-}witness\ A\ B\ xy) (snd\ xy)$

<proof>

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

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

lemma *relcompp-witness-eq* [*simp*]: *relcompp-witness* (=) (=) (x, x) = x

<proof>

1.3 Pairs

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

<proof>

definition *corec-prod* :: ('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow 'b) \Rightarrow 's \Rightarrow 'a \times 'b

where *corec-prod* f g = ($\lambda s. (f\ s, g\ s)$)

lemma *corec-prod-apply*: *corec-prod* f g s = (f s, g s)

<proof>

lemma *corec-prod-sel* [*simp*]:

shows *fst-corec-prod*: *fst* (*corec-prod* f g s) = f s

and *snd-corec-prod*: *snd* (*corec-prod* f g s) = g s

<proof>

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

<proof>

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

s

<proof>

lemma *map-corec-prod* [*simp*]: $\text{map-prod } f \ g \ (\text{corec-prod } h \ k \ s) = \text{corec-prod } (f \circ h) \ (g \circ k) \ s$
<proof>

lemma *split-corec-prod* [*simp*]: $\text{case-prod } h \ (\text{corec-prod } f \ g \ s) = h \ (f \ s) \ (g \ s)$
<proof>

lemma *Pair-fst-Unity*: $(\text{fst } x, ()) = x$
<proof>

definition *rprodl* :: $('a \times 'b) \times 'c \Rightarrow 'a \times ('b \times 'c)$ **where** $rprodl = (\lambda((a, b), c). (a, (b, c)))$

lemma *rprodl-simps* [*simp*]: $rprodl \ ((a, b), c) = (a, (b, c))$
<proof>

lemma *rprodl-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 $(\text{rel-prod } (\text{rel-prod } A \ B) \ C) ==> \text{rel-prod } A \ (\text{rel-prod } B \ C)$ *rprodl rprodl*
<proof>

definition *lprodr* :: $'a \times ('b \times 'c) \Rightarrow ('a \times 'b) \times 'c$ **where** $lprodr = (\lambda(a, b), c). ((a, b), c)$

lemma *lprodr-simps* [*simp*]: $lprodr \ (a, b, c) = ((a, b), c)$
<proof>

lemma *lprodr-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 $(\text{rel-prod } A \ (\text{rel-prod } B \ C)) ==> \text{rel-prod } (\text{rel-prod } A \ B) \ C$ *lprodr lprodr*
<proof>

lemma *lprodr-inverse* [*simp*]: $rprodl \ (lprodr \ x) = x$
<proof>

lemma *rprodl-inverse* [*simp*]: $lprodr \ (rprodl \ x) = x$
<proof>

lemma *pred-prod-mono'* [*mono*]:
 $\text{pred-prod } A \ B \ xy \longrightarrow \text{pred-prod } A' \ B' \ xy$
if $\bigwedge x. A \ x \longrightarrow A' \ x \ \bigwedge y. B \ y \longrightarrow B' \ y$
<proof>

fun *rel-witness-prod* :: $('a \times 'b) \times ('c \times 'd) \Rightarrow (('a \times 'c) \times ('b \times 'd))$ **where**
 $\text{rel-witness-prod } ((a, b), (c, d)) = ((a, c), (b, d))$

1.4 Sums

lemma *isLE*:

assumes *isl* *x*
obtains *l* **where** $x = \text{Inl } l$
 ⟨*proof*⟩

lemma *Inl-in-Plus* [*simp*]: $\text{Inl } x \in A <+> B \longleftrightarrow x \in A$
 ⟨*proof*⟩

lemma *Inr-in-Plus* [*simp*]: $\text{Inr } x \in A <+> B \longleftrightarrow x \in B$
 ⟨*proof*⟩

lemma *Inl-eq-map-sum-iff*: $\text{Inl } x = \text{map-sum } f \ g \ y \longleftrightarrow (\exists z. y = \text{Inl } z \wedge x = f \ z)$
 ⟨*proof*⟩

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

lemma *inj-on-map-sum* [*simp*]:
 $\llbracket \text{inj-on } f \ A; \text{inj-on } g \ B \rrbracket \implies \text{inj-on } (\text{map-sum } f \ g) (A <+> B)$
 ⟨*proof*⟩

lemma *inv-into-map-sum*:
 $\text{inv-into } (A <+> B) (\text{map-sum } f \ g) \ x = \text{map-sum } (\text{inv-into } A \ f) (\text{inv-into } B \ g) \ x$
if $x \in f \ 'A <+> g \ 'B$ *inj-on* *f* *A* *inj-on* *g* *B*
 ⟨*proof*⟩

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

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

lemma *rsuml-lsumr* [*simp*]: $\text{rsuml } (\text{lsumr } x) = x$
 ⟨*proof*⟩

lemma *lsumr-rsuml* [*simp*]: $\text{lsumr } (\text{rsuml } x) = x$
 ⟨*proof*⟩

1.5 Option

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

lemma *case-option-collapse*: $\text{case-option } x \ (\lambda-. \ x) \ y = x$
 ⟨*proof*⟩

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

1.5.1 Predicate and relator

lemma *option-pred-mono-strong*:

[[$\text{pred-option } P x; \bigwedge a. \llbracket a \in \text{set-option } x; P a \rrbracket \implies P' a \rrbracket \implies \text{pred-option } P' x$]]
 ⟨proof⟩

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

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

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

lemma *pred-option-conj* [simp]:

$\text{pred-option } (\lambda x. P x \wedge Q x) = (\lambda x. \text{pred-option } P x \wedge \text{pred-option } Q x)$
 ⟨proof⟩

lemma *pred-option-top* [simp]:

$\text{pred-option } (\lambda-. \text{True}) = (\lambda-. \text{True})$
 ⟨proof⟩

lemma *rel-option-restrict-relpI* [intro?]:

[[$\text{rel-option } R x y; \text{pred-option } P x; \text{pred-option } Q y \rrbracket \implies \text{rel-option } (R \upharpoonright P \otimes Q) x y$]]
 ⟨proof⟩

lemma *rel-option-restrict-relpE* [elim?]:

assumes $\text{rel-option } (R \upharpoonright P \otimes Q) x y$
obtains $\text{rel-option } R x y \text{ pred-option } P x \text{ pred-option } Q y$
 ⟨proof⟩

lemma *rel-option-restrict-relp-iff*:

$\text{rel-option } (R \upharpoonright P \otimes Q) x y \iff \text{rel-option } R x y \wedge \text{pred-option } P x \wedge \text{pred-option } Q y$
 ⟨proof⟩

lemma *option-rel-map-restrict-relp*:

shows *option-rel-map-restrict-relp1*:

$\text{rel-option } (R \upharpoonright P \otimes Q) (\text{map-option } f x) = \text{rel-option } (R \circ f \upharpoonright P \circ f \otimes Q) x$

and *option-rel-map-restrict-relp2*:

$\text{rel-option } (R \upharpoonright P \otimes Q) x (\text{map-option } g y) = \text{rel-option } ((\lambda x. R x \circ g) \upharpoonright P \otimes Q)$

$\circ g$) $x y$
(proof)

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

lemma *rel-witness-option*:
 shows *set-rel-witness-option*: $\llbracket \text{rel-option } A \ x \ y; (a, b) \in \text{set-option } (\text{rel-witness-option } (x, y)) \rrbracket \Longrightarrow A \ a \ b$
 and *map1-rel-witness-option*: $\text{rel-option } A \ x \ y \Longrightarrow \text{map-option } \text{fst } (\text{rel-witness-option } (x, y)) = x$
 and *map2-rel-witness-option*: $\text{rel-option } A \ x \ y \Longrightarrow \text{map-option } \text{snd } (\text{rel-witness-option } (x, y)) = y$
 (proof)

lemma *rel-witness-option1*:
 assumes *rel-option* $A \ x \ y$
 shows *rel-option* $(\lambda a \ (a', b). a = a' \wedge A \ a' \ b) \ x \ (\text{rel-witness-option } (x, y))$
 (proof)

lemma *rel-witness-option2*:
 assumes *rel-option* $A \ x \ y$
 shows *rel-option* $(\lambda(a, b') \ b. b = b' \wedge A \ a \ b') \ (\text{rel-witness-option } (x, y)) \ y$
 (proof)

1.5.2 Orders on option

abbreviation *le-option* :: 'a option \Rightarrow 'a option \Rightarrow bool
where *le-option* \equiv *ord-option* (=)

lemma *le-option-bind-mono*:
 $\llbracket \text{le-option } x \ y; \bigwedge a. a \in \text{set-option } x \Longrightarrow \text{le-option } (f \ a) \ (g \ a) \rrbracket$
 $\Longrightarrow \text{le-option } (\text{Option.bind } x \ f) \ (\text{Option.bind } y \ g)$
 (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 \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b option \Rightarrow bool
where *pcr-Some* $R \ x \ y \longleftrightarrow (\exists z. y = \text{Some } z \wedge R \ x \ z)$

lemma *pcr-Some-simps* [simp]: *pcr-Some* $R \ x \ (\text{Some } y) \longleftrightarrow R \ x \ y$
(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* \Rightarrow *bool*) \Rightarrow *'a option* \Rightarrow *'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 \in set-option x. P y}*
<proof>

lemma *filter-map-option*: *filter-option P (map-option f x) = map-option f (filter-option (P \circ f) x)*
<proof>

lemma *is-none-filter-option* [*simp*]: *Option.is-none (filter-option P x) \longleftrightarrow Option.is-none x \vee \neg P (the x)*
<proof>

lemma *filter-option-eq-Some-iff* [*simp*]: *filter-option P x = Some y \longleftrightarrow x = Some y \wedge P y*
<proof>

lemma *Some-eq-filter-option-iff* [*simp*]: *Some y = filter-option P x \longleftrightarrow x = Some y \wedge 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* \Rightarrow *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 \in set-option (assert-option b) \longleftrightarrow b*
<proof>

1.5.5 Join on options

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

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

lemma *set-join-option* [*simp*]: *set-option* (*join-option* x) = \bigcup (*set-option* ' *set-option* x)
<proof>

lemma *in-set-join-option*: x \in *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) \implies *rel-option* R) *join-option join-option*
<proof>

lemma *join-option-eq-Some* [*simp*]: *join-option* x = Some y \longleftrightarrow x = Some (Some y)
<proof>

lemma *Some-eq-join-option* [*simp*]: Some y = *join-option* x \longleftrightarrow x = Some (Some y)
<proof>

lemma *join-option-eq-None*: *join-option* x = None \longleftrightarrow x = None \vee x = Some None
<proof>

lemma *None-eq-join-option*: None = *join-option* x \longleftrightarrow x = None \vee x = Some None
<proof>

1.5.6 Zip on options

function *zip-option* :: 'a option \Rightarrow 'b option \Rightarrow ('a \times '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) \longleftrightarrow x = Some a \wedge y = Some b
 <proof>

lemma *set-zip-option* [simp]:

set-option (zip-option x y) = set-option x \times 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 (λx . Option.bind y (λy . Some (x, y)))
 <proof>

lemma *zip-option-parametric* [transfer-rule]:

includes *lifting-syntax shows*
(rel-option R \implies rel-option Q \implies 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 \Rightarrow 'a option \Rightarrow '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*: $\text{sup-option } (\text{sup-option } x \ y) \ z = \text{sup-option } x \ (\text{sup-option } y \ z)$
 ⟨proof⟩

lemma *sup-option-left-idem*: $\text{sup-option } x \ (\text{sup-option } x \ y) = \text{sup-option } x \ y$
 ⟨proof⟩

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

lemma *sup-option-None* [simp]: $\text{sup-option } \text{None } y = y$
 ⟨proof⟩

1.5.8 Restriction on 'a option

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

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

lemma *enforce-map-option*: $\text{enforce-option } P \ (\text{map-option } f \ x) = \text{map-option } f \ (\text{enforce-option } (P \circ f) \ x)$
 ⟨proof⟩

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

lemma *enforce-option-alt-def*:
 $\text{enforce-option } P \ x = \text{Option.bind } x \ (\lambda a. \ \text{Option.bind } (\text{assert-option } (P \ a)) \ (\lambda - :: \text{unit}. \ \text{Some } a))$
 ⟨proof⟩

lemma *enforce-option-eq-None-iff* [simp]:
 $\text{enforce-option } P \ x = \text{None} \iff (\forall a. \ x = \text{Some } a \longrightarrow \neg P \ a)$
 ⟨proof⟩

lemma *enforce-option-eq-Some-iff* [simp]:
 $\text{enforce-option } P \ x = \text{Some } y \iff x = \text{Some } y \wedge P \ y$
 ⟨proof⟩

lemma *Some-eq-enforce-option-iff* [simp]:
 $\text{Some } y = \text{enforce-option } P \ x \iff x = \text{Some } y \wedge P \ y$
 ⟨proof⟩

lemma *enforce-option-top* [simp]: $\text{enforce-option } \top = \text{id}$
 ⟨proof⟩

lemma *enforce-option-K-True* [simp]: *enforce-option* ($\lambda\cdot$. *True*) $x = x$
 ⟨*proof*⟩

lemma *enforce-option-bot* [simp]: *enforce-option* $\perp = (\lambda\cdot$. *None*)
 ⟨*proof*⟩

lemma *enforce-option-K-False* [simp]: *enforce-option* ($\lambda\cdot$. *False*) $x = \text{None}$
 ⟨*proof*⟩

lemma *enforce-pred-id-option*: *pred-option* $P x \implies \text{enforce-option } P x = x$
 ⟨*proof*⟩

1.5.9 Maps

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

lemma *map-le-map-upd2*: $\llbracket f \subseteq_m g; \bigwedge y'. f x = \text{Some } y' \implies y' = y \rrbracket \implies f \subseteq_m g(x \mapsto y)$
 ⟨*proof*⟩

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

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

lemma *dom-subset-ran-iff*:
assumes *finite* (*ran* m)
shows $\text{dom } m \subseteq \text{ran } m \iff \text{dom } m = \text{ran } m$
 ⟨*proof*⟩

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

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

lemma *map-le-SomeID*: $\llbracket m \subseteq_m m'; m x = \text{Some } y \rrbracket \implies m' x = \text{Some } y$
 ⟨*proof*⟩

lemma *map-le-fun-upd2*: $\llbracket f \subseteq_m g; x \notin \text{dom } f \rrbracket \implies f \subseteq_m g(x := y)$
 ⟨*proof*⟩

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

1.6 Countable

lemma *countable-lfp*:

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

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

1.7 Extended naturals

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

lemma *eSuc-SUP*: $A \neq \{\} \implies e\text{Suc } (\bigsqcup (f \text{ ' } A)) = (\bigsqcup x \in A. e\text{Suc } (f x))$
 ⟨*proof*⟩

lemma *ereal-of-enat-1*: *ereal-of-enat* $1 = \text{ereal } 1$
 ⟨*proof*⟩

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

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

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

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

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

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

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

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

1.8 Extended non-negative reals

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

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

lemma *nn-integral-indicator-map*:

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

1.9 BNF material

lemma *transp-rel-fun*: $\llbracket \text{is-equality } Q; \text{transp } R \rrbracket \implies \text{transp } (\text{rel-fun } Q\ R)$
 ⟨proof⟩

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

lemma *reflp-fun1*: **includes** *lifting-syntax* **shows** $\llbracket \text{is-equality } A; \text{reflp } B \rrbracket \implies \text{reflp } (A \text{ ===> } B)$
 ⟨proof⟩

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

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

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

lemma *rel-fun-Grp-copy-Abs*:
includes *lifting-syntax*
assumes *type-definition* $\text{Rep Abs } A$
shows $\text{rel-fun } (\text{BNF-Def.Grp } A\ \text{Abs}) (\text{BNF-Def.Grp } B\ g) = \text{BNF-Def.Grp } \{f. f - ' A \subseteq B\} (\text{Rep } \text{---->} g)$
 ⟨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 $R = (\text{BNF-Def.Grp } \{x. x \subseteq \{(x, y). R x y\}\} ((\cdot) \text{fst}))^{-1-1} \text{ OO } \text{BNF-Def.Grp } \{x. x \subseteq \{(x, y). R x y\}\} ((\cdot) \text{snd})$
<proof>

lemma *Domainp-Grp*: *Domainp* (BNF-Def.Grp $A f$) = ($\lambda x. x \in A$)

<proof>

lemma *pred-prod-conj [simp]*:

shows *pred-prod-conj1*: $\bigwedge P Q R. \text{pred-prod } (\lambda x. P x \wedge Q x) R = (\lambda x. \text{pred-prod } P R x \wedge \text{pred-prod } Q R x)$

and *pred-prod-conj2*: $\bigwedge P Q R. \text{pred-prod } P (\lambda x. Q x \wedge R x) = (\lambda x. \text{pred-prod } P Q x \wedge \text{pred-prod } P R x)$

<proof>

lemma *pred-sum-conj [simp]*:

shows *pred-sum-conj1*: $\bigwedge P Q R. \text{pred-sum } (\lambda x. P x \wedge Q x) R = (\lambda x. \text{pred-sum } P R x \wedge \text{pred-sum } Q R x)$

and *pred-sum-conj2*: $\bigwedge P Q R. \text{pred-sum } P (\lambda x. Q x \wedge R x) = (\lambda x. \text{pred-sum } P Q x \wedge \text{pred-sum } P R x)$

<proof>

lemma *pred-list-conj [simp]*: *list-all* ($\lambda x. P x \wedge Q x$) = ($\lambda x. \text{list-all } P x \wedge \text{list-all } Q x$)

<proof>

lemma *pred-prod-top [simp]*:

pred-prod ($\lambda-. \text{True}$) ($\lambda-. \text{True}$) = ($\lambda-. \text{True}$)

<proof>

lemma *rel-fun-conversep*: **includes** *lifting-syntax* **shows**

$(A^{\hat{\ }--1} ==> B^{\hat{\ }--1}) = (A ==> B)^{\hat{\ }--1}$

<proof>

lemma *left-unique-Grp [iff]*:

left-unique (BNF-Def.Grp $A f$) \longleftrightarrow *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 $A f$) \longleftrightarrow *inj-on* $f A$

<proof>

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

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

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

lemma *left-unique-vimage2p* [simp]:
 $\llbracket \text{left-unique } P; \text{inj } f \rrbracket \Longrightarrow \text{left-unique } (\text{BNF-Def.vimage2p } f g P)$
 ⟨proof⟩

lemma *right-unique-vimage2p* [simp]:
 $\llbracket \text{right-unique } P; \text{inj } g \rrbracket \Longrightarrow \text{right-unique } (\text{BNF-Def.vimage2p } f g P)$
 ⟨proof⟩

lemma *bi-unique-vimage2p* [simp]:
 $\llbracket \text{bi-unique } P; \text{inj } f; \text{inj } g \rrbracket \Longrightarrow \text{bi-unique } (\text{BNF-Def.vimage2p } f g P)$
 ⟨proof⟩

lemma *left-total-vimage2p* [simp]:
 $\llbracket \text{left-total } P; \text{surj } g \rrbracket \Longrightarrow \text{left-total } (\text{BNF-Def.vimage2p } f g P)$
 ⟨proof⟩

lemma *right-total-vimage2p* [simp]:
 $\llbracket \text{right-total } P; \text{surj } f \rrbracket \Longrightarrow \text{right-total } (\text{BNF-Def.vimage2p } f g P)$
 ⟨proof⟩

lemma *bi-total-vimage2p* [simp]:
 $\llbracket \text{bi-total } P; \text{surj } f; \text{surj } g \rrbracket \Longrightarrow \text{bi-total } (\text{BNF-Def.vimage2p } f g P)$
 ⟨proof⟩

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

lemma *vimage2p-conversep*: $\text{BNF-Def.vimage2p } f g R^{\hat{-} - 1} = (\text{BNF-Def.vimage2p } g f R)^{\hat{-} - 1}$
 ⟨proof⟩

lemma *rel-fun-refl*: $\llbracket A \leq (=); (=) \leq B \rrbracket \Longrightarrow (=) \leq \text{rel-fun } A B$
 ⟨proof⟩

lemma *rel-fun-mono-strong*:
 $\llbracket \text{rel-fun } A B f g; A' \leq A; \bigwedge x y. \llbracket x \in f \text{ ' } \{x. \text{Domainp } A' x\}; y \in g \text{ ' } \{x. \text{Rangep}$

$A' x \}; B x y \]] \implies B' x y \]] \implies \text{rel-fun } A' B' f g$
 ⟨proof⟩

lemma *rel-fun-refl-strong*:

assumes $A \leq (=) \wedge x. x \in f' \{x. \text{Domainp } A x\} \implies B x x$

shows $\text{rel-fun } A B f f$

⟨proof⟩

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

lemma *Rangep-Grp*: $\text{Rangep } (\text{BNF-Def.Grp } A f) = (\lambda x. x \in f' A)$

⟨proof⟩

lemma *rel-fun-Grp*:

$\text{rel-fun } (\text{BNF-Def.Grp } \text{UNIV } h)^{-1-1} (\text{BNF-Def.Grp } A g) = \text{BNF-Def.Grp } \{f. f$
 ‘ $\text{range } h \subseteq A\}$ (map-fun $h g$)

⟨proof⟩

1.10 Transfer and lifting material

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

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

assumes [transfer-rule]: *bi-total* A

shows $((A \implies A \implies (=)) \implies (B \implies B \implies (=)) \implies (A \implies B) \implies (=))$ *monotone monotone*

⟨proof⟩

lemma *fun-ord-parametric* [transfer-rule]:

assumes [transfer-rule]: *bi-total* C

shows $((A \implies B \implies (=)) \implies (C \implies A) \implies (C \implies B) \implies (=))$ *fun-ord fun-ord*

⟨proof⟩

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

$(\text{rel-set } A \implies \text{rel-set } B \implies \text{rel-set } (\text{rel-sum } A B)) (<+>) (<+>)$

⟨proof⟩

lemma *pred-fun-parametric* [transfer-rule]:

assumes [transfer-rule]: *bi-total* A

shows $((A \implies (=)) \implies (B \implies (=)) \implies (A \implies B) \implies (=))$ *pred-fun pred-fun*

⟨proof⟩

lemma *rel-fun-eq-OO*: $((=) \implies A) \text{ OO } ((=) \implies B) = ((=) \implies A \text{ OO } B)$

⟨proof⟩

end

lemma *Quotient-set-rel-eq*:
includes *lifting-syntax*
assumes *Quotient R Abs Rep T*
shows $(\text{rel-set } T \text{ } \text{====>} \text{ rel-set } T \text{ } \text{====>} (=)) (\text{rel-set } R) (=)$
 $\langle \text{proof} \rangle$

lemma *Domainp-eq*: $\text{Domainp } (=) = (\lambda \cdot. \text{True})$
 $\langle \text{proof} \rangle$

lemma *rel-fun-eq-onpI*: $\text{eq-onp } (\text{pred-fun } P \ Q) \ f \ g \implies \text{rel-fun } (\text{eq-onp } P) (\text{eq-onp } Q) \ f \ g$
 $\langle \text{proof} \rangle$

lemma *bi-unique-eq-onp*: $\text{bi-unique } (\text{eq-onp } P)$
 $\langle \text{proof} \rangle$

lemma *rel-fun-eq-conversep*: **includes** *lifting-syntax* **shows** $(A^{-1-1} \text{ } \text{====>} (=)) = (A \text{ } \text{====>} (=))^{-1-1}$
 $\langle \text{proof} \rangle$

lemma *rel-fun-comp*:
 $\bigwedge f \ g \ h. \text{rel-fun } A \ B \ (f \circ g) \ h = \text{rel-fun } A \ (\lambda x. B \ (f \ x)) \ g \ h$
 $\bigwedge f \ g \ h. \text{rel-fun } A \ B \ f \ (g \circ h) = \text{rel-fun } A \ (\lambda x \ y. B \ x \ (g \ y)) \ f \ h$
 $\langle \text{proof} \rangle$

lemma *rel-fun-map-fun1*: $\text{rel-fun } (\text{BNF-Def.Grp } UNIV \ h)^{-1-1} \ A \ f \ g \implies \text{rel-fun } (=) \ A \ (\text{map-fun } h \ id \ f) \ g$
 $\langle \text{proof} \rangle$

lemma *map-fun2-id*: $\text{map-fun } f \ g \ x = g \circ \text{map-fun } f \ id \ x$
 $\langle \text{proof} \rangle$

lemma *map-fun-id2-in*: $\text{map-fun } g \ h \ f = \text{map-fun } g \ id \ (h \circ f)$
 $\langle \text{proof} \rangle$

lemma *Domainp-rel-fun-le*: $\text{Domainp } (\text{rel-fun } A \ B) \leq \text{pred-fun } (\text{Domainp } A) (\text{Domainp } B)$
 $\langle \text{proof} \rangle$

definition *rel-witness-fun* :: $(\ 'a \Rightarrow \ 'b \Rightarrow \ \text{bool}) \Rightarrow (\ 'b \Rightarrow \ 'c \Rightarrow \ \text{bool}) \Rightarrow (\ 'a \Rightarrow \ 'd) \times (\ 'c \Rightarrow \ 'e) \Rightarrow (\ 'b \Rightarrow \ 'd \times \ 'e)$ **where**
 $\text{rel-witness-fun } A \ A' = (\lambda (f, g) \ b. (f \ (\text{THE } a. A \ a \ b), g \ (\text{THE } c. A' \ b \ c)))$

lemma
assumes *fg*: $\text{rel-fun } (A \ OO \ A') \ B \ f \ g$
and *A*: *left-unique A right-total A*
and *A'*: *right-unique A' left-total A'*
shows *rel-witness-fun1*: $\text{rel-fun } A \ (\lambda x \ (x', y). x = x' \wedge B \ x' \ y) \ f \ (\text{rel-witness-fun } A \ A' \ B \ f \ g)$

$A A' (f, g)$
and *rel-witness-fun2*: *rel-fun* $A' (\lambda(x, y') y. y = y' \wedge B x y')$ (*rel-witness-fun*
 $A A' (f, g)$) g
 $\langle proof \rangle$

lemma *rel-witness-fun-eq* [*simp*]: *rel-witness-fun* $(=)$ $(=)$ $(f, g) = (\lambda x. (f x, g x))$
 $\langle proof \rangle$

1.11 Arithmetic

lemma *abs-diff-triangle-ineq2*: $|a - b| + |c - b| \leq |a - c|$
 $\langle proof \rangle$

lemma (**in** *ordered-ab-semigroup-add*) *add-left-mono-trans*:
 $\llbracket x \leq a + b; b \leq c \rrbracket \implies x \leq a + c$
 $\langle proof \rangle$

lemma *of-nat-le-one-cancel-iff* [*simp*]:
fixes $n :: nat$ **shows** *real* $n \leq 1 \iff n \leq 1$
 $\langle proof \rangle$

lemma (**in** *linordered-semidom*) *mult-right-le*: $c \leq 1 \implies 0 \leq a \implies c * a \leq a$
 $\langle proof \rangle$

1.12 Chain-complete partial orders and partial-function

lemma *fun-ordD*: *fun-ord* $ord f g \implies ord (f x) (g x)$
 $\langle proof \rangle$

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*) $(\lambda x. P (fst x)$
 $(snd x))$
and *f*: *monotone* *orda* *orda* *f*
and *g*: *monotone* *ordb* *ordb* *g*
and *bot*: $P (luba \{\}) (lubb \{\})$
and *step*: $\bigwedge x y. \llbracket orda x (ccpo.fixp luba orda f); ordb y (ccpo.fixp lubb ordb g); P$
 $x y \rrbracket \implies P (f x) (g y)$
shows $P (ccpo.fixp luba orda f) (ccpo.fixp lubb ordb g)$
 $\langle proof \rangle$

lemma *parallel-fixp-induct-strong-uc*:
assumes *a*: *partial-function-definitions* *orda* *luba*
and *b*: *partial-function-definitions* *ordb* *lubb*
and *F*: $\bigwedge x. monotone (fun-ord orda) orda (\lambda f. U1 (F (C1 f)) x)$
and *G*: $\bigwedge y. monotone (fun-ord ordb) ordb (\lambda g. U2 (G (C2 g)) y)$
and *eq1*: $f \equiv C1 (ccpo.fixp (fun-lub luba) (fun-ord orda) (\lambda f. U1 (F (C1 f))))$
and *eq2*: $g \equiv C2 (ccpo.fixp (fun-lub lubb) (fun-ord ordb) (\lambda g. U2 (G (C2 g))))$

and inverse: $\bigwedge f. U1 (C1 f) = f$
and inverse2: $\bigwedge g. U2 (C2 g) = g$
and adm: $ccpo.admissible (prod-lub (fun-lub luba) (fun-lub lubb)) (rel-prod (fun-ord orda) (fun-ord ordb)) (\lambda x. P (fst x) (snd x))$
and bot: $P (\lambda-. luba \{\}) (\lambda-. lubb \{\})$
and step: $\bigwedge f' g'. \llbracket \bigwedge x. orda (U1 f' x) (U1 f x); \bigwedge y. ordb (U2 g' y) (U2 g y); P (U1 f') (U2 g') \rrbracket \implies P (U1 (F f')) (U2 (G g'))$
shows $P (U1 f) (U2 g)$
 $\langle proof \rangle$

lemmas parallel-fixp-induct-strong-1-1 = parallel-fixp-induct-strong-uc
of - - - - $\lambda x. x - \lambda x. x \lambda x. x - \lambda x. x,$
OF - - - - - *refl refl*]

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

lemma fixp-induct-option': — Stronger induction rule
fixes $F :: 'c \Rightarrow 'c$ **and**
 $U :: 'c \Rightarrow 'b \Rightarrow 'a$ **option and**
 $C :: ('b \Rightarrow 'a \text{ option}) \Rightarrow 'c$ **and**
 $P :: 'b \Rightarrow 'a \Rightarrow bool$
assumes *mono:* $\bigwedge x. mono-option (\lambda f. U (F (C f)) x)$
assumes *eq:* $f \equiv C (ccpo.fixp (fun-lub (flat-lub None)) (fun-ord option-ord) (\lambda f. U (F (C f))))$
assumes *inverse2:* $\bigwedge f. U (C f) = f$
assumes *step:* $\bigwedge g x y. \llbracket \bigwedge x y. U g x = Some y \implies P x y; U (F g) x = Some y; \bigwedge x. option-ord (U g x) (U f x) \rrbracket \implies P x y$
assumes *defined:* $U f x = Some y$
shows $P x y$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma bot-fun-least [simp]: $(\lambda-. bot :: 'a :: order-bot) \leq x$
 $\langle proof \rangle$

lemma fun-ord-conv-rel-fun: $fun-ord = rel-fun (=)$
 $\langle proof \rangle$

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

lemma *finite-chainsD*: $\llbracket \text{finite-chains ord}; \text{Complete-Partial-Order.chain ord } Y \rrbracket$
 $\implies \text{finite } Y$
 $\langle \text{proof} \rangle$

lemma *finite-chains-flat-ord* [*simp, intro!*]: *finite-chains (flat-ord x)*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *rel-fun-curry: includes lifting-syntax shows*
 $(A \implies B \implies C) f g \longleftrightarrow (\text{rel-prod } A \ B \implies C) (\text{case-prod } f) (\text{case-prod } g)$
 $\langle \text{proof} \rangle$

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 $\text{Sup } (f \text{ ` } A) \leq (f (\text{luba } A))$
 $\langle \text{proof} \rangle$

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

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: $\bigwedge x. \llbracket x \leq \text{ccpo-class.fixp } f; x \leq f x; P x \rrbracket \implies P (f x)$*
shows $P (\text{ccpo-class.fixp } f)$
 $\langle \text{proof} \rangle$

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 $P :: ('b \Rightarrow 'a) \Rightarrow \text{bool}$

assumes *mono*: $\bigwedge x. \text{mono-body } (\lambda f. U (F (C f))) x$
and *eq*: $f \equiv C (\text{fixp-fun } (\lambda f. U (F (C f))))$
and *inverse*: $\bigwedge f. U (C f) = f$
and *adm*: *ccpo.admissible lub-fun le-fun* *P*
and *bot*: $P (\lambda-. \text{lub } \{\})$
and *step*: $\bigwedge f'. \llbracket \text{le-fun } (U f') (U f); \text{le-fun } (U f') (U (F f')); P (U f') \rrbracket \implies$
 $P (U (F f'))$
shows $P (U f)$
<proof>

end

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

lemma (**in** *ccpo*) *fixp-greatest*:
assumes *f*: *monotone* $(\leq) (\leq) f$
and *ge*: $\bigwedge y. f y \leq y \implies x \leq y$
shows $x \leq \text{ccpo.fixp Sup } (\leq) f$
<proof>

lemma *fixp-rolling*:
assumes *class.ccpo lub1 leq1* (*mk-less leq1*)
and *class.ccpo lub2 leq2* (*mk-less leq2*)
and *f*: *monotone leq1 leq2* *f*
and *g*: *monotone leq2 leq1* *g*
shows $\text{ccpo.fixp lub1 leq1 } (\lambda x. g (f x)) = g (\text{ccpo.fixp lub2 leq2 } (\lambda x. f (g x)))$
<proof>

lemma *fixp-lfp-parametric-eq*:
includes *lifting-syntax*
assumes *f*: $\bigwedge x. \text{lfp.mono-body } (\lambda f. F f x)$
and *g*: $\bigwedge x. \text{lfp.mono-body } (\lambda f. G f x)$
and *param*: $((A \text{====>} (=)) \text{====>} A \text{====>} (=)) F G$
shows $(A \text{====>} (=)) (\text{lfp.fixp-fun } F) (\text{lfp.fixp-fun } G)$
<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) op-*
tion-ord (*map-option* *f*)

<proof>

lemma *mono2mono-set-option* [*THEN lfp.mono2mono*]:

shows *monotone-set-option: monotone option-ord* (\subseteq) *set-option*

<proof>

lemma *mcont2mcont-set-option* [*THEN lfp.mcont2mcont, cont-intro, simp*]:

shows *mcont-set-option: mcont* (*flat-lub None*) *option-ord Union* (\subseteq) *set-option*

<proof>

lemma *eadd-gfp-partial-function-mono* [*partial-function-mono*]:

\llbracket *monotone* (*fun-ord* (\geq)) (\geq) *f*; *monotone* (*fun-ord* (\geq)) (\geq) *g* \rrbracket

\implies *monotone* (*fun-ord* (\geq)) (\geq) ($\lambda x. f x + g x :: \text{enat}$)

<proof>

lemma *map-option-mono* [*partial-function-mono*]:

mono-option B \implies *mono-option* ($\lambda f. \text{map-option } g (B f)$)

<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: $\bigwedge x s A'. \llbracket x \in A'; I A' s; A' \subseteq A \rrbracket \implies I (A' - \{x\}) (f x s)$

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: $\bigwedge x s A'. \llbracket I A' s; x \notin A'; x \in A; A' \subseteq A \rrbracket \implies I (\text{insert } x A') (f x s)$

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-tII*: $xs ! n = z \implies (x \# xs) ! \text{Suc } n = z$

<proof>

lemma *list-all2-append'*:

$length\ us = length\ vs \implies list\text{-}all2\ P\ (xs\ @\ us)\ (ys\ @\ vs) \longleftrightarrow list\text{-}all2\ P\ xs\ ys \wedge list\text{-}all2\ P\ us\ vs$
<proof>

definition *disjointp* :: ('a \Rightarrow bool) list \Rightarrow bool

where *disjointp* xs = *disjoint-family-on* ($\lambda n. \{x. (xs\ !\ n)\ x\}$) {0..*length* xs}

lemma *disjointpD*:

$\llbracket disjointp\ xs; (xs\ !\ n)\ x; (xs\ !\ m)\ x; n < length\ xs; m < length\ xs \rrbracket \implies n = m$
<proof>

lemma *disjointpD'*:

$\llbracket disjointp\ xs; P\ x; Q\ x; xs\ !\ n = P; xs\ !\ m = Q; n < length\ xs; m < length\ xs \rrbracket \implies n = m$
<proof>

lemma *wf-strict-prefix*: *wfP* *strict-prefix*

<proof>

lemma *strict-prefix-setD*:

strict-prefix xs ys $\implies set\ xs \subseteq set\ ys$
<proof>

1.15.1 List of a given length

inductive-set *nlists* :: 'a set \Rightarrow nat \Rightarrow 'a list set **for** A n

where *nlists*: $\llbracket set\ xs \subseteq A; length\ xs = n \rrbracket \implies xs \in nlists\ A\ n$

hide-fact (**open**) *nlists*

lemma *nlists-alt-def*: $nlists\ A\ n = \{xs. set\ xs \subseteq A \wedge length\ xs = n\}$

<proof>

lemma *nlists-empty*: $nlists\ \{\} n = (if\ n = 0\ then\ \{\}\ else\ \{\})$

<proof>

lemma *nlists-empty-gt0* [*simp*]: $n > 0 \implies nlists\ \{\} n = \{\}$

<proof>

lemma *nlists-0* [*simp*]: $nlists\ A\ 0 = \{\}\}$

<proof>

lemma *Cons-in-nlists-Suc* [*simp*]: $x \# xs \in nlists\ A\ (Suc\ n) \longleftrightarrow x \in A \wedge xs \in nlists\ A\ n$

<proof>

lemma *Nil-in-nlists* [*simp*]: $\llbracket \rrbracket \in nlists\ A\ n \longleftrightarrow n = 0$

<proof>

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

lemma *in-nlists-Suc-iff*: $xs \in nlists\ A\ (Suc\ n) \longleftrightarrow (\exists x\ xs'. xs = x \# xs' \wedge x \in A \wedge xs' \in nlists\ A\ n)$
 <proof>

lemma *nlists-Suc*: $nlists\ A\ (Suc\ n) = (\bigcup x \in A. (\#)\ x\ ' nlists\ A\ n)$
 <proof>

lemma *replicate-in-nlists* [*simp, intro*]: $x \in A \implies replicate\ n\ x \in nlists\ A\ n$
 <proof>

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

lemma *finite-nlists* [*simp*]: $finite\ A \implies finite\ (nlists\ A\ n)$
 <proof>

lemma *finite-nlistsD*:
 assumes $finite\ (nlists\ A\ n)$
 shows $finite\ A \vee n = 0$
 <proof>

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

lemma *card-nlists*: $card\ (nlists\ A\ n) = card\ A \wedge n$
 <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 ≤ length ys ⇒ *sprefix xs (ys @- zs) ↔* *prefix xs ys*
<proof>

lemma *prefixeq-stake2* [*simp*]: *prefix xs (stake n ys) ↔ length xs ≤ n ∧* *sprefix xs ys*
<proof>

lemma *tlength-eq-infinity-iff*: *tlength xs = ∞ ↔ ¬ tfinite xs*
including *tlift.list.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*:
 $\bigwedge_{\text{monad.}} (x = \text{None} \implies m = \text{fail}) \implies \text{bind-option fail } x (\lambda-. m) = m$
<proof>

end

lemma *bind-option-option* [*simp*]: *monad.bind-option None = Option.bind*
<proof>

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

lemma *hom-bind-option*: $h \text{ (monad.bind-option fail1 } x f) = \text{monad.bind-option fail2 } x (h \circ f)$
<proof>

end

lemma *bind-option-set* [*simp*]: $\text{monad.bind-option fail-set} = (\lambda x f. \bigcup (f \text{ ' set-option } x))$
<proof>

lemma *run-bind-option-stateT* [*simp*]:
 $\bigwedge \text{more. run-state (monad.bind-option (fail-state fail) } x f) s = \text{monad.bind-option fail } x (\lambda y. \text{run-state (f } y) s)$
<proof>

lemma *run-bind-option-envT* [*simp*]:
 $\bigwedge \text{more. run-env (monad.bind-option (fail-env fail) } x f) s = \text{monad.bind-option fail } x (\lambda y. \text{run-env (f } y) s)$
<proof>

1.18 Measures

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

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

lemma *pred-countable-Ex1* [*measurable*]:
 $(\bigwedge i :: - :: \text{countable. Measurable.pred } M (\lambda x. P \ i \ x)) \implies \text{Measurable.pred } M (\lambda x. \exists !i. P \ i \ x)$
<proof>

lemma *measurable-snd-count-space* [*measurable*]:
 $A \subseteq B \implies \text{snd} \in \text{measurable } (M1 \otimes_M \text{count-space } A) (\text{count-space } B)$
<proof>

lemma *integrable-scale-measure* [*simp*]:
 $\llbracket \text{integrable } M \ f; r < \top \rrbracket \implies \text{integrable (scale-measure } r \ M) \ f$
for $f :: 'a \Rightarrow 'b :: \{\text{banach, second-countable-topology}\}$
<proof>

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

1.19 Sequence space

lemma (in *sequence-space*) *nn-integral-split*:

assumes $f[\text{measurable}]$: $f \in \text{borel-measurable } S$

shows $(\int^{+\omega}. f \ \omega \ \partial S) = (\int^{+\omega}. (\int^{+\omega'}. f \ (\text{comb-seq } i \ \omega \ \omega') \ \partial S) \ \partial S)$

<proof>

lemma (in *sequence-space*) *prob-Collect-split*:

assumes $f[\text{measurable}]$: $\{x \in \text{space } S. P \ x\} \in \text{sets } S$

shows $\mathcal{P}(x \text{ in } S. P \ x) = (\int^{+x}. \mathcal{P}(x' \text{ in } S. P \ (\text{comb-seq } i \ x \ x')) \ \partial S)$

<proof>

1.20 Probability mass functions

lemma *measure-map-pmf-conv-distr*:

$\text{measure-pmf} \ (\text{map-pmf } f \ p) = \text{distr} \ (\text{measure-pmf } p) \ (\text{count-space } \text{UNIV}) \ f$

<proof>

abbreviation *coin-pmf* :: *bool pmf where coin-pmf* \equiv *pmf-of-set UNIV*

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

notepad begin

<proof>

end

lemma *pred-rel-pmf*:

$\llbracket \text{pred-pmf } P \ p; \text{rel-pmf } R \ p \ q \rrbracket \implies \text{pred-pmf} \ (\text{Imagep } R \ P) \ q$

<proof>

lemma *pmf-rel-mono'*: $\llbracket \text{rel-pmf } P \ x \ y; P \leq Q \rrbracket \implies \text{rel-pmf } Q \ x \ y$

<proof>

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

<proof>

lemma *rel-pmf-bind-reflI*:

$(\bigwedge x. x \in \text{set-pmf } p \implies \text{rel-pmf } R \ (f \ x) \ (g \ x))$

$\implies \text{rel-pmf } R \ (\text{bind-pmf } p \ f) \ (\text{bind-pmf } p \ g)$

<proof>

lemma *pmf-pred-mono-strong*:

$\llbracket \text{pred-pmf } P \ p; \bigwedge a. \llbracket a \in \text{set-pmf } p; P \ a \rrbracket \implies P' \ a \rrbracket \implies \text{pred-pmf } P' \ p$

<proof>

lemma *rel-pmf-restrict-relpI [intro?]*:

$\llbracket \text{rel-pmf } R \ x \ y; \text{pred-pmf } P \ x; \text{pred-pmf } Q \ y \rrbracket \implies \text{rel-pmf} \ (R \upharpoonright P \otimes Q) \ x \ y$

<proof>

lemma *rel-pmf-restrict-relpE [elim?]*:

assumes $\text{rel-pmf} \ (R \upharpoonright P \otimes Q) \ x \ y$

obtains $rel\text{-pmf } R \ x \ y \ pred\text{-pmf } P \ x \ pred\text{-pmf } Q \ y$
 $\langle proof \rangle$

lemma $rel\text{-pmf-restrict-relp-iff}$:
 $rel\text{-pmf } (R \upharpoonright P \otimes Q) \ x \ y \longleftrightarrow rel\text{-pmf } R \ x \ y \wedge pred\text{-pmf } P \ x \wedge pred\text{-pmf } Q \ y$
 $\langle proof \rangle$

lemma $rel\text{-pmf-OO-trans}$ [*trans*]:
 $\llbracket rel\text{-pmf } R \ p \ q; rel\text{-pmf } S \ q \ r \rrbracket \Longrightarrow rel\text{-pmf } (R \ OO \ S) \ p \ r$
 $\langle proof \rangle$

lemma $pmf\text{-pred-map}$ [*simp*]: $pred\text{-pmf } P \ (map\text{-pmf } f \ p) = pred\text{-pmf } (P \circ f) \ p$
 $\langle proof \rangle$

lemma $pred\text{-pmf-bind}$ [*simp*]: $pred\text{-pmf } P \ (bind\text{-pmf } p \ f) = pred\text{-pmf } (pred\text{-pmf } P \circ f) \ p$
 $\langle proof \rangle$

lemma $pred\text{-pmf-return}$ [*simp*]: $pred\text{-pmf } P \ (return\text{-pmf } x) = P \ x$
 $\langle proof \rangle$

lemma $pred\text{-pmf-of-set}$ [*simp*]: $\llbracket finite \ A; A \neq \{\} \rrbracket \Longrightarrow pred\text{-pmf } P \ (pmf\text{-of-set } A) = Ball \ A \ P$
 $\langle proof \rangle$

lemma $pred\text{-pmf-of-multiset}$ [*simp*]: $M \neq \{\#\} \Longrightarrow pred\text{-pmf } P \ (pmf\text{-of-multiset } M) = Ball \ (set\text{-mset } M) \ P$
 $\langle proof \rangle$

lemma $pred\text{-pmf-cond}$ [*simp*]:
 $set\text{-pmf } p \cap A \neq \{\} \Longrightarrow pred\text{-pmf } P \ (cond\text{-pmf } p \ A) = pred\text{-pmf } (\lambda x. x \in A \longrightarrow P \ x) \ p$
 $\langle proof \rangle$

lemma $pred\text{-pmf-pair}$ [*simp*]:
 $pred\text{-pmf } P \ (pair\text{-pmf } p \ q) = pred\text{-pmf } (\lambda x. pred\text{-pmf } (P \circ Pair \ x) \ q) \ p$
 $\langle proof \rangle$

lemma $pred\text{-pmf-join}$ [*simp*]: $pred\text{-pmf } P \ (join\text{-pmf } p) = pred\text{-pmf } (pred\text{-pmf } P) \ p$
 $\langle proof \rangle$

lemma $pred\text{-pmf-bernoulli}$ [*simp*]: $\llbracket 0 < p; p < 1 \rrbracket \Longrightarrow pred\text{-pmf } P \ (bernoulli\text{-pmf } p) = All \ P$
 $\langle proof \rangle$

lemma $pred\text{-pmf-geometric}$ [*simp*]: $\llbracket 0 < p; p < 1 \rrbracket \Longrightarrow pred\text{-pmf } P \ (geometric\text{-pmf } p) = All \ P$
 $\langle proof \rangle$

lemma *pred-pmf-poisson* [simp]: $0 < \text{rate} \implies \text{pred-pmf } P \text{ (poisson-pmf rate)} = \text{All } P$
 <proof>

lemma *pmf-rel-map-restrict-relp*:
 shows *pmf-rel-map-restrict-relp1*: $\text{rel-pmf } (R \upharpoonright P \otimes Q) (\text{map-pmf } f \ p) = \text{rel-pmf } (R \circ f \upharpoonright P \circ f \otimes Q) \ p$
 and *pmf-rel-map-restrict-relp2*: $\text{rel-pmf } (R \upharpoonright P \otimes Q) \ p (\text{map-pmf } g \ q) = \text{rel-pmf } ((\lambda x. R \ x \circ g) \upharpoonright P \otimes Q \circ g) \ p \ q$
 <proof>

lemma *pred-pmf-conj* [simp]: $\text{pred-pmf } (\lambda x. P \ x \wedge Q \ x) = (\lambda x. \text{pred-pmf } P \ x \wedge \text{pred-pmf } Q \ x)$
 <proof>

lemma *pred-pmf-top* [simp]:
 $\text{pred-pmf } (\lambda _. \text{True}) = (\lambda _. \text{True})$
 <proof>

lemma *rel-pmf-of-setI*:
 assumes *A*: $A \neq \{\}$ *finite A*
 and *B*: $B \neq \{\}$ *finite B*
 and *card*: $\bigwedge X. X \subseteq A \implies \text{card } B * \text{card } X \leq \text{card } A * \text{card } \{y \in B. \exists x \in X. R \ x \ y\}$
 shows $\text{rel-pmf } R \ (\text{pmf-of-set } A) \ (\text{pmf-of-set } B)$
 <proof>

consts *rel-witness-pmf* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \ \text{pmf} \times 'b \ \text{pmf} \Rightarrow ('a \times 'b) \ \text{pmf}$
specification (*rel-witness-pmf*)
set-rel-witness-pmf': $\text{rel-pmf } A \ (\text{fst } xy) \ (\text{snd } xy) \implies \text{set-pmf } (\text{rel-witness-pmf } A \ xy) \subseteq \{(a, b). A \ a \ b\}$
map1-rel-witness-pmf': $\text{rel-pmf } A \ (\text{fst } xy) \ (\text{snd } xy) \implies \text{map-pmf } \text{fst} \ (\text{rel-witness-pmf } A \ xy) = \text{fst } xy$
map2-rel-witness-pmf': $\text{rel-pmf } A \ (\text{fst } xy) \ (\text{snd } xy) \implies \text{map-pmf } \text{snd} \ (\text{rel-witness-pmf } A \ xy) = \text{snd } xy$
 <proof>

lemmas $\text{set-rel-witness-pmf} = \text{set-rel-witness-pmf}'[\text{of } - \ (x, y) \ \text{for } x \ y, \ \text{simplified}]$
lemmas $\text{map1-rel-witness-pmf} = \text{map1-rel-witness-pmf}'[\text{of } - \ (x, y) \ \text{for } x \ y, \ \text{simplified}]$
lemmas $\text{map2-rel-witness-pmf} = \text{map2-rel-witness-pmf}'[\text{of } - \ (x, y) \ \text{for } x \ y, \ \text{simplified}]$
lemmas $\text{rel-witness-pmf} = \text{set-rel-witness-pmf} \ \text{map1-rel-witness-pmf} \ \text{map2-rel-witness-pmf}$

lemma *rel-witness-pmf1*:
 assumes *rel-pmf A p q*
 shows $\text{rel-pmf } (\lambda a \ (a', b). a = a' \wedge A \ a' \ b) \ p \ (\text{rel-witness-pmf } A \ (p, q))$
 <proof>

lemma *rel-witness-pmf2*:

assumes *rel-pmf* A p q

shows *rel-pmf* $(\lambda(a, b'). b. b = b' \wedge A a b')$ (*rel-witness-pmf* A (p, q)) q

<proof>

lemma *cond-pmf-of-set*:

assumes *fin*: *finite* A **and** *nonempty*: $A \cap B \neq \{\}$

shows *cond-pmf* (*pmf-of-set* A) $B = \text{pmf-of-set } (A \cap B)$ (**is** *?lhs = ?rhs*)

<proof>

lemma *pair-pmf-of-set*:

assumes A : *finite* A $A \neq \{\}$

and B : *finite* B $B \neq \{\}$

shows *pair-pmf* (*pmf-of-set* A) (*pmf-of-set* B) = *pmf-of-set* $(A \times B)$

<proof>

lemma *emeasure-cond-pmf*:

fixes p A

defines $q \equiv \text{cond-pmf } p$ A

assumes *set-pmf* $p \cap A \neq \{\}$

shows *emeasure* (*measure-pmf* q) $B = \text{emeasure } (\text{measure-pmf } p)$ $(A \cap B)$ /
emeasure (*measure-pmf* p) A

<proof>

lemma *measure-cond-pmf*:

measure (*measure-pmf* (*cond-pmf* p A)) $B = \text{measure } (\text{measure-pmf } p)$ $(A \cap B)$
/ *measure* (*measure-pmf* p) A

if *set-pmf* $p \cap A \neq \{\}$

<proof>

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

<proof>

1.21 Subprobability mass functions

lemma *ord-spmf-return-spmf1*: *ord-spmf* R (*return-spmf* x) $p \iff \text{lossless-spmf } p$
 $\wedge (\forall y \in \text{set-spmf } p. R x y)$

<proof>

lemma *ord-spmf-conv*:

ord-spmf $R = \text{rel-spmf } R$ *OO* *ord-spmf* $(=)$

<proof>

lemma *ord-spmf-expand*:

NO-MATCH $(=)$ $R \implies \text{ord-spmf } R = \text{rel-spmf } R$ *OO* *ord-spmf* $(=)$

<proof>

lemma *ord-spmf-eqD-measure*: *ord-spmf* $(=)$ p $q \implies \text{measure } (\text{measure-spmf } p)$

$A \leq \text{measure } (\text{measure-spmf } q) A$
 $\langle \text{proof} \rangle$

lemma *ord-spmf-measureD*:

assumes *ord-spmf* R p q
shows $\text{measure } (\text{measure-spmf } p) A \leq \text{measure } (\text{measure-spmf } q) \{y. \exists x \in A. R$
 $x\ y\}$
(is ?lhs \leq ?rhs)
 $\langle \text{proof} \rangle$

lemma *ord-spmf-bind-pmfI1*:

$(\bigwedge x. x \in \text{set-pmf } p \implies \text{ord-spmf } R (f\ x) q) \implies \text{ord-spmf } R (\text{bind-pmf } p\ f) q$
 $\langle \text{proof} \rangle$

lemma *ord-spmf-bind-spmfI1*:

$(\bigwedge x. x \in \text{set-spmf } p \implies \text{ord-spmf } R (f\ x) q) \implies \text{ord-spmf } R (\text{bind-spmf } p\ f) q$
 $\langle \text{proof} \rangle$

lemma *spmf-of-set-empty*: $\text{spmf-of-set } \{\} = \text{return-pmf } \text{None}$
 $\langle \text{proof} \rangle$

lemma *rel-spmf-of-setI*:

assumes *card*: $\bigwedge X. X \subseteq A \implies \text{card } B * \text{card } X \leq \text{card } A * \text{card } \{y \in B. \exists x \in X. R$
 $x\ y\}$
and eq: $(\text{finite } A \wedge A \neq \{\}) \longleftrightarrow (\text{finite } B \wedge B \neq \{\})$
shows $\text{rel-spmf } R (\text{spmf-of-set } A) (\text{spmf-of-set } B)$
 $\langle \text{proof} \rangle$

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

lemma *nn-integral-measure-spmf-conv-measure-pmf*:

assumes [*measurable*]: $f \in \text{borel-measurable } (\text{count-space } \text{UNIV})$
shows $\text{nn-integral } (\text{measure-spmf } p) f = \text{nn-integral } (\text{restrict-space } (\text{measure-pmf } p) (\text{range } \text{Some})) (f \circ \text{the})$
 $\langle \text{proof} \rangle$

lemma *nn-integral-spmf-neq-infinity*: $(\int^+ x. \text{spmf } p\ x \partial \text{count-space } \text{UNIV}) \neq \infty$
 $\langle \text{proof} \rangle$

lemma *return-pmf-bind-option*:

$\text{return-pmf } (\text{Option.bind } x\ f) = \text{bind-spmf } (\text{return-pmf } x) (\text{return-pmf } \circ f)$
 $\langle \text{proof} \rangle$

lemma *rel-spmf-pos-distr*: $\text{rel-spmf } A\ \text{OO } \text{rel-spmf } B \leq \text{rel-spmf } (A\ \text{OO } B)$
 $\langle \text{proof} \rangle$

lemma *rel-spmf-OO-trans* [*trans*]:

$\llbracket \text{rel-spmf } R\ p\ q; \text{rel-spmf } S\ q\ r \rrbracket \implies \text{rel-spmf } (R\ \text{OO } S) p\ r$
 $\langle \text{proof} \rangle$

lemma *map-spmf-eq-map-spmf-iff*: $\text{map-spmf } f \ p = \text{map-spmf } g \ q \longleftrightarrow \text{rel-spmf } (\lambda x \ y. f \ x = g \ y) \ p \ q$
 ⟨proof⟩

lemma *map-spmf-eq-map-spmfI*: $\text{rel-spmf } (\lambda x \ y. f \ x = g \ y) \ p \ q \implies \text{map-spmf } f \ p = \text{map-spmf } g \ q$
 ⟨proof⟩

lemma *spmf-rel-mono-strong*:
 $\llbracket \text{rel-spmf } A \ f \ g; \bigwedge x \ y. \llbracket x \in \text{set-spmf } f; y \in \text{set-spmf } g; A \ x \ y \rrbracket \implies B \ x \ y \rrbracket \implies \text{rel-spmf } B \ f \ g$
 ⟨proof⟩

lemma *set-spmf-eq-empty*: $\text{set-spmf } p = \{\} \longleftrightarrow p = \text{return-pmf } \text{None}$
 ⟨proof⟩

lemma *measure-pair-spmf-times*:
 $\text{measure } (\text{measure-spmf } (\text{pair-spmf } p \ q)) \ (A \times B) = \text{measure } (\text{measure-spmf } p) \ A * \text{measure } (\text{measure-spmf } q) \ B$
 ⟨proof⟩

lemma *lossless-spmfD-set-spmf-nonempty*: $\text{lossless-spmf } p \implies \text{set-spmf } p \neq \{\}$
 ⟨proof⟩

lemma *set-spmf-return-pmf*: $\text{set-spmf } (\text{return-pmf } x) = \text{set-option } x$
 ⟨proof⟩

lemma *bind-spmf-pmf-assoc*: $\text{bind-spmf } (\text{bind-pmf } p \ f) \ g = \text{bind-pmf } p \ (\lambda x. \text{bind-spmf } (f \ x) \ g)$
 ⟨proof⟩

lemma *bind-spmf-of-set*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{bind-spmf } (\text{spmof-of-set } A) \ f = \text{bind-pmf } (\text{pmf-of-set } A) \ f$
 ⟨proof⟩

lemma *bind-spmf-map-pmf*:
 $\text{bind-spmf } (\text{map-pmf } f \ p) \ g = \text{bind-pmf } p \ (\lambda x. \text{bind-spmf } (\text{return-pmf } (f \ x)) \ g)$
 ⟨proof⟩

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

lemma *set-spmf-map-pmf*: $\text{set-spmf } (\text{map-pmf } f \ p) = \bigcup_{x \in \text{set-pmf } p} \text{set-option } (f \ x)$
 ⟨proof⟩

lemma *ord-spmf-return-spmf [simp]*: $\text{ord-spmf } (=) \ (\text{return-spmf } x) \ p \longleftrightarrow p =$

return-spmf x
 \langle proof \rangle

declare

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

lemma *bind-spmf-pmf-commute*:

$bind\text{-}spmf\ p\ (\lambda x. bind\text{-}pmf\ q\ (f\ x)) = bind\text{-}pmf\ q\ (\lambda y. bind\text{-}spmf\ p\ (\lambda x. f\ x\ y))$
 \langle proof \rangle

lemma *return-pmf-map-option-conv-bind*:

$return\text{-}pmf\ (map\text{-}option\ f\ x) = bind\text{-}spmf\ (return\text{-}pmf\ x)\ (return\text{-}spmf\ \circ\ f)$
 \langle proof \rangle

lemma *lossless-return-pmf-iff* [*simp*]: $lossless\text{-}spmf\ (return\text{-}pmf\ x) \longleftrightarrow x \neq None$
 \langle proof \rangle

lemma *lossless-map-pmf*: $lossless\text{-}spmf\ (map\text{-}pmf\ f\ p) \longleftrightarrow (\forall x \in set\text{-}pmf\ p. f\ x \neq None)$
 \langle proof \rangle

lemma *bind-pmf-spmf-assoc*:

$g\ None = return\text{-}pmf\ None$
 $\implies bind\text{-}pmf\ (bind\text{-}spmf\ p\ f)\ g = bind\text{-}spmf\ p\ (\lambda x. bind\text{-}pmf\ (f\ x)\ g)$
 \langle proof \rangle

abbreviation $pred\text{-}spmf :: ('a \Rightarrow bool) \Rightarrow 'a\ spmf \Rightarrow bool$

where $pred\text{-}spmf\ P \equiv pred\text{-}pmf\ (pred\text{-}option\ P)$

lemma *pred-spmf-def*: $pred\text{-}spmf\ P\ p \longleftrightarrow (\forall x \in set\text{-}spmf\ p. P\ x)$
 \langle proof \rangle

lemma *spmf-pred-mono-strong*:

$\llbracket pred\text{-}spmf\ P\ p; \bigwedge a. \llbracket a \in set\text{-}spmf\ p; P\ a \rrbracket \implies P'\ a \rrbracket \implies pred\text{-}spmf\ P'\ p$
 \langle proof \rangle

lemma *spmf-Domainp-rel*: $Domainp\ (rel\text{-}spmf\ R) = pred\text{-}spmf\ (Domainp\ R)$
 \langle proof \rangle

lemma *rel-spmf-restrict-relpI* [*intro?*]:

$\llbracket rel\text{-}spmf\ R\ p\ q; pred\text{-}spmf\ P\ p; pred\text{-}spmf\ Q\ q \rrbracket \implies rel\text{-}spmf\ (R \upharpoonright P \otimes Q)\ p\ q$
 \langle proof \rangle

lemma *rel-spmf-restrict-relpE* [*elim?*]:

assumes $rel\text{-}spmf\ (R \upharpoonright P \otimes Q)\ x\ y$
obtains $rel\text{-}spmf\ R\ x\ y\ pred\text{-}spmf\ P\ x\ pred\text{-}spmf\ Q\ y$
 \langle proof \rangle

lemma *rel-spmf-restrict-relp-iff*:

$rel\text{-}spmf (R \upharpoonright P \otimes Q) x y \iff rel\text{-}spmf R x y \wedge pred\text{-}spmf P x \wedge pred\text{-}spmf Q y$
<proof>

lemma *spmf-pred-map*: $pred\text{-}spmf P (map\text{-}spmf f p) = pred\text{-}spmf (P \circ f) p$
<proof>

lemma *pred-spmf-bind [simp]*: $pred\text{-}spmf P (bind\text{-}spmf p f) = pred\text{-}spmf (pred\text{-}spmf P \circ f) p$
<proof>

lemma *pred-spmf-return*: $pred\text{-}spmf P (return\text{-}spmf x) = P x$
<proof>

lemma *pred-spmf-return-pmf-None*: $pred\text{-}spmf P (return\text{-}pmf None)$
<proof>

lemma *pred-spmf-spmf-of-pmf [simp]*: $pred\text{-}spmf P (spmf\text{-of-pmf p) = pred\text{-}pmf P p$
<proof>

lemma *pred-spmf-of-set [simp]*: $pred\text{-}spmf P (spmf\text{-of-set A) = (finite A \longrightarrow Ball A P)$
<proof>

lemma *pred-spmf-assert-spmf [simp]*: $pred\text{-}spmf P (assert\text{-}spmf b) = (b \longrightarrow P ())$
<proof>

lemma *pred-spmf-pair [simp]*:
 $pred\text{-}spmf P (pair\text{-}spmf p q) = pred\text{-}spmf (\lambda x. pred\text{-}spmf (P \circ Pair x) q) p$
<proof>

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

lemma *try-spmf-bind-out1*:
 $(\bigwedge x. lossless\text{-}spmf (f x)) \implies bind\text{-}spmf (TRY p ELSE q) f = TRY (bind\text{-}spmf p f) ELSE (bind\text{-}spmf q f)$
<proof>

lemma *pred-spmf-try [simp]*:
 $pred\text{-}spmf P (try\text{-}spmf p q) = (pred\text{-}spmf P p \wedge (\neg lossless\text{-}spmf p \longrightarrow pred\text{-}spmf P q))$
<proof>

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

<proof>

lemma *spmf-rel-map-restrict-relp*:

shows *spmf-rel-map-restrict-relp1*: $rel\text{-}spmf\ (R \upharpoonright P \otimes Q)\ (map\text{-}spmf\ f\ p) = rel\text{-}spmf\ (R \circ f \upharpoonright P \circ f \otimes Q)\ p$

and *spmf-rel-map-restrict-relp2*: $rel\text{-}spmf\ (R \upharpoonright P \otimes Q)\ p\ (map\text{-}spmf\ g\ q) = rel\text{-}spmf\ ((\lambda x. R\ x \circ g) \upharpoonright P \otimes Q \circ g)\ p\ q$

<proof>

lemma *pred-spmf-conj*: $pred\text{-}spmf\ (\lambda x. P\ x \wedge Q\ x) = (\lambda x. pred\text{-}spmf\ P\ x \wedge pred\text{-}spmf\ Q\ x)$

<proof>

lemma *spmf-of-pmf-parametric* [*transfer-rule*]:

includes *lifting-syntax* **shows**

$(rel\text{-}pmf\ A ==> rel\text{-}spmf\ A)\ spmf\text{-}of\text{-}pmf\ spmf\text{-}of\text{-}pmf$

<proof>

lemma *mono2mono-return-pmf*[*THEN* *spmf.mono2mono*, *simp*, *cont-intro*]:

shows *monotone-return-pmf*: $monotone\ option\text{-}ord\ (ord\text{-}spmf\ (=))\ return\text{-}pmf$

<proof>

lemma *mcont2mcont-return-pmf*[*THEN* *spmf.mcont2mcont*, *simp*, *cont-intro*]:

shows *mcont-return-pmf*: $mcont\ (flat\text{-}lub\ None)\ option\text{-}ord\ lub\text{-}spmf\ (ord\text{-}spmf\ (=))\ return\text{-}pmf$

<proof>

lemma *pred-spmf-top*:

$pred\text{-}spmf\ (\lambda\cdot. True) = (\lambda\cdot. True)$

<proof>

lemma *rel-spmf-restrict-relpI'* [*intro?*]:

$\llbracket rel\text{-}spmf\ (\lambda x\ y. P\ x \longrightarrow Q\ y \longrightarrow R\ x\ y)\ p\ q;\ pred\text{-}spmf\ P\ p;\ pred\text{-}spmf\ Q\ q \rrbracket \Longrightarrow rel\text{-}spmf\ (R \upharpoonright P \otimes Q)\ p\ q$

<proof>

lemma *set-spmf-map-pmf-MATCH* [*simp*]:

assumes *NO-MATCH* (*map-option* *g*) *f*

shows $set\text{-}spmf\ (map\text{-}pmf\ f\ p) = (\bigcup x \in set\text{-}pmf\ p. set\text{-}option\ (f\ x))$

<proof>

lemma *rel-spmf-bindI'*:

$\llbracket rel\text{-}spmf\ A\ p\ q;\ \bigwedge x\ y. \llbracket A\ x\ y;\ x \in set\text{-}spmf\ p;\ y \in set\text{-}spmf\ q \rrbracket \Longrightarrow rel\text{-}spmf\ B\ (f\ x)\ (g\ y) \rrbracket$

$\Longrightarrow rel\text{-}spmf\ B\ (p \ggg f)\ (q \ggg g)$

<proof>

definition *rel-witness-spmf* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ spmf \times 'b\ spmf \Rightarrow ('a \times 'b)\ spmf$ **where**

$rel\text{-}witness\text{-}spmf\ A = map\text{-}pmf\ rel\text{-}witness\text{-}option \circ rel\text{-}witness\text{-}pmf\ (rel\text{-}option\ A)$

lemma assumes $rel\text{-}spmf\ A\ p\ q$

shows $rel\text{-}witness\text{-}spmf1: rel\text{-}spmf\ (\lambda a\ (a', b). a = a' \wedge A\ a'\ b)\ p\ (rel\text{-}witness\text{-}spmf\ A\ (p, q))$

and $rel\text{-}witness\text{-}spmf2: rel\text{-}spmf\ (\lambda(a, b'). b = b' \wedge A\ a\ b')\ (rel\text{-}witness\text{-}spmf\ A\ (p, q))\ q$

$\langle proof \rangle$

lemma $weight\text{-}assert\text{-}spmf\ [simp]: weight\text{-}spmf\ (assert\text{-}spmf\ b) = indicator\ \{True\}\ b$

$\langle proof \rangle$

definition $enforce\text{-}spmf :: ('a \Rightarrow bool) \Rightarrow 'a\ spmf \Rightarrow 'a\ spmf$ **where**

$enforce\text{-}spmf\ P = map\text{-}pmf\ (enforce\text{-}option\ P)$

lemma $enforce\text{-}spmf\text{-}parametric\ [transfer\text{-}rule]: includes\ lifting\text{-}syntax\ shows$

$((A\ ==> (=))\ ==> rel\text{-}spmf\ A\ ==> rel\text{-}spmf\ A)\ enforce\text{-}spmf\ enforce\text{-}spmf$

$\langle proof \rangle$

lemma $enforce\text{-}return\text{-}spmf\ [simp]:$

$enforce\text{-}spmf\ P\ (return\text{-}spmf\ x) = (if\ P\ x\ then\ return\text{-}spmf\ x\ else\ return\text{-}pmf\ None)$

$\langle proof \rangle$

lemma $enforce\text{-}return\text{-}pmf\ None\ [simp]:$

$enforce\text{-}spmf\ P\ (return\text{-}pmf\ None) = return\text{-}pmf\ None$

$\langle proof \rangle$

lemma $enforce\text{-}map\text{-}spmf:$

$enforce\text{-}spmf\ P\ (map\text{-}spmf\ f\ p) = map\text{-}spmf\ f\ (enforce\text{-}spmf\ (P \circ f)\ p)$

$\langle proof \rangle$

lemma $enforce\text{-}bind\text{-}spmf\ [simp]:$

$enforce\text{-}spmf\ P\ (bind\text{-}spmf\ p\ f) = bind\text{-}spmf\ p\ (enforce\text{-}spmf\ P \circ f)$

$\langle proof \rangle$

lemma $set\text{-}enforce\text{-}spmf\ [simp]: set\text{-}spmf\ (enforce\text{-}spmf\ P\ p) = \{a \in set\text{-}spmf\ p.$

$P\ a\}$

$\langle proof \rangle$

lemma $enforce\text{-}spmf\text{-}alt\text{-}def:$

$enforce\text{-}spmf\ P\ p = bind\text{-}spmf\ p\ (\lambda a. bind\text{-}spmf\ (assert\text{-}spmf\ (P\ a))\ (\lambda - :: unit. return\text{-}spmf\ a))$

$\langle proof \rangle$

lemma $bind\text{-}enforce\text{-}spmf\ [simp]:$

$bind\text{-}spmf\ (enforce\text{-}spmf\ P\ p)\ f = bind\text{-}spmf\ p\ (\lambda x. if\ P\ x\ then\ f\ x\ else\ return\text{-}pmf\ None)$

<proof>

lemma *weight-enforce-spmf*:

$weight\text{-}spmf\ (enforce\text{-}spmf\ P\ p) = weight\text{-}spmf\ p - measure\ (measure\text{-}spmf\ p)\ \{x.\ \neg\ P\ x\}$ (is ?lhs = ?rhs)
<proof>

lemma *lossless-enforce-spmf* [simp]:

$lossless\text{-}spmf\ (enforce\text{-}spmf\ P\ p) \longleftrightarrow lossless\text{-}spmf\ p \wedge set\text{-}spmf\ p \subseteq \{x.\ P\ x\}$
<proof>

lemma *enforce-spmf-top* [simp]: $enforce\text{-}spmf\ \top = id$

<proof>

lemma *enforce-spmf-K-True* [simp]: $enforce\text{-}spmf\ (\lambda_.\ True)\ p = p$

<proof>

lemma *enforce-spmf-bot* [simp]: $enforce\text{-}spmf\ \perp = (\lambda_.\ return\text{-}pmf\ None)$

<proof>

lemma *enforce-spmf-K-False* [simp]: $enforce\text{-}spmf\ (\lambda_.\ False)\ p = return\text{-}pmf\ None$

<proof>

lemma *enforce-pred-id-spmf*: $enforce\text{-}spmf\ P\ p = p$ if $pred\text{-}spmf\ P\ p$

<proof>

lemma *map-the-spmf-of-pmf* [simp]: $map\text{-}pmf\ the\ (spmf\text{-}of\text{-}pmf\ p) = p$

<proof>

lemma *bind-bind-conv-pair-spmf*:

$bind\text{-}spmf\ p\ (\lambda x.\ bind\text{-}spmf\ q\ (f\ x)) = bind\text{-}spmf\ (pair\text{-}spmf\ p\ q)\ (\lambda(x, y).\ f\ x\ y)$
<proof>

lemma *cond-spmf-spmf-of-set*:

$cond\text{-}spmf\ (spmf\text{-}of\text{-}set\ A)\ B = spmf\text{-}of\text{-}set\ (A \cap B)$ if *finite* A
<proof>

lemma *pair-spmf-of-set*:

$pair\text{-}spmf\ (spmf\text{-}of\text{-}set\ A)\ (spmf\text{-}of\text{-}set\ B) = spmf\text{-}of\text{-}set\ (A \times B)$
<proof>

lemma *emeasure-cond-spmf*:

$emeasure\ (measure\text{-}spmf\ (cond\text{-}spmf\ p\ A))\ B = emeasure\ (measure\text{-}spmf\ p)\ (A \cap B) / emeasure\ (measure\text{-}spmf\ p)\ A$
<proof>

lemma *measure-cond-spmf*:

$measure\ (measure\text{-}spmf\ (cond\text{-}spmf\ p\ A))\ B = measure\ (measure\text{-}spmf\ p)\ (A \cap B) / measure\ (measure\text{-}spmf\ p)\ A$

<proof>

lemma *lossless-cond-spmf* [simp]: *lossless-spmf* (cond-spmf p A) \longleftrightarrow set-spmf p \cap A \neq {}
<proof>

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

lemma *integral-measure-spmf*:
fixes f :: 'a \Rightarrow 'b::{banach, second-countable-topology}
assumes A: finite A
shows ($\bigwedge a. a \in$ set-spmf M \implies f a \neq 0 \implies a \in A) \implies (LINT x | *measure-spmf* M. f x) = ($\sum a \in A. \text{spmf } M \ a \ *_{\mathbb{R}} \ f \ a$)
<proof>

lemma *image-set-spmf-eq*:
f ' set-spmf p = g ' set-spmf q **if** ASSUMPTION (map-spmf f p = map-spmf g q)
<proof>

lemma *map-spmf-const*: map-spmf ($\lambda \cdot. x$) p = scale-spmf (weight-spmf p) (return-spmf x)
<proof>

lemma *cond-return-pmf* [simp]: cond-pmf (return-pmf x) A = return-pmf x **if** x \in A
<proof>

lemma *cond-return-spmf* [simp]: cond-spmf (return-spmf x) A = (if x \in A then return-spmf x else return-pmf None)
<proof>

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

lemma *restrict-spmf-eq-return-pmf-None* [simp]:
restrict-spmf p A = return-pmf None \longleftrightarrow set-spmf p \cap A = {}
<proof>

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

lemma *mk-lossless-idem* [simp]: *mk-lossless* (mk-lossless p) = *mk-lossless* p
<proof>

lemma *mk-lossless-return* [*simp*]: $mk\text{-lossless} (return\text{-pmf } x) = return\text{-pmf } x$
(*proof*)

lemma *mk-lossless-map* [*simp*]: $mk\text{-lossless} (map\text{-spmf } f p) = map\text{-spmf } f (mk\text{-lossless } p)$
(*proof*)

lemma *spmf-mk-lossless* [*simp*]: $spmf (mk\text{-lossless } p) x = spmf p x / weight\text{-spmf } p$
(*proof*)

lemma *set-spmf-mk-lossless* [*simp*]: $set\text{-spmf} (mk\text{-lossless } p) = set\text{-spmf } p$
(*proof*)

lemma *mk-lossless-lossless* [*simp*]: $lossless\text{-spmf } p \implies mk\text{-lossless } p = p$
(*proof*)

lemma *mk-lossless-eq-return-pmf-None* [*simp*]: $mk\text{-lossless } p = return\text{-pmf } None \iff p = return\text{-pmf } None$
(*proof*)

lemma *return-pmf-None-eq-mk-lossless* [*simp*]: $return\text{-pmf } None = mk\text{-lossless } p \iff p = return\text{-pmf } None$
(*proof*)

lemma *mk-lossless-spmf-of-set* [*simp*]: $mk\text{-lossless} (spmf\text{-of-set } A) = spmf\text{-of-set } A$
(*proof*)

lemma *weight-mk-lossless*: $weight\text{-spmf} (mk\text{-lossless } p) = (if p = return\text{-pmf } None \text{ then } 0 \text{ else } 1)$
(*proof*)

lemma *mk-lossless-parametric* [*transfer-rule*]: **includes** *lifting-syntax shows*
(*rel-spmf* $A \implies rel\text{-spmf } A$) *mk-lossless mk-lossless*
(*proof*)

lemma *rel-spmf-mk-losslessI*:
 $rel\text{-spmf } A p q \implies rel\text{-spmf } A (mk\text{-lossless } p) (mk\text{-lossless } q)$
(*proof*)

lemma *rel-spmf-restrict-spmfI*:
 $rel\text{-spmf} (\lambda x y. (x \in A \wedge y \in B \wedge R x y) \vee x \notin A \wedge y \notin B) p q$
 $\implies rel\text{-spmf } R (restrict\text{-spmf } p A) (restrict\text{-spmf } q B)$
(*proof*)

lemma *cond-spmf-alt*: $cond\text{-spmf } p A = mk\text{-lossless} (restrict\text{-spmf } p A)$
(*proof*)

lemma *cond-spmf-bind*:

$cond\text{-}spmf\ (bind\text{-}spmf\ p\ f)\ A = mk\text{-}lossless\ (p \gg= (\lambda x. f\ x \mid A))$
 $\langle proof \rangle$

lemma *cond-spmf-UNIV* [simp]: $cond\text{-}spmf\ p\ UNIV = mk\text{-}lossless\ p$

$\langle proof \rangle$

lemma *cond-pmf-singleton*:

$cond\text{-}pmf\ p\ A = return\text{-}pmf\ x$ **if** $set\text{-}pmf\ p \cap A = \{x\}$
 $\langle proof \rangle$

definition *cond-spmf-fst* :: $('a \times 'b)\ spmf \Rightarrow 'a \Rightarrow 'b\ spmf$ **where**

$cond\text{-}spmf\text{-}fst\ p\ a = map\text{-}spmf\ snd\ (cond\text{-}spmf\ p\ (\{a\} \times UNIV))$

lemma *cond-spmf-fst-return-spmf* [simp]:

$cond\text{-}spmf\text{-}fst\ (return\text{-}spmf\ (x, y))\ x = return\text{-}spmf\ y$
 $\langle proof \rangle$

lemma *cond-spmf-fst-map-Pair* [simp]: $cond\text{-}spmf\text{-}fst\ (map\text{-}spmf\ (Pair\ x)\ p)\ x =$
 $mk\text{-}lossless\ p$

$\langle proof \rangle$

lemma *cond-spmf-fst-map-Pair'* [simp]: $cond\text{-}spmf\text{-}fst\ (map\text{-}spmf\ (\lambda y. (x, f\ y))\ p)$
 $x = map\text{-}spmf\ f\ (mk\text{-}lossless\ p)$

$\langle proof \rangle$

lemma *cond-spmf-fst-eq-return-None* [simp]: $cond\text{-}spmf\text{-}fst\ p\ x = return\text{-}pmf\ None$

$\longleftrightarrow x \notin fst\ `set\text{-}spmf\ p$

$\langle proof \rangle$

lemma *cond-spmf-fst-map-Pair1*:

$cond\text{-}spmf\text{-}fst\ (map\text{-}spmf\ (\lambda x. (f\ x, g\ x))\ p)\ (f\ x) = return\text{-}spmf\ (g\ (inv\text{-}into$
 $(set\text{-}spmf\ p)\ f\ (f\ x)))$

if $x \in set\text{-}spmf\ p\ inj\text{-}on\ f\ (set\text{-}spmf\ p)$

$\langle proof \rangle$

lemma *lossless-cond-spmf-fst* [simp]: $lossless\text{-}spmf\ (cond\text{-}spmf\text{-}fst\ p\ x) \longleftrightarrow x \in fst$
 $\ `set\text{-}spmf\ p$

$\langle proof \rangle$

lemma *cond-spmf-fst-inverse*:

$bind\text{-}spmf\ (map\text{-}spmf\ fst\ p)\ (\lambda x. map\text{-}spmf\ (Pair\ x)\ (cond\text{-}spmf\text{-}fst\ p\ x)) = p$
(is ?lhs = ?rhs)

$\langle proof \rangle$

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 \Rightarrow '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. \exists 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* \equiv *cr-option*

abbreviation *pcr-spmf-option* **where** *pcr-spmf-option* \equiv *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*]:

$((=) \implies \text{cr-spmf-option}) \text{ return-spmf Some}$
 $\langle \text{proof} \rangle$

lemma *map-option-spmf-transfer* [*transfer-parametric map-spmf-parametric, transfer-rule*]:

$((=(=) \implies (=)) \implies \text{cr-spmf-option} \implies \text{cr-spmf-option}) \text{ map-spmf map-option}$
 $\langle \text{proof} \rangle$

lemma *fail-option-spmf-transfer* [*transfer-parametric return-spmf-None-parametric, transfer-rule*]:

$\text{cr-spmf-option (return-pmf None) None}$
 $\langle \text{proof} \rangle$

lemma *bind-option-spmf-transfer* [*transfer-parametric bind-spmf-parametric, transfer-rule*]:

$(\text{cr-spmf-option} \implies ((=) \implies \text{cr-spmf-option}) \implies \text{cr-spmf-option})$
 $\text{bind-spmf Option.bind}$
 $\langle \text{proof} \rangle$

lemma *set-option-spmf-transfer* [*transfer-parametric set-spmf-parametric, transfer-rule*]:

$(\text{cr-spmf-option} \implies \text{rel-set } (=)) \text{ set-spmf set-option}$
 $\langle \text{proof} \rangle$

lemma *rel-option-spmf-transfer* [*transfer-parametric rel-spmf-parametric, transfer-rule*]:

$((=(=) \implies (=) \implies (=)) \implies \text{cr-spmf-option} \implies \text{cr-spmf-option} \implies (=)) \text{ rel-spmf rel-option}$
 $\langle \text{proof} \rangle$

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* :: $'a \text{ option} \Rightarrow 'a \text{ spmf} \Rightarrow \text{bool}$
where $\text{cr-option-le-spmf } x \ p \longleftrightarrow \text{ord-spmf } (=) \text{ (return-pmf } x) \ p$

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

lemma *return-option-le-spmf-transfer* [*transfer-rule*]:

$((=) \implies \text{cr-option-le-spmf}) (\lambda x. x) \text{ return-pmf}$
 $\langle \text{proof} \rangle$

lemma *map-option-le-spmf-transfer* [*transfer-rule*]:

$((=(=) \implies (=)) \implies \text{cr-option-le-spmf} \implies \text{cr-option-le-spmf}) \text{ map-option}$

map-spmf
<proof>

lemma *bind-option-le-spmf-transfer* [*transfer-rule*]:
 $(cr\text{-option-le-spmf} \implies ((=) \implies cr\text{-option-le-spmf}) \implies cr\text{-option-le-spmf})$
Option.bind bind-spmf
<proof>

end

end

interpretation *rel-spmf-characterisation* *<proof>*

lemma *if-distrib-bind-spmf1* [*if-distrib*]:
 $bind\text{-spmf} (if\ b\ then\ x\ else\ y) f = (if\ b\ then\ bind\text{-spmf}\ x\ f\ else\ bind\text{-spmf}\ y\ f)$
<proof>

lemma *if-distrib-bind-spmf2* [*if-distrib*]:
 $bind\text{-spmf}\ x (\lambda y. if\ b\ then\ f\ y\ else\ g\ y) = (if\ b\ then\ bind\text{-spmf}\ x\ f\ else\ bind\text{-spmf}\ x\ g)$
<proof>

lemma *rel-spmf-if-distrib* [*if-distrib*]:
 $rel\text{-spmf}\ R (if\ b\ then\ x\ else\ y) (if\ b\ then\ x'\ else\ y') \longleftrightarrow$
 $(b \longrightarrow rel\text{-spmf}\ R\ x\ x') \wedge (\neg b \longrightarrow rel\text{-spmf}\ R\ y\ y')$
<proof>

lemma *if-distrib-map-spmf* [*if-distrib*]:
 $map\text{-spmf}\ f (if\ b\ then\ p\ else\ q) = (if\ b\ then\ map\text{-spmf}\ f\ p\ else\ map\text{-spmf}\ f\ q)$
<proof>

lemma *if-distrib-restrict-spmf1* [*if-distrib*]:
 $restrict\text{-spmf} (if\ b\ then\ p\ else\ q) A = (if\ b\ then\ restrict\text{-spmf}\ p\ A\ else\ restrict\text{-spmf}\ q\ A)$
<proof>

end

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

1.22 Applicative instance for 'a set

lemma *ap-set-conv-bind*: $ap\text{-set}\ f\ x = Set.bind\ f (\lambda f. Set.bind\ x (\lambda x. \{f\ x\}))$
<proof>

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

lemma *in-ap-setI*: $\llbracket f' \in f; x' \in x \rrbracket \implies f' x' \in f \diamond x$
 ⟨proof⟩

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

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

end

end

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

begin

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

1.23 Applicative instance for 'a spmf

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

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

lemma *ap-spmf-conv-bind*: *ap-spmf* f x = *bind-spmf* f (λf. *bind-spmf* x (λx. *return-spmf* (f x)))
 ⟨proof⟩

ad hoc overloading *Applicative.ap* ⇒ *ap-spmf*

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

lemma *ap-spmf-id*: *pure-spmf* (λx. 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* (λf. f x) ◊ u
 ⟨proof⟩

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

applicative *spmf* (*C*)
for

pure: *pure-spmf*
ap: *ap-spmf*
 $\langle proof \rangle$

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

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

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

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

end

end

1.24 Exclusive or on lists

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

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

lemma *xor-bool-def* [*iff*]: **fixes** *x y* :: *bool* **shows** $x \oplus y \longleftrightarrow x \neq y$
 $\langle proof \rangle$

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

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

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

<proof>

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

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

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

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

definition *xor-list* :: $'a :: \{\text{uminus}, \text{inf}, \text{sup}\}$ list $\Rightarrow 'a$ list $\Rightarrow 'a$ list (**infixr** $\langle [\oplus] \rangle$ 67)
where *xor-list* $xs\ ys = \text{map } (\text{case-prod } (\oplus)) (\text{zip } xs\ ys)$

lemma *xor-list-unfold*:
 $xs\ [\oplus]\ ys = (\text{case } xs\ \text{of } [] \Rightarrow [] \mid x \# xs' \Rightarrow (\text{case } ys\ \text{of } [] \Rightarrow [] \mid y \# ys' \Rightarrow x \oplus y \# xs' [\oplus] ys'))$
<proof>

lemma *xor-list-commute*: **fixes** $xs\ ys :: 'a :: \{\text{semilattice-sup}, \text{semilattice-inf}, \text{uminus}\}$ list
 shows $xs\ [\oplus]\ ys = ys\ [\oplus]\ xs$
<proof>

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

lemma *xor-list-left-commute*:
 fixes $xs\ ys\ zs :: 'a :: \text{boolean-algebra}$ list
 shows $xs\ [\oplus]\ (ys\ [\oplus]\ zs) = ys\ [\oplus]\ (xs\ [\oplus]\ zs)$
<proof>

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

lemma *xor-list-inverse* [*simp*]:
fixes $xs :: 'a :: \text{boolean-algebra list}$
shows $xs [\oplus] xs = \text{replicate } (\text{length } xs) \text{ bot}$
 $\langle \text{proof} \rangle$

lemma *xor-replicate-bot-right* [*simp*]:
fixes $xs :: 'a :: \text{boolean-algebra list}$
shows $\llbracket \text{length } xs \leq n; x = \text{bot} \rrbracket \implies xs [\oplus] \text{replicate } n \ x = xs$
 $\langle \text{proof} \rangle$

lemma *xor-replicate-bot-left* [*simp*]:
fixes $xs :: 'a :: \text{boolean-algebra list}$
shows $\llbracket \text{length } xs \leq n; x = \text{bot} \rrbracket \implies \text{replicate } n \ x [\oplus] xs = xs$
 $\langle \text{proof} \rangle$

lemma *xor-list-left-inverse* [*simp*]:
fixes $xs :: 'a :: \text{boolean-algebra list}$
shows $\text{length } ys \leq \text{length } xs \implies xs [\oplus] (xs [\oplus] ys) = ys$
 $\langle \text{proof} \rangle$

lemma *length-xor-list* [*simp*]: $\text{length } (\text{xor-list } xs \ ys) = \min (\text{length } xs) (\text{length } ys)$
 $\langle \text{proof} \rangle$

lemma *inj-on-xor-list-nlists* [*simp*]:
fixes $xs :: 'a :: \text{boolean-algebra list}$
shows $n \leq \text{length } xs \implies \text{inj-on } (\text{xor-list } xs) (\text{nlists UNIV } n)$
 $\langle \text{proof} \rangle$

lemma *one-time-pad*:
fixes $xs :: - :: \text{boolean-algebra list}$
shows $\text{length } xs \geq n \implies \text{map-spmf } (\text{xor-list } xs) (\text{spmof-of-set } (\text{nlists UNIV } n))$
 $= \text{spmof-of-set } (\text{nlists UNIV } n)$
 $\langle \text{proof} \rangle$

end

theory *Environment-Function imports*
Applicative-Lifting.Applicative-Environment
begin

1.25 The environment functor

type-synonym $(i, 'a) \text{ envir} = i \Rightarrow 'a$

lemma *const-apply* [*simp*]: $\text{const } x \ i = x$
 $\langle \text{proof} \rangle$

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* ⇒ bool
where *all-envir* p ⇔ (∀ x. p x)

lemma *all-envirI* [*Pure.intro!*, *intro!*]: (∧ x. p x) ⇒ *all-envir* p
<proof>

lemma *all-envirE* [*Pure.elim 2*, *elim*]: *all-envir* p ⇒ (p x ⇒ *thesis*) ⇒ *thesis*
<proof>

lemma *all-envirD*: *all-envir* p ⇒ p x
<proof>

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

lemma *pred-envir-conv*: *pred-envir* p f ⇔ (∀ x. p (f x))
<proof>

lemma *pred-envirI* [*Pure.intro!*, *intro!*]: (∧ x. p (f x)) ⇒ *pred-envir* p f
<proof>

lemma *pred-envirD*: *pred-envir* p f ⇒ p (f x)
<proof>

lemma *pred-envirE* [*Pure.elim 2*, *elim*]: *pred-envir* p f ⇒ (p (f x) ⇒ *thesis*)
⇒ *thesis*
<proof>

lemma *pred-envir-mono*: [*pred-envir* p f; ∧ x. p (f x) ⇒ q (g x)] ⇒ *pred-envir* q g
<proof>

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

lemma *rel-envir-conv*: *rel-envir* p f g ⇔ (∀ x. p (f x) (g x))
<proof>

lemma *rel-envir-conv-rel-fun*: *rel-envir* = *rel-fun* (=)
<proof>

lemma *rel-envirI* [*Pure.intro!*, *intro!*]: (∧ x. p (f x) (g x)) ⇒ *rel-envir* p f g
<proof>

lemma *rel-envirD*: *rel-envir* p f g ⇒ p (f x) (g x)
<proof>

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

lemma *rel-envir-mono*: \llbracket *rel-envir p f g*; $\bigwedge x. p (f x) (g x) \implies q (f' x) (g' x)$ $\rrbracket \implies$
rel-envir q f' g'
 ⟨*proof*⟩

lemma *rel-envir-mono1*: \llbracket *pred-envir p f*; $\bigwedge x. p (f x) \implies q (f' x) (g' x)$ $\rrbracket \implies$
rel-envir q f' g'
 ⟨*proof*⟩

lemma *pred-envir-mono2*: \llbracket *rel-envir p f g*; $\bigwedge x. p (f x) (g x) \implies q (f' x)$ $\rrbracket \implies$
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 (\leq) *Sup*
 ⟨*proof*⟩

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

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

lemma *set-admissible*: *set.admissible* ($\lambda f :: 'a \Rightarrow 'b \text{ set}. \forall x y. y \in f x \longrightarrow P x y$)
 ⟨*proof*⟩

abbreviation *mono-set* \equiv *monotone* (*fun-ord* (\subseteq)) (\subseteq)

lemma *fixp-induct-set-scott*:

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*: $\bigwedge x. \text{mono-set } (\lambda f. U (F (C f)) x)$

and *eq*: *f* \equiv *C* (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* (\leq)) ($\lambda f. U (F (C f))$))

and *inverse2*: $\bigwedge f. U (C f) = f$
and *step*: $\bigwedge f x y. [\bigwedge x y. y \in U f x \implies P x y; y \in U (F f) x] \implies P x y$
and *enforce-variable-ordering*: $x = x$
and *elem*: $y \in U f x$
shows $P x y$
 $\langle proof \rangle$

lemma *fixp-Sup-le*:
defines $le \equiv ((\leq) :: - :: complete-lattice \Rightarrow -)$
shows $ccpo.fixp\ Sup\ le = cppo-class.fixp$
 $\langle proof \rangle$

lemma *fun-ord-le*: $fun-ord (\leq) = (\leq)$
 $\langle proof \rangle$

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*: $\bigwedge x. mono-set (\lambda f. U (F (C f)) x)$
and *eq*: $f \equiv C (ccpo.fixp (fun-lub\ Sup) (fun-ord (\leq)) (\lambda f. U (F (C f))))$
and *inverse2*: $\bigwedge f. U (C f) = f$

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

$\langle ML \rangle$

lemma [*partial-function-mono*]:
shows *insert-mono*: $mono-set A \implies mono-set (\lambda f. insert\ x\ (A\ f))$
and *UNION-mono*: $[\mathbb{[} mono-set\ B; \bigwedge y. mono-set (\lambda f. C\ y\ f) \mathbb{]}} \implies mono-set (\lambda f. \bigcup_{y \in B} f. C\ y\ f)$
and *set-bind-mono*: $[\mathbb{[} mono-set\ B; \bigwedge y. mono-set (\lambda f. C\ y\ f) \mathbb{]}} \implies mono-set (\lambda f. Set.bind\ (B\ f)\ (\lambda y. C\ y\ f))$
and *Un-mono*: $[\mathbb{[} mono-set\ A; mono-set\ B \mathbb{]}} \implies mono-set (\lambda f. A\ f \cup B\ f)$
and *Int-mono*: $[\mathbb{[} mono-set\ A; mono-set\ B \mathbb{]}} \implies mono-set (\lambda f. A\ f \cap B\ f)$
and *Diff-mono1*: $mono-set A \implies mono-set (\lambda f. A\ f - X)$
and *image-mono*: $mono-set A \implies mono-set (\lambda f. g\ 'A\ f)$
and *vimage-mono*: $mono-set A \implies mono-set (\lambda f. g\ -'A\ f)$
 $\langle proof \rangle$

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

where

$test\ xs\ i\ j = insert\ 4\ (test\ []\ 0\ j \cup test\ []\ 1\ True \cap test\ []\ 2\ False - \{5\} \cup uminus$
 $'\ test\ [undefined]\ 0\ True \cup uminus - '\ test\ []\ 1\ False)$

interpretation *coset: partial-function-definitions* (\supseteq) *Inter*
(*proof*)

lemma *fun-lub-Inf: fun-lub Inf = (Inf :: - \Rightarrow - :: complete-lattice)*
(*proof*)

lemma *fun-ord-ge: fun-ord (\geq) = (\geq)*
(*proof*)

lemma *coset-admissible: coset.admissible ($\lambda f :: 'a \Rightarrow 'b\ set. \forall x\ y. P\ x\ y \longrightarrow y \in f\ x$)*
(*proof*)

abbreviation *mono-coset* \equiv *monotone (fun-ord (\supseteq))* (\supseteq)

lemma *gfp-eq-fixp:*
fixes $f :: 'a :: complete-lattice \Rightarrow 'a$
assumes $f: monotone\ (\geq)\ (\geq)\ f$
shows $gfp\ f = ccpo.fixp\ Inf\ (\geq)\ f$
(*proof*)

lemma *fixp-coinduct-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*: $\bigwedge x. mono-coset\ (\lambda f. U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C\ (ccpo.fixp\ (fun-lub\ Inter)\ (fun-ord\ (\geq))\ (\lambda f. U\ (F\ (C\ f))))$
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'* \equiv *monotone (fun-ord (\supseteq))* (\supseteq)

lemma [*partial-function-mono*]:
shows *insert-mono'*: $mono-set'\ A \Longrightarrow mono-set'\ (\lambda f. insert\ x\ (A\ f))$
and *UNION-mono'*: $[\![mono-set'\ B; \bigwedge y. mono-set'\ (\lambda f. C\ y\ f)]\!] \Longrightarrow mono-set'$

$(\lambda f. \bigcup_{y \in B} f. C y f)$
and *set-bind-mono'*: $\llbracket \text{mono-set}' B; \bigwedge y. \text{mono-set}' (\lambda f. C y f) \rrbracket \implies \text{mono-set}'$
 $(\lambda f. \text{Set.bind } (B f) (\lambda y. C y f))$
and *Un-mono'*: $\llbracket \text{mono-set}' A; \text{mono-set}' B \rrbracket \implies \text{mono-set}' (\lambda f. A f \cup B f)$
and *Int-mono'*: $\llbracket \text{mono-set}' A; \text{mono-set}' B \rrbracket \implies \text{mono-set}' (\lambda f. A f \cap B f)$
 $\langle \text{proof} \rangle$

context begin

private partial-function (*coset*) *test2* :: $\text{nat} \Rightarrow \text{nat set}$

where *test2* $x = \text{insert } x (\text{test2 } (\text{Suc } x))$

private lemma *test2-coinduct*:

assumes $P x y$

and $*$: $\bigwedge x y. P x y \implies y = x \vee (P (\text{Suc } x) y \vee y \in \text{test2 } (\text{Suc } x))$

shows $y \in \text{test2 } x$

$\langle \text{proof} \rangle$

end

end

2 Negligibility

theory *Negligible* **imports**

Complex-Main

Landau-Symbols.Landau-More

begin

named-theorems *negligible-intros*

definition *negligible* :: $(\text{nat} \Rightarrow \text{real}) \Rightarrow \text{bool}$

where *negligible* $f \longleftrightarrow (\forall c > 0. f \in o(\lambda x. \text{inverse } (x \text{ powr } c)))$

lemma *negligibleI* [*intro?*]:

$(\bigwedge c. c > 0 \implies f \in o(\lambda x. \text{inverse } (x \text{ powr } c))) \implies \text{negligible } f$

$\langle \text{proof} \rangle$

lemma *negligibleD*:

$\llbracket \text{negligible } f; c > 0 \rrbracket \implies f \in o(\lambda x. \text{inverse } (x \text{ powr } c))$

$\langle \text{proof} \rangle$

lemma *negligibleD-real*:

assumes *negligible* f

shows $f \in o(\lambda x. \text{inverse } (x \text{ powr } c))$

$\langle \text{proof} \rangle$

lemma *negligible-mono*: $\llbracket \text{negligible } g; f \in O(g) \rrbracket \implies \text{negligible } f$

$\langle \text{proof} \rangle$

lemma *negligible-le*: $\llbracket \text{negligible } g; \bigwedge \eta. |f \ \eta| \leq g \ \eta \rrbracket \implies \text{negligible } f$
 <proof>

lemma *negligible-K0* [*negligible-intros, simp, intro!*]: *negligible* ($\lambda-. 0$)
 <proof>

lemma *negligible-0* [*negligible-intros, simp, intro!*]: *negligible* 0
 <proof>

lemma *negligible-const-iff* [*simp*]: *negligible* ($\lambda-. c :: \text{real}$) $\longleftrightarrow c = 0$
 <proof>

lemma *not-negligible-1*: $\neg \text{negligible } (\lambda-. 1 :: \text{real})$
 <proof>

lemma *negligible-plus* [*negligible-intros*]:
 $\llbracket \text{negligible } f; \text{negligible } g \rrbracket \implies \text{negligible } (\lambda \eta. f \ \eta + g \ \eta)$
 <proof>

lemma *negligible-uminus* [*simp*]: *negligible* ($\lambda \eta. - f \ \eta$) $\longleftrightarrow \text{negligible } f$
 <proof>

lemma *negligible-uminusI* [*negligible-intros*]: *negligible* $f \implies \text{negligible } (\lambda \eta. - f \ \eta)$
 <proof>

lemma *negligible-minus* [*negligible-intros*]:
 $\llbracket \text{negligible } f; \text{negligible } g \rrbracket \implies \text{negligible } (\lambda \eta. f \ \eta - g \ \eta)$
 <proof>

lemma *negligible-cmult*: *negligible* ($\lambda \eta. c * f \ \eta$) $\longleftrightarrow \text{negligible } f \vee c = 0$
 <proof>

lemma *negligible-cmultI* [*negligible-intros*]:
 $(c \neq 0 \implies \text{negligible } f) \implies \text{negligible } (\lambda \eta. c * f \ \eta)$
 <proof>

lemma *negligible-multc*: *negligible* ($\lambda \eta. f \ \eta * c$) $\longleftrightarrow \text{negligible } f \vee c = 0$
 <proof>

lemma *negligible-multcI* [*negligible-intros*]:
 $(c \neq 0 \implies \text{negligible } f) \implies \text{negligible } (\lambda \eta. f \ \eta * c)$
 <proof>

lemma *negligible-times* [*negligible-intros*]:
assumes $f: \text{negligible } f$
and $g: \text{negligible } g$
shows *negligible* ($\lambda \eta. f \ \eta * g \ \eta :: \text{real}$)
 <proof>

lemma *negligible-power* [*negligible-intros*]:
assumes *negligible f*
and $n > 0$
shows *negligible* $(\lambda\eta. f \eta \wedge n :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *negligible-powr* [*negligible-intros*]:
assumes $f: \text{negligible } f$
and $p: p > 0$
shows *negligible* $(\lambda x. |f x| \text{ powr } p :: \text{real})$
 $\langle \text{proof} \rangle$

lemma *negligible-abs* [*simp*]: *negligible* $(\lambda x. |f x|) \longleftrightarrow \text{negligible } f$
 $\langle \text{proof} \rangle$

lemma *negligible-absI* [*negligible-intros*]: *negligible* $f \implies \text{negligible } (\lambda x. |f x|)$
 $\langle \text{proof} \rangle$

lemma *negligible-powrI* [*negligible-intros*]:
assumes $0 \leq k \ k < 1$
shows *negligible* $(\lambda x. k \text{ powr } x)$
 $\langle \text{proof} \rangle$

lemma *negligible-powerI* [*negligible-intros*]:
fixes $k :: \text{real}$
assumes $|k| < 1$
shows *negligible* $(\lambda n. k \wedge n)$
 $\langle \text{proof} \rangle$

lemma *negligible-inverse-powerI* [*negligible-intros*]: $|k| > 1 \implies \text{negligible } (\lambda\eta. 1 / k \wedge \eta)$
 $\langle \text{proof} \rangle$

inductive *polynomial* :: $(\text{nat} \Rightarrow \text{real}) \Rightarrow \text{bool}$
for f
where $f \in O(\lambda x. x \text{ powr } n) \implies \text{polynomial } f$

lemma *negligible-times-poly*:
assumes $f: \text{negligible } f$
and $g: g \in O(\lambda x. x \text{ powr } n)$
shows *negligible* $(\lambda x. f x * g x)$
 $\langle \text{proof} \rangle$

lemma *negligible-poly-times*:
 $\llbracket f \in O(\lambda x. x \text{ powr } n); \text{negligible } g \rrbracket \implies \text{negligible } (\lambda x. f x * g x)$
 $\langle \text{proof} \rangle$

lemma *negligible-times-polynomial* [*negligible-intros*]:
 $\llbracket \text{negligible } f; \text{polynomial } g \rrbracket \implies \text{negligible } (\lambda x. f x * g x)$

<proof>

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

$\llbracket \text{polynomial } f; \text{negligible } g \rrbracket \implies \text{negligible } (\lambda x. f\ x * g\ x)$

<proof>

lemma *negligible-divide-poly1*:

$\llbracket f \in O(\lambda x. x \text{ powr } n); \text{negligible } (\lambda \eta. 1 / g\ \eta) \rrbracket \implies \text{negligible } (\lambda \eta. \text{real } (f\ \eta) / g\ \eta)$

<proof>

lemma *negligible-divide-polynomial1* [*negligible-intros*]:

$\llbracket \text{polynomial } f; \text{negligible } (\lambda \eta. 1 / g\ \eta) \rrbracket \implies \text{negligible } (\lambda \eta. \text{real } (f\ \eta) / g\ \eta)$

<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*) = (*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

$\llbracket \text{is-Done } x; \text{result } x \neq \text{None} \longrightarrow \text{is-Done } (f\ (\text{the } (\text{result } x))) \rrbracket \implies \text{is-Done } (\text{bind-resumption } x\ f)$

| *result* (*bind-resumption x f*) = *result x* \gg *result* \circ *f*

| *output* (*bind-resumption x f*) = (*if is-Done x then output (f (the (result x))) else output x*)

| *resume* (*bind-resumption x f*) = (*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* \equiv *bind-resumption*

lemma *is-Done-bind-resumption* [simp]:
 $is\text{-}Done\ (x \gg f) \longleftrightarrow is\text{-}Done\ x \wedge (result\ x \neq None \longrightarrow is\text{-}Done\ (f\ (the\ (result\ x))))$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

lemma *results-ABORT* [*simp*]: *results* *ABORT* = $\{\}$
 \langle *proof* \rangle

lemma *outputs-ABORT* [*simp*]: *outputs* *ABORT* = $\{\}$
 \langle *proof* \rangle

lemma *outputs-DONE* [*simp*]: *outputs* (*DONE* x) = $\{\}$
 \langle *proof* \rangle

lemma *is-Done-cases* [*cases pred*]:
assumes *is-Done* r
obtains (*DONE*) x **where** $r = \text{DONE } x \mid (\text{ABORT}) r = \text{ABORT}$
 \langle *proof* \rangle

lemma *not-is-Done-conv-Pause*: $\neg \text{is-Done } r \longleftrightarrow (\exists \text{ out } c. r = \text{Pause out } c)$
 \langle *proof* \rangle

lemma *Done-bind* [*code*]:
 $\text{Done } a \ggg f = (\text{case } a \text{ of } \text{None} \Rightarrow \text{Done None} \mid \text{Some } a \Rightarrow f a)$
 \langle *proof* \rangle

lemma *DONE-bind* [*simp*]:
 $\text{DONE } a \ggg f = f a$
 \langle *proof* \rangle

lemma *bind-resumption-Pause* [*simp, code*]: **fixes** *cont* **shows**
 $\text{Pause out } cont \ggg f$
 $= \text{Pause out } (\lambda \text{ inp. } cont \text{ inp } \ggg f)$
 \langle *proof* \rangle

lemma *bind-DONE* [*simp*]:
 $x \ggg \text{DONE} = x$
 \langle *proof* \rangle

lemma *bind-bind-resumption*:
fixes $r :: ('a, 'in, 'out) \text{ resumption}$
shows $(r \ggg f) \ggg g = \text{do } \{ x \leftarrow r; f x \ggg g \}$
 \langle *proof* \rangle

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

lemma *ABORT-bind* [*simp*]: *ABORT* $\ggg f = \text{ABORT}$
 \langle *proof* \rangle

lemma *bind-resumption-is-Done*: $is\text{-}Done\ f \implies f \ggg g = (if\ result\ f = None\ then\ ABORT\ else\ g\ (the\ (result\ f)))$

<proof>

lemma *bind-resumption-eq-Done-iff* [*simp*]:

$f \ggg g = Done\ x \iff (\exists y. f = DONE\ y \wedge g\ y = Done\ x) \vee f = ABORT \wedge x = None$

<proof>

lemma *bind-resumption-cong*:

assumes $x = y$

and $\bigwedge z. z \in results\ y \implies f\ z = g\ z$

shows $x \ggg f = y \ggg g$

<proof>

lemma *results-bind-resumption*:

$results\ (bind\text{-}resumption\ x\ f) = (\bigcup a \in results\ x. results\ (f\ a))$

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

<proof>

lemma *outputs-bind-resumption* [*simp*]:

$outputs\ (bind\text{-}resumption\ r\ f) = outputs\ r \cup (\bigcup x \in results\ r. outputs\ (f\ x))$

(**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*]:

$is\text{-}Done\ (map\text{-}resumption\ f1\ f2\ r) \iff is\text{-}Done\ r$

<proof>

lemma *result-map-resumption* [*simp*]:

$is\text{-}Done\ r \implies result\ (map\text{-}resumption\ f1\ f2\ r) = map\text{-}option\ f1\ (result\ r)$

<proof>

lemma *output-map-resumption* [*simp*]:

$\neg is\text{-}Done\ r \implies output\ (map\text{-}resumption\ f1\ f2\ r) = f2\ (output\ r)$

<proof>

lemma *resume-map-resumption* [*simp*]:

$\neg is\text{-}Done\ r$

$\implies resume\ (map\text{-}resumption\ f1\ f2\ r) = map\text{-}resumption\ f1\ f2 \circ resume\ r$

<proof>

lemma *rel-resumption-is-DoneD*: $rel\text{-}resumption\ A\ B\ r1\ r2 \implies is\text{-}Done\ r1 \iff is\text{-}Done\ r2$

<proof>

lemma *rel-resumption-resultD1*:

$\llbracket \text{rel-resumption } A \ B \ r1 \ r2; \text{is-Done } r1 \rrbracket \implies \text{rel-option } A \ (\text{result } r1) \ (\text{result } r2)$
<proof>

lemma *rel-resumption-resultD2*:

$\llbracket \text{rel-resumption } A \ B \ r1 \ r2; \text{is-Done } r2 \rrbracket \implies \text{rel-option } A \ (\text{result } r1) \ (\text{result } r2)$
<proof>

lemma *rel-resumption-outputD1*:

$\llbracket \text{rel-resumption } A \ B \ r1 \ r2; \neg \text{is-Done } r1 \rrbracket \implies B \ (\text{output } r1) \ (\text{output } r2)$
<proof>

lemma *rel-resumption-outputD2*:

$\llbracket \text{rel-resumption } A \ B \ r1 \ r2; \neg \text{is-Done } r2 \rrbracket \implies B \ (\text{output } r1) \ (\text{output } r2)$
<proof>

lemma *rel-resumption-resumeD1*:

$\llbracket \text{rel-resumption } A \ B \ r1 \ r2; \neg \text{is-Done } r1 \rrbracket$
 $\implies \text{rel-resumption } A \ B \ (\text{resume } r1 \ \text{inp}) \ (\text{resume } r2 \ \text{inp})$
<proof>

lemma *rel-resumption-resumeD2*:

$\llbracket \text{rel-resumption } A \ B \ r1 \ r2; \neg \text{is-Done } r2 \rrbracket$
 $\implies \text{rel-resumption } A \ B \ (\text{resume } r1 \ \text{inp}) \ (\text{resume } r2 \ \text{inp})$
<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: $\bigwedge r1 \ r2. X \ r1 \ r2 \implies (\text{is-Done } r1 \longleftrightarrow \text{is-Done } r2) \wedge (\text{is-Done } r1 \longrightarrow \text{is-Done } r2 \longrightarrow \text{rel-option } A \ (\text{result } r1) \ (\text{result } r2))$

and Pause: $\bigwedge r1 \ r2. \llbracket X \ r1 \ r2; \neg \text{is-Done } r1; \neg \text{is-Done } r2 \rrbracket \implies B \ (\text{output } r1) \ (\text{output } r2) \wedge (\forall \text{inp}. X \ (\text{resume } r1 \ \text{inp}) \ (\text{resume } r2 \ \text{inp}))$

shows $\text{rel-resumption } A \ B \ r1 \ r2$

<proof>

3.1 Setup for partial-function

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

coinductive *resumption-ord* :: ('a, 'out, 'in) *resumption* \Rightarrow ('a, 'out, 'in) *resumption* \Rightarrow bool

where

Done-Done: $\text{flat-ord } \text{None } a \ a' \implies \text{resumption-ord } (\text{Done } a) \ (\text{Done } a')$

| *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*:
 $\llbracket \text{resumption-ord } r \ r'; \text{ is-Done } r' \rrbracket \implies \text{is-Done } r$
 <proof>

lemma *resumption-ord-resultD*:
 $\llbracket \text{resumption-ord } r \ r'; \text{ is-Done } r' \rrbracket \implies \text{flat-ord None (result } r) \text{ (result } r')$
 <proof>

lemma *resumption-ord-outputD*:
 $\llbracket \text{resumption-ord } r \ r'; \neg \text{is-Done } r \rrbracket \implies \text{output } r = \text{output } r'$
 <proof>

lemma *resumption-ord-resumeD*:
 $\llbracket \text{resumption-ord } r \ r'; \neg \text{is-Done } r \rrbracket \implies ((=) ===> \text{resumption-ord}) \text{ (resume } r) \text{ (resume } r')$
 <proof>

lemma *resumption-ord-abort*:
 $\llbracket \text{resumption-ord } r \ r'; \text{ is-Done } r; \neg \text{is-Done } r' \rrbracket \implies \text{result } r = \text{None}$
 <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*: $\bigwedge r \ r'. \llbracket X \ r \ r'; \text{ is-Done } r' \rrbracket \implies \text{is-Done } r \wedge \text{flat-ord None (result } r) \text{ (result } r')$
and *Abort*: $\bigwedge r \ r'. \llbracket X \ r \ r'; \neg \text{is-Done } r'; \text{ is-Done } r \rrbracket \implies \text{result } r = \text{None}$
and *Pause*: $\bigwedge r \ r'. \llbracket X \ r \ r'; \neg \text{is-Done } r; \neg \text{is-Done } r' \rrbracket \implies \text{output } r = \text{output } r' \wedge ((=) ===> (\lambda r \ r'. X \ r \ r' \vee \text{resumption-ord } r \ r'))$
 (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* $\longleftrightarrow r = \text{ABORT}$
 <proof>

lemma *resumption-ord-DONE1* [simp]: *resumption-ord* (DONE *x*) *r* \longleftrightarrow *r* =
 DONE *x*
 <proof>

lemma *resumption-ord-refl*: *resumption-ord* *r r*
 <proof>

lemma *resumption-ord-antisym*:
 [*resumption-ord* *r r'*; *resumption-ord* *r' r*]
 $\implies r = r'$
 <proof>

lemma *resumption-ord-trans*:
 [*resumption-ord* *r r'*; *resumption-ord* *r' r''*]
 \implies *resumption-ord* *r r''*
 <proof>

primcorec *resumption-lub* :: ('a, 'out, 'in) *resumption set* \Rightarrow ('a, 'out, 'in) *re-*
sumption

where

$\forall r \in R. \text{is-Done } r \implies \text{is-Done } (\text{resumption-lub } R)$
 | *result* (*resumption-lub* *R*) = *flat-lub* None (*result* ' *R*)
 | *output* (*resumption-lub* *R*) = (THE *out. out* \in *output* ' (*R* \cap {*r. \neg is-Done r*}))
 | *resume* (*resumption-lub* *R*) = (λ *inp. resumption-lub* ((λ *c. c inp*) ' *resume* ' (*R* \cap
 {*r. \neg is-Done r*})))

lemma *is-Done-resumption-lub* [simp]:
is-Done (*resumption-lub* *R*) \longleftrightarrow ($\forall r \in R. \text{is-Done } r$)
 <proof>

lemma *result-resumption-lub* [simp]:
 $\forall r \in R. \text{is-Done } r \implies \text{result } (\text{resumption-lub } R) = \text{flat-lub None } (\text{result } ' R)$
 <proof>

lemma *output-resumption-lub* [simp]:
 $\exists r \in R. \neg \text{is-Done } r \implies \text{output } (\text{resumption-lub } R) = (\text{THE } \text{out. out} \in \text{output } ' (R \cap \{r. \neg \text{is-Done } r\}))$
 <proof>

lemma *resume-resumption-lub* [simp]:
 $\exists r \in R. \neg \text{is-Done } r$
 $\implies \text{resume } (\text{resumption-lub } R) \text{ inp} =$
 $\text{resumption-lub } ((\lambda c. c \text{ inp}) ' \text{resume } ' (R \cap \{r. \neg \text{is-Done } r\}))$
 <proof>

lemma *resumption-lub-empty*: *resumption-lub* {} = ABORT
 <proof>

context

```

fixes R state inp R'
defines R'-def:  $R' \equiv (\lambda c. c \text{ inp}) \text{ ' resume ' } (R \cap \{r. \neg \text{is-Done } r\})$ 
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  $\equiv$  monotone (fun-ord resumption-ord) resump-
tion-ord

lemma mono-resumption-resume:
assumes mono-resumption B
shows mono-resumption ( $\lambda f. \text{resume } (B f) \text{ inp}$ )
<proof>

lemma bind-resumption-mono [partial-function-mono]:
assumes mf: mono-resumption B
and mg:  $\bigwedge y. \text{mono-resumption } (C y)$ 
shows mono-resumption ( $\lambda f. \text{do } \{ y \leftarrow B f; C y f \}$ )
<proof>

lemma fixes f F
defines  $F \equiv \lambda \text{results } r. \text{case } r \text{ of } \text{resumption.Done } x \Rightarrow \text{set-option } x \mid \text{resump-}$ 
 $\text{tion.Pause out } c \Rightarrow \bigcup \text{input. results } (c \text{ input})$ 
shows results-conv-fixp:  $\text{results} \equiv \text{ccpo.fixp } (\text{fun-lub Union}) (\text{fun-ord } (\subseteq)) F$  (is
 $- \equiv ?\text{fixp}$ )
and results-mono:  $\bigwedge x. \text{monotone } (\text{fun-ord } (\subseteq)) (\subseteq) (\lambda f. F f x)$  (is PROP ?mono)
<proof>

lemma mcont-case-resumption:
fixes f g
defines  $h \equiv \lambda r. \text{if is-Done } r \text{ then } f (\text{result } r) \text{ else } g (\text{output } r) (\text{resume } r)$ 
assumes mcont1: mcont (flat-lub None) option-ord lub ord f
and mcont2:  $\bigwedge \text{out. mcont } (\text{fun-lub resumption-lub}) (\text{fun-ord resumption-ord}) \text{ lub}$ 

```

ord ($\lambda c. g \text{ out } c \text{ (Pause out } c)$)
and *ccpo*: *class.ccpo lub ord (mk-less ord)*
and *bot*: $\bigwedge x. \text{ord } (f \text{ None}) x$
shows *mcont resumption-lub resumption-ord lub ord* ($\lambda r. \text{case } r \text{ of Done } x \Rightarrow f x$
 $| \text{Pause out } c \Rightarrow g \text{ out } c r$)
(is mcont ?lub ?ord - - ?f)
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-results*[*THEN mcont2mcont, cont-intro, simp*]:
shows *mcont-results: mcont resumption-lub resumption-ord Union* (\subseteq) *results*
 $\langle \text{proof} \rangle$

lemma *mono2mono-results*[*THEN lfp.mono2mono, cont-intro, simp*]:
shows *monotone-results: monotone resumption-ord* (\subseteq) *results*
 $\langle \text{proof} \rangle$

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

lemma *mcont2mcont-outputs*[*THEN lfp.mcont2mcont, cont-intro, simp*]:
shows *mcont-outputs: mcont resumption-lub resumption-ord Union* (\subseteq) *outputs*
 $\langle \text{proof} \rangle$

lemma *mono2mono-outputs*[*THEN lfp.mono2mono, cont-intro, simp*]:
shows *monotone-outputs: monotone resumption-ord* (\subseteq) *outputs*
 $\langle \text{proof} \rangle$

lemma *pred-resumption-antimono*:
assumes *r: pred-resumption A C r'*
and *le: resumption-ord r r'*
shows *pred-resumption A C r*
 $\langle \text{proof} \rangle$

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*]:
 $\text{rel-resumption } A B \text{ OO rel-resumption } C D = \text{rel-resumption } (A \text{ OO } C) (B \text{ OO } D)$
 $\langle \text{proof} \rangle$

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

$\llbracket \text{left-total } R1; \text{left-total } R2 \rrbracket \implies \text{left-total } (\text{rel-resumption } R1 \ R2)$
 $\langle \text{proof} \rangle$

lemma *left-unique-rel-resumption* [*transfer-rule*]:
 $\llbracket \text{left-unique } R1; \text{left-unique } R2 \rrbracket \implies \text{left-unique } (\text{rel-resumption } R1 \ R2)$
 $\langle \text{proof} \rangle$

lemma *right-total-rel-resumption* [*transfer-rule*]:
 $\llbracket \text{right-total } R1; \text{right-total } R2 \rrbracket \implies \text{right-total } (\text{rel-resumption } R1 \ R2)$
 $\langle \text{proof} \rangle$

lemma *right-unique-rel-resumption* [*transfer-rule*]:
 $\llbracket \text{right-unique } R1; \text{right-unique } R2 \rrbracket \implies \text{right-unique } (\text{rel-resumption } R1 \ R2)$
 $\langle \text{proof} \rangle$

lemma *bi-total-rel-resumption* [*transfer-rule*]:
 $\llbracket \text{bi-total } A; \text{bi-total } B \rrbracket \implies \text{bi-total } (\text{rel-resumption } A \ B)$
 $\langle \text{proof} \rangle$

lemma *bi-unique-rel-resumption* [*transfer-rule*]:
 $\llbracket \text{bi-unique } A; \text{bi-unique } B \rrbracket \implies \text{bi-unique } (\text{rel-resumption } A \ B)$
 $\langle \text{proof} \rangle$

lemma *Quotient-resumption* [*quot-map*]:
 $\llbracket \text{Quotient } R1 \text{ Abs1 Rep1 } T1; \text{Quotient } R2 \text{ Abs2 Rep2 } T2 \rrbracket$
 $\implies \text{Quotient } (\text{rel-resumption } R1 \ R2) (\text{map-resumption Abs1 Abs2}) (\text{map-resumption Rep1 Rep2}) (\text{rel-resumption } T1 \ T2)$
 $\langle \text{proof} \rangle$

end

4 Generative probabilistic values

theory *Generat imports*

Misc-CryptHOL

begin

4.1 Single-step generative

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

datatype-compat *generat*

lemma *IO-code-cong*: $\text{out} = \text{out}' \implies \text{IO out } c = \text{IO out}' \ c$ $\langle \text{proof} \rangle$
 $\langle \text{ML} \rangle$

lemma *is-Pure-map-generat* [*simp*]: $\text{is-Pure } (\text{map-generat } f \ g \ h \ x) = \text{is-Pure } x$

$\langle \text{proof} \rangle$

lemma *result-map-generat* [*simp*]: $\text{is-Pure } x \implies \text{result } (\text{map-generat } f \ g \ h \ x) = f$
 $(\text{result } x)$
 $\langle \text{proof} \rangle$

lemma *output-map-generat* [*simp*]: $\neg \text{is-Pure } x \implies \text{output } (\text{map-generat } f \ g \ h \ x)$
 $= g \ (\text{output } x)$
 $\langle \text{proof} \rangle$

lemma *continuation-map-generat* [*simp*]: $\neg \text{is-Pure } x \implies \text{continuation } (\text{map-generat}$
 $f \ g \ h \ x) = h \ (\text{continuation } x)$
 $\langle \text{proof} \rangle$

lemma [*simp*]:

shows *map-generat-eq-Pure*:

$\text{map-generat } f \ g \ h \ \text{generat} = \text{Pure } x \iff (\exists x'. \text{generat} = \text{Pure } x' \wedge x = f \ x')$

and *Pure-eq-map-generat*:

$\text{Pure } x = \text{map-generat } f \ g \ h \ \text{generat} \iff (\exists x'. \text{generat} = \text{Pure } x' \wedge x = f \ x')$

$\langle \text{proof} \rangle$

lemma [*simp*]:

shows *map-generat-eq-IO*:

$\text{map-generat } f \ g \ h \ \text{generat} = \text{IO } \text{out } c \iff (\exists \text{out}' \ c'. \text{generat} = \text{IO } \text{out}' \ c' \wedge \text{out}$
 $= g \ \text{out}' \wedge c = h \ c')$

and *IO-eq-map-generat*:

$\text{IO } \text{out } c = \text{map-generat } f \ g \ h \ \text{generat} \iff (\exists \text{out}' \ c'. \text{generat} = \text{IO } \text{out}' \ c' \wedge \text{out}$
 $= g \ \text{out}' \wedge c = h \ c')$

$\langle \text{proof} \rangle$

lemma *is-PureE* [*cases pred*]:

assumes *is-Pure generat*

obtains $(\text{Pure}) \ x$ **where** $\text{generat} = \text{Pure } x$

$\langle \text{proof} \rangle$

lemma *not-is-PureE*:

assumes $\neg \text{is-Pure } \text{generat}$

obtains $(\text{IO}) \ \text{out } c$ **where** $\text{generat} = \text{IO } \text{out } c$

$\langle \text{proof} \rangle$

lemma *rel-generatI*:

$\llbracket \text{is-Pure } x \iff \text{is-Pure } y;$

$\llbracket \text{is-Pure } x; \text{is-Pure } y \rrbracket \implies A \ (\text{result } x) \ (\text{result } y);$

$\llbracket \neg \text{is-Pure } x; \neg \text{is-Pure } y \rrbracket \implies \text{Out } (\text{output } x) \ (\text{output } y) \wedge R \ (\text{continuation}$
 $x) \ (\text{continuation } y) \rrbracket$

$\implies \text{rel-generat } A \ \text{Out } R \ x \ y$

$\langle \text{proof} \rangle$

lemma *rel-generatD'*:

$rel\text{-}generat\ A\ Out\ R\ x\ y$
 $\implies (is\text{-}Pure\ x \longleftrightarrow is\text{-}Pure\ y) \wedge$
 $(is\text{-}Pure\ x \longrightarrow is\text{-}Pure\ y \longrightarrow A\ (result\ x)\ (result\ y)) \wedge$
 $(\neg is\text{-}Pure\ x \longrightarrow \neg is\text{-}Pure\ y \longrightarrow Out\ (output\ x)\ (output\ y) \wedge R\ (continuation\ x)\ (continuation\ y))$
 <proof>

lemma *rel-generatD*:

assumes *rel-generat A Out R x y*
shows *rel-generat-is-PureD*: $is\text{-}Pure\ x \longleftrightarrow is\text{-}Pure\ y$
and *rel-generat-resultD*: $is\text{-}Pure\ x \vee is\text{-}Pure\ y \implies A\ (result\ x)\ (result\ y)$
and *rel-generat-outputD*: $\neg is\text{-}Pure\ x \vee \neg is\text{-}Pure\ y \implies Out\ (output\ x)\ (output\ y)$
and *rel-generat-continuationD*: $\neg is\text{-}Pure\ x \vee \neg is\text{-}Pure\ y \implies R\ (continuation\ x)\ (continuation\ y)$
 <proof>

lemma *rel-generat-mono*:

$\llbracket rel\text{-}generat\ A\ B\ C\ x\ y; \bigwedge x\ y. A\ x\ y \implies A'\ x\ y; \bigwedge x\ y. B\ x\ y \implies B'\ x\ y; \bigwedge x\ y. C\ x\ y \implies C'\ x\ y \rrbracket$
 $\implies rel\text{-}generat\ A'\ B'\ C'\ x\ y$
 <proof>

lemma *rel-generat-mono' [mono]*:

$\llbracket \bigwedge x\ y. A\ x\ y \longrightarrow A'\ x\ y; \bigwedge x\ y. B\ x\ y \longrightarrow B'\ x\ y; \bigwedge x\ y. C\ x\ y \longrightarrow C'\ x\ y \rrbracket$
 $\implies rel\text{-}generat\ A\ B\ C\ x\ y \longrightarrow rel\text{-}generat\ A'\ B'\ C'\ x\ y$
 <proof>

lemma *rel-generat-same*:

$rel\text{-}generat\ A\ B\ C\ r\ r \longleftrightarrow$
 $(\forall x \in generat\text{-}pures\ r. A\ x\ x) \wedge$
 $(\forall out \in generat\text{-}outs\ r. B\ out\ out) \wedge$
 $(\forall c \in generat\text{-}conts\ r. C\ c\ c)$
 <proof>

lemma *rel-generat-reflI*:

$\llbracket \bigwedge y. y \in generat\text{-}pures\ x \implies A\ y\ y;$
 $\bigwedge out. out \in generat\text{-}outs\ x \implies B\ out\ out;$
 $\bigwedge cont. cont \in generat\text{-}conts\ x \implies C\ cont\ cont \rrbracket$
 $\implies rel\text{-}generat\ A\ B\ C\ x\ x$
 <proof>

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

lemma *transp-rel-generatI*:

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

<proof>

lemma *rel-generat-inf*:

$\text{inf } (\text{rel-generat } A \ B \ C) \ (\text{rel-generat } A' \ B' \ C') = \text{rel-generat } (\text{inf } A \ A') \ (\text{inf } B \ B') \ (\text{inf } C \ C')$

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

<proof>

lemma *rel-generat-Pure1*: $\text{rel-generat } A \ B \ C \ (\text{Pure } x) = (\lambda r. \exists y. r = \text{Pure } y \wedge A \ x \ y)$

<proof>

lemma *rel-generat-IO1*: $\text{rel-generat } A \ B \ C \ (\text{IO } \text{out } c) = (\lambda r. \exists \text{out}' \ c'. r = \text{IO } \text{out}' \ c' \wedge B \ \text{out } \text{out}' \wedge C \ c \ c')$

<proof>

lemma *not-is-Pure-conv*: $\neg \text{is-Pure } r \iff (\exists \text{out } c. r = \text{IO } \text{out } c)$

<proof>

lemma *finite-generat-outs [simp]*: $\text{finite } (\text{generat-outs } \text{generat})$

<proof>

lemma *countable-generat-outs [simp]*: $\text{countable } (\text{generat-outs } \text{generat})$

<proof>

lemma *case-map-generat*:

$\text{case-generat } \text{pure } \text{io } (\text{map-generat } a \ b \ d \ r) =$

$\text{case-generat } (\text{pure } \circ a) \ (\lambda \text{out}. \text{io } (b \ \text{out}) \circ d) \ r$

<proof>

lemma *continuation-in-generat-contr*:

$\neg \text{is-Pure } r \implies \text{continuation } r \in \text{generat-contrs}$

<proof>

fun *dest-IO* :: $('a, 'out, 'c) \text{ generat} \Rightarrow ('out \times 'c) \text{ option}$

where

$\text{dest-IO } (\text{Pure } -) = \text{None}$

| $\text{dest-IO } (\text{IO } \text{out } c) = \text{Some } (\text{out}, c)$

lemma *dest-IO-eq-Some-iff [simp]*: $\text{dest-IO } \text{generat} = \text{Some } (\text{out}, c) \iff \text{generat} = \text{IO } \text{out } c$

<proof>

lemma *dest-IO-eq-None-iff [simp]*: $\text{dest-IO } \text{generat} = \text{None} \iff \text{is-Pure } \text{generat}$

<proof>

lemma *dest-IO-comp-Pure [simp]*: $\text{dest-IO} \circ \text{Pure} = (\lambda -. \text{None})$

<proof>

lemma *dom-dest-IO*: $\text{dom dest-IO} = \{x. \neg \text{is-Pure } x\}$

<proof>

definition *generat-lub* :: $('a \text{ set} \Rightarrow 'b) \Rightarrow ('out \text{ set} \Rightarrow 'out') \Rightarrow ('cont \text{ set} \Rightarrow 'cont')$

$\Rightarrow ('a, 'out, 'cont) \text{ generat set} \Rightarrow ('b, 'out', 'cont') \text{ generat}$

where

$\text{generat-lub } lub1 \text{ } lub2 \text{ } lub3 \text{ } A =$
 $(\text{if } \exists x \in A. \text{is-Pure } x \text{ then } \text{Pure } (lub1 \text{ (result ' (A } \cap \{f. \text{is-Pure } f\})))$
 $\text{else } \text{IO } (lub2 \text{ (output ' (A } \cap \{f. \neg \text{is-Pure } f\}))) (lub3 \text{ (continuation ' (A } \cap \{f.$
 $\neg \text{is-Pure } f\}))))$

lemma *is-Pure-generat-lub* [*simp*]:

$\text{is-Pure } (\text{generat-lub } lub1 \text{ } lub2 \text{ } lub3 \text{ } A) \longleftrightarrow (\exists x \in A. \text{is-Pure } x)$

<proof>

lemma *result-generat-lub* [*simp*]:

$\exists x \in A. \text{is-Pure } x \implies \text{result } (\text{generat-lub } lub1 \text{ } lub2 \text{ } lub3 \text{ } A) = lub1 \text{ (result ' (A } \cap$
 $\{f. \text{is-Pure } f\})))$

<proof>

lemma *output-generat-lub*:

$\forall x \in A. \neg \text{is-Pure } x \implies \text{output } (\text{generat-lub } lub1 \text{ } lub2 \text{ } lub3 \text{ } A) = lub2 \text{ (output ' (A } \cap$
 $\{f. \neg \text{is-Pure } f\})))$

<proof>

lemma *continuation-generat-lub*:

$\forall x \in A. \neg \text{is-Pure } x \implies \text{continuation } (\text{generat-lub } lub1 \text{ } lub2 \text{ } lub3 \text{ } A) = lub3$
 $(\text{continuation ' (A } \cap \{f. \neg \text{is-Pure } f\})))$

<proof>

lemma *generat-lub-map* [*simp*]:

$\text{generat-lub } lub1 \text{ } lub2 \text{ } lub3 \text{ (map-generat } f \text{ } g \text{ } h \text{ ' } A) = \text{generat-lub } (lub1 \circ (\cdot) \text{ } f)$
 $(lub2 \circ (\cdot) \text{ } g) (lub3 \circ (\cdot) \text{ } h) \text{ } A$

<proof>

lemma *map-generat-lub* [*simp*]:

$\text{map-generat } f \text{ } g \text{ } h \text{ (generat-lub } lub1 \text{ } lub2 \text{ } lub3 \text{ } A) = \text{generat-lub } (f \circ lub1) (g \circ$
 $lub2) (h \circ lub3) \text{ } A$

<proof>

abbreviation *generat-lub'* :: $('cont \text{ set} \Rightarrow 'cont') \Rightarrow ('a, 'out, 'cont) \text{ generat set}$
 $\Rightarrow ('a, 'out, 'cont) \text{ generat}$

where $\text{generat-lub}' \equiv \text{generat-lub } (\lambda A. \text{THE } x. x \in A) (\lambda A. \text{THE } x. x \in A)$

fun *rel-witness-generat* :: $('a, 'c, 'e) \text{ generat} \times ('b, 'd, 'f) \text{ generat} \Rightarrow ('a \times 'b, 'c$

$\times 'd, 'e \times 'f)$ **generat where**
 $rel\text{-}witness\text{-}generat (Pure\ x, Pure\ y) = Pure\ (x, y)$
 $| rel\text{-}witness\text{-}generat (IO\ out\ c, IO\ out'\ c') = IO\ (out, out')\ (c, c')$

lemma *rel-witness-generat*:

assumes *rel-generat* $A\ C\ R\ x\ y$
shows *pures-rel-witness-generat*: $generat\text{-}pures\ (rel\text{-}witness\text{-}generat\ (x, y)) \subseteq \{(a, b). A\ a\ b\}$
and *outs-rel-witness-generat*: $generat\text{-}outs\ (rel\text{-}witness\text{-}generat\ (x, y)) \subseteq \{(c, d). C\ c\ d\}$
and *conts-rel-witness-generat*: $generat\text{-}conts\ (rel\text{-}witness\text{-}generat\ (x, y)) \subseteq \{(e, f). R\ e\ f\}$
and *map1-rel-witness-generat*: $map\text{-}generat\ fst\ fst\ fst\ (rel\text{-}witness\text{-}generat\ (x, y)) = x$
and *map2-rel-witness-generat*: $map\text{-}generat\ snd\ snd\ snd\ (rel\text{-}witness\text{-}generat\ (x, y)) = y$
<proof>

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

lemma *rel-witness-generat1*:

assumes *rel-generat* $A\ C\ R\ x\ y$
shows *rel-generat* $(\lambda a\ (a', b). a = a' \wedge A\ a'\ b)$ $(\lambda c\ (c', d). c = c' \wedge C\ c'\ d)$ $(\lambda r\ (r', s). r = r' \wedge R\ r'\ s)$ $x\ (rel\text{-}witness\text{-}generat\ (x, y))$
<proof>

lemma *rel-witness-generat2*:

assumes *rel-generat* $A\ C\ R\ x\ y$
shows *rel-generat* $(\lambda(a, b')\ b. b = b' \wedge A\ a\ b')$ $(\lambda(c, d')\ d. d = d' \wedge C\ c\ d')$
 $(\lambda(r, s')\ s. s = s' \wedge R\ r\ s')$ $(rel\text{-}witness\text{-}generat\ (x, y))\ y$
<proof>

end

theory *Generative-Probabilistic-Value imports*

Resumption

Generat

HOL-Types-To-Sets.Types-To-Sets

begin

hide-const (open) *Done*

4.2 Type definition

context notes *[[bnf-internals]] begin*

codatatype *(results'-gpv: 'a, outs'-gpv: 'out, 'in) gpv*

= *GPV* (*the-gpv*: ('a, 'out, 'in \Rightarrow ('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 \Rightarrow ('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*: $f = \text{GPV } g \iff \text{the-gpv } f = g$

<proof>

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

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

lemma *rel-gpv-def'*:

$\text{rel-gpv } A B \text{ gpv } \text{gpv}' \iff$

$(\exists \text{gpv}''. (\forall (x, y) \in \text{results}'\text{-gpv } \text{gpv}''. A x y) \wedge (\forall (x, y) \in \text{outs}'\text{-gpv } \text{gpv}''. B x y))$

\wedge

$\text{map-gpv } \text{fst } \text{fst } \text{gpv}'' = \text{gpv} \wedge \text{map-gpv } \text{snd } \text{snd } \text{gpv}'' = \text{gpv}'$

<proof>

definition *results'-rpv* :: ('a, 'out, 'in) *rpv* \Rightarrow 'a *set*

where *results'-rpv* *rpv* = *range* *rpv* $\gg=$ *results'-gpv*

definition *outs'-rpv* :: ('a, 'out, 'in) *rpv* \Rightarrow 'out *set*

where *outs'-rpv* *rpv* = *range* *rpv* $\gg=$ *outs'-gpv*

abbreviation *rel-rpv*

:: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('out \Rightarrow 'out' \Rightarrow bool)

\Rightarrow ('in \Rightarrow ('a, 'out, 'in) *gpv*) \Rightarrow ('in \Rightarrow ('b, 'out', 'in) *gpv*) \Rightarrow bool

where *rel-rpv* *A B* \equiv *rel-fun* (=) (*rel-gpv* *A B*)

lemma *in-results'-rpv* [*iff*]: $x \in \text{results}'\text{-rpv } \text{rpv} \iff (\exists \text{input}. x \in \text{results}'\text{-gpv } (\text{rpv } \text{input}))$

<proof>

lemma *in-outs'-rpv* [*iff*]: $\text{out} \in \text{outs}'\text{-rpv } \text{rpv} \iff (\exists \text{input}. \text{out} \in \text{outs}'\text{-gpv } (\text{rpv } \text{input}))$

<proof>

lemma *results'-GPV* [*simp*]:

$\text{results}'\text{-gpv } (\text{GPV } r) =$

$(\text{set-spmf } r \gg= \text{generat-pures}) \cup$

$((\text{set-spmf } r \gg= \text{generat-contrs}) \gg= \text{results}'\text{-rpv})$

$\langle \text{proof} \rangle$

lemma *outs'-GPV* [*simp*]:

$\text{outs}'\text{-gpv} (\text{GPV } r) =$
 $(\text{set-spmf } r \gg \text{generat-outs}) \cup$
 $((\text{set-spmf } r \gg \text{generat-contrs}) \gg \text{outs}'\text{-rpv})$
 $\langle \text{proof} \rangle$

lemma *outs'-gpv-unfold*:

$\text{outs}'\text{-gpv } r =$
 $(\text{set-spmf} (\text{the-gpv } r) \gg \text{generat-outs}) \cup$
 $((\text{set-spmf} (\text{the-gpv } r) \gg \text{generat-contrs}) \gg \text{outs}'\text{-rpv})$
 $\langle \text{proof} \rangle$

lemma *outs'-gpv-induct* [*consumes 1, case-names Out Cont, induct set: outs'-gpv*]:

assumes $x: x \in \text{outs}'\text{-gpv } \text{gpv}$
and *Out*: $\bigwedge \text{generat } \text{gpv}. \llbracket \text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv}); x \in \text{generat-outs } \text{generat} \rrbracket \implies P \text{ gpv}$
and *Cont*: $\bigwedge \text{generat } \text{gpv } c \text{ input}.$
 $\llbracket \text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv}); c \in \text{generat-contrs } \text{generat}; x \in \text{outs}'\text{-gpv} (c \text{ input}); P (c \text{ input}) \rrbracket \implies P \text{ gpv}$
shows $P \text{ gpv}$
 $\langle \text{proof} \rangle$

lemma *outs'-gpv-cases* [*consumes 1, case-names Out Cont, cases set: outs'-gpv*]:

assumes $x \in \text{outs}'\text{-gpv } \text{gpv}$
obtains (*Out*) generat **where** $\text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv})$ $x \in \text{generat-outs } \text{generat}$
 \mid (*Cont*) $\text{generat } c \text{ input}$ **where** $\text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv})$ $c \in \text{generat-contrs } \text{generat}$ $x \in \text{outs}'\text{-gpv} (c \text{ input})$
 $\langle \text{proof} \rangle$

lemma *outs'-gpvI* [*intro?*]:

shows *outs'-gpv-Out*: $\llbracket \text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv}); x \in \text{generat-outs } \text{generat} \rrbracket \implies x \in \text{outs}'\text{-gpv } \text{gpv}$
and *outs'-gpv-Cont*: $\llbracket \text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv}); c \in \text{generat-contrs } \text{generat}; x \in \text{outs}'\text{-gpv} (c \text{ input}) \rrbracket \implies x \in \text{outs}'\text{-gpv } \text{gpv}$
 $\langle \text{proof} \rangle$

lemma *results'-gpv-induct* [*consumes 1, case-names Pure Cont, induct set: results'-gpv*]:

assumes $x: x \in \text{results}'\text{-gpv } \text{gpv}$
and *Pure*: $\bigwedge \text{generat } \text{gpv}. \llbracket \text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv}); x \in \text{generat-pures } \text{generat} \rrbracket \implies P \text{ gpv}$
and *Cont*: $\bigwedge \text{generat } \text{gpv } c \text{ input}.$
 $\llbracket \text{generat} \in \text{set-spmf} (\text{the-gpv } \text{gpv}); c \in \text{generat-contrs } \text{generat}; x \in \text{results}'\text{-gpv} (c \text{ input}); P (c \text{ input}) \rrbracket \implies P \text{ gpv}$
shows $P \text{ gpv}$
 $\langle \text{proof} \rangle$

lemma *results'-gpv-cases* [consumes 1, case-names Pure Cont, cases set: results'-gpv]:

assumes $x \in \text{results}'\text{-gpv } \text{gpv}$

obtains (Pure) *generat* **where** $\text{generat} \in \text{set-spmf } (\text{the-gpv } \text{gpv})$ $x \in \text{generat-pures } \text{generat}$

| (Cont) *generat c input* **where** $\text{generat} \in \text{set-spmf } (\text{the-gpv } \text{gpv})$ $c \in \text{generat-contrs } \text{generat}$ $x \in \text{results}'\text{-gpv } (c \text{ input})$

$\langle \text{proof} \rangle$

lemma *results'-gpvI* [intro?]:

shows *results'-gpv-Pure*: $\llbracket \text{generat} \in \text{set-spmf } (\text{the-gpv } \text{gpv}); x \in \text{generat-pures } \text{generat} \rrbracket \implies x \in \text{results}'\text{-gpv } \text{gpv}$

and *results'-gpv-Cont*: $\llbracket \text{generat} \in \text{set-spmf } (\text{the-gpv } \text{gpv}); c \in \text{generat-contrs } \text{generat}; x \in \text{results}'\text{-gpv } (c \text{ input}) \rrbracket \implies x \in \text{results}'\text{-gpv } \text{gpv}$

$\langle \text{proof} \rangle$

lemma *left-unique-rel-gpv* [transfer-rule]:

$\llbracket \text{left-unique } A; \text{left-unique } B \rrbracket \implies \text{left-unique } (\text{rel-gpv } A B)$

$\langle \text{proof} \rangle$

lemma *right-unique-rel-gpv* [transfer-rule]:

$\llbracket \text{right-unique } A; \text{right-unique } B \rrbracket \implies \text{right-unique } (\text{rel-gpv } A B)$

$\langle \text{proof} \rangle$

lemma *bi-unique-rel-gpv* [transfer-rule]:

$\llbracket \text{bi-unique } A; \text{bi-unique } B \rrbracket \implies \text{bi-unique } (\text{rel-gpv } A B)$

$\langle \text{proof} \rangle$

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

$\llbracket \text{left-total } A; \text{left-total } B \rrbracket \implies \text{left-total } (\text{rel-gpv } A B)$

$\langle \text{proof} \rangle$

lemma *right-total-rel-gpv* [transfer-rule]:

$\llbracket \text{right-total } A; \text{right-total } B \rrbracket \implies \text{right-total } (\text{rel-gpv } A B)$

$\langle \text{proof} \rangle$

lemma *bi-total-rel-gpv* [transfer-rule]: $\llbracket \text{bi-total } A; \text{bi-total } B \rrbracket \implies \text{bi-total } (\text{rel-gpv } A B)$

$\langle \text{proof} \rangle$

declare *gpv.map-transfer*[transfer-rule]

lemma *if-distrib-map-gpv* [if-distrib]:

$\text{map-gpv } f g (\text{if } b \text{ then } \text{gpv } \text{else } \text{gpv}') = (\text{if } b \text{ then } \text{map-gpv } f g \text{ gpv } \text{else } \text{map-gpv } f g \text{ gpv}')$

$\langle \text{proof} \rangle$

lemma *gpv-pred-mono-strong*:

$\llbracket \text{pred-gpv } P Q x; \bigwedge a. \llbracket a \in \text{results}'\text{-gpv } x; P a \rrbracket \implies P' a; \bigwedge b. \llbracket b \in \text{outs}'\text{-gpv } x \rrbracket \implies P' b$

$x; Q b \parallel \implies Q' b \parallel \implies \text{pred-gpv } P' Q' x$
 $\langle \text{proof} \rangle$

lemma *pred-gpv-top* [*simp*]:
 $\text{pred-gpv } (\lambda-. \text{True}) (\lambda-. \text{True}) = (\lambda-. \text{True})$
 $\langle \text{proof} \rangle$

lemma *pred-gpv-conj* [*simp*]:
shows *pred-gpv-conj1*: $\bigwedge P Q R. \text{pred-gpv } (\lambda x. P x \wedge Q x) R = (\lambda x. \text{pred-gpv } P R x \wedge \text{pred-gpv } Q R x)$
and *pred-gpv-conj2*: $\bigwedge P Q R. \text{pred-gpv } P (\lambda x. Q x \wedge R x) = (\lambda x. \text{pred-gpv } P Q x \wedge \text{pred-gpv } P R x)$
 $\langle \text{proof} \rangle$

lemma *rel-gpv-restrict-relp1I* [*intro?*]:
 $\parallel \text{rel-gpv } R R' x y; \text{pred-gpv } P P' x; \text{pred-gpv } Q Q' y \parallel \implies \text{rel-gpv } (R \upharpoonright P \otimes Q) (R' \upharpoonright P' \otimes Q') x y$
 $\langle \text{proof} \rangle$

lemma *rel-gpv-restrict-relpE* [*elim?*]:
assumes $\text{rel-gpv } (R \upharpoonright P \otimes Q) (R' \upharpoonright P' \otimes Q') x y$
obtains $\text{rel-gpv } R R' x y \text{ pred-gpv } P P' x \text{ pred-gpv } Q Q' y$
 $\langle \text{proof} \rangle$

lemma *gpv-pred-map* [*simp*]: $\text{pred-gpv } P Q (\text{map-gpv } f g \text{ gpv}) = \text{pred-gpv } (P \circ f) (Q \circ g) \text{ gpv}$
 $\langle \text{proof} \rangle$

4.3 Generalised mapper and relator

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

primcorec *map-gpv'* :: $('a \Rightarrow 'b) \Rightarrow ('out \Rightarrow 'out') \Rightarrow ('ret' \Rightarrow 'ret) \Rightarrow ('a, 'out, 'ret) \text{ gpv} \Rightarrow ('b, 'out', 'ret') \text{ gpv}$

where

$\text{map-gpv}' f g h \text{ gpv} =$
 $\text{GPV } (\text{map-spmf } (\text{map-generat } f g ((\circ) (\text{map-gpv}' f g h))) (\text{map-spmf } (\text{map-generat } id id (\text{map-fun } h id)) (\text{the-gpv } \text{gpv})))$

declare *map-gpv'.sel* [*simp del*]

lemma *map-gpv'.sel* [*simp*]:
 $\text{the-gpv } (\text{map-gpv}' f g h \text{ gpv}) = \text{map-spmf } (\text{map-generat } f g (h \text{ ----> } \text{map-gpv}' f g h)) (\text{the-gpv } \text{gpv})$
 $\langle \text{proof} \rangle$

lemma *map-gpv'-GPV* [*simp*]:
 $\text{map-gpv}' f g h (\text{GPV } p) = \text{GPV } (\text{map-spmf } (\text{map-generat } f g (h \text{ ----> } \text{map-gpv}' f g h)) p)$

$\langle \text{proof} \rangle$

lemma *map-gpv'-id*: $\text{map-gpv}' \text{ id id id} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *map-gpv'-comp*: $\text{map-gpv}' f g h (\text{map-gpv}' f' g' h' \text{ gpv}) = \text{map-gpv}' (f \circ f') (g \circ g') (h' \circ h) \text{ gpv}$
 $\langle \text{proof} \rangle$

functor *gpv*: $\text{map-gpv}' \langle \text{proof} \rangle$

lemma *map-gpv-conv-map-gpv'*: $\text{map-gpv} f g = \text{map-gpv}' f g \text{ id}$
 $\langle \text{proof} \rangle$

coinductive *rel-gpv''* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('out \Rightarrow 'out' \Rightarrow \text{bool}) \Rightarrow ('ret \Rightarrow 'ret' \Rightarrow \text{bool}) \Rightarrow ('a, 'out, 'ret) \text{ gpv} \Rightarrow ('b, 'out', 'ret') \text{ gpv} \Rightarrow \text{bool}$

for $A C R$

where

$\text{rel-spmf} (\text{rel-generat } A C (R \Longrightarrow \text{rel-gpv}'' A C R)) (\text{the-gpv } \text{gpv}) (\text{the-gpv } \text{gpv}')$
 $\Longrightarrow \text{rel-gpv}'' A C R \text{ gpv } \text{gpv}'$

lemma *rel-gpv''-coinduct* [*consumes 1*, *case-names rel-gpv''*, *coinduct pred: rel-gpv''*]:

$\llbracket X \text{ gpv } \text{gpv}' ;$

$\wedge \text{gpv } \text{gpv}'. X \text{ gpv } \text{gpv}'$

$\Longrightarrow \text{rel-spmf} (\text{rel-generat } A C (R \Longrightarrow (\lambda \text{gpv } \text{gpv}'. X \text{ gpv } \text{gpv}' \vee \text{rel-gpv}'' A C R \text{ gpv } \text{gpv}'))$

$(\text{the-gpv } \text{gpv}) (\text{the-gpv } \text{gpv}') \rrbracket$

$\Longrightarrow \text{rel-gpv}'' A C R \text{ gpv } \text{gpv}'$

$\langle \text{proof} \rangle$

lemma *rel-gpv''D*:

$\text{rel-gpv}'' A C R \text{ gpv } \text{gpv}'$

$\Longrightarrow \text{rel-spmf} (\text{rel-generat } A C (R \Longrightarrow \text{rel-gpv}'' A C R)) (\text{the-gpv } \text{gpv}) (\text{the-gpv } \text{gpv}')$

$\langle \text{proof} \rangle$

lemma *rel-gpv''-GPV* [*simp*]:

$\text{rel-gpv}'' A C R (\text{GPV } p) (\text{GPV } q) \longleftrightarrow$

$\text{rel-spmf} (\text{rel-generat } A C (R \Longrightarrow \text{rel-gpv}'' A C R)) p q$

$\langle \text{proof} \rangle$

lemma *rel-gpv-conv-rel-gpv''*: $\text{rel-gpv} A C = \text{rel-gpv}'' A C (=)$

$\langle \text{proof} \rangle$

lemma *rel-gpv''-eq* :

$\text{rel-gpv}'' (=) (=) (=) = (=)$

$\langle \text{proof} \rangle$

lemma *rel-gpv''-mono*:

assumes $A \leq A' \ C \leq C' \ R' \leq R$

shows $\text{rel-gpv}'' \ A \ C \ R \leq \text{rel-gpv}'' \ A' \ C' \ R'$

<proof>

lemma *rel-gpv''-conversep*: $\text{rel-gpv}'' \ A^{-1-1} \ C^{-1-1} \ R^{-1-1} = (\text{rel-gpv}'' \ A \ C \ R)^{-1-1}$

<proof>

lemma *rel-gpv''-pos-distr*:

$\text{rel-gpv}'' \ A \ C \ R \ OO \ \text{rel-gpv}'' \ A' \ C' \ R' \leq \text{rel-gpv}'' \ (A \ OO \ A') \ (C \ OO \ C') \ (R \ OO \ R')$

<proof>

lemma *left-unique-rel-gpv''*:

$\llbracket \text{left-unique } A; \text{left-unique } C; \text{left-total } R \rrbracket \implies \text{left-unique } (\text{rel-gpv}'' \ A \ C \ R)$

<proof>

lemma *right-unique-rel-gpv''*:

$\llbracket \text{right-unique } A; \text{right-unique } C; \text{right-total } R \rrbracket \implies \text{right-unique } (\text{rel-gpv}'' \ A \ C \ R)$

<proof>

lemma *bi-unique-rel-gpv''* [*transfer-rule*]:

$\llbracket \text{bi-unique } A; \text{bi-unique } C; \text{bi-total } R \rrbracket \implies \text{bi-unique } (\text{rel-gpv}'' \ A \ C \ R)$

<proof>

lemma *rel-gpv''-map-gpv1*:

$\text{rel-gpv}'' \ A \ C \ R \ (\text{map-gpv } f \ g \ \text{gpv}) \ \text{gpv}' = \text{rel-gpv}'' \ (\lambda a. \ A \ (f \ a)) \ (\lambda c. \ C \ (g \ c)) \ R$

gpv gpv' (is ?lhs = ?rhs)

<proof>

lemma *rel-gpv''-map-gpv2*:

$\text{rel-gpv}'' \ A \ C \ R \ \text{gpv} \ (\text{map-gpv } f \ g \ \text{gpv}') = \text{rel-gpv}'' \ (\lambda a \ b. \ A \ a \ (f \ b)) \ (\lambda c \ d. \ C \ c \ (g \ d)) \ R \ \text{gpv} \ \text{gpv}'$

<proof>

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

lemma *rel-gpv''-map-gpv'* [*simp*]:

shows $\bigwedge f \ g \ h \ \text{gpv}. \ \text{NO-MATCH } id \ f \ \vee \ \text{NO-MATCH } id \ g$

$\implies \text{rel-gpv}'' \ A \ C \ R \ (\text{map-gpv}' \ f \ g \ h \ \text{gpv}) = \text{rel-gpv}'' \ (\lambda a. \ A \ (f \ a)) \ (\lambda c. \ C \ (g \ c))$

$R \ (\text{map-gpv}' \ id \ id \ h \ \text{gpv})$

and $\bigwedge f \ g \ h \ \text{gpv} \ \text{gpv}'. \ \text{NO-MATCH } id \ f \ \vee \ \text{NO-MATCH } id \ g$

$\implies \text{rel-gpv}'' \ A \ C \ R \ \text{gpv} \ (\text{map-gpv}' \ f \ g \ h \ \text{gpv}') = \text{rel-gpv}'' \ (\lambda a \ b. \ A \ a \ (f \ b)) \ (\lambda c \ d. \ C \ c \ (g \ d)) \ R \ \text{gpv} \ (\text{map-gpv}' \ id \ id \ h \ \text{gpv}')$

<proof>

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

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

lemma $rel\text{-}witness\text{-}gpv\text{-}sel$ [simp]:
 $the\text{-}gpv \ (rel\text{-}witness\text{-}gpv \ A \ C \ R \ R' \ (gpv, gpv')) =$
 $map\text{-}spmf \ (map\text{-}generat \ id \ id \ (\lambda(rpv, rpv'). (rel\text{-}witness\text{-}gpv \ A \ C \ R \ R' \circ$
 $rel\text{-}witness\text{-}fun \ R \ R' \ (rpv, rpv'))) \circ rel\text{-}witness\text{-}generat$
 $(rel\text{-}witness\text{-}spmf \ (rel\text{-}generat \ A \ C \ (rel\text{-}fun \ (R \ OO \ R') \ (rel\text{-}gpv'' \ A \ C \ (R \ OO$
 $R')))) \ (the\text{-}gpv \ gpv, the\text{-}gpv \ gpv'))$
 $\langle proof \rangle$

lemma **assumes** $rel\text{-}gpv'' \ A \ C \ (R \ OO \ R') \ gpv \ gpv'$
and R : $left\text{-}unique \ R \ right\text{-}total \ R$
and R' : $right\text{-}unique \ R' \ left\text{-}total \ R'$
shows $rel\text{-}witness\text{-}gpv1$: $rel\text{-}gpv'' \ (\lambda a \ (a', b). a = a' \wedge A \ a' \ b) \ (\lambda c \ (c', d). c = c' \wedge C \ c' \ d) \ R \ gpv \ (rel\text{-}witness\text{-}gpv \ A \ C \ R \ R' \ (gpv, gpv'))$ (**is** $?thesis1$)
and $rel\text{-}witness\text{-}gpv2$: $rel\text{-}gpv'' \ (\lambda(a, b') \ b. b = b' \wedge A \ a \ b') \ (\lambda(c, d') \ d. d = d' \wedge C \ c \ d') \ R' \ (rel\text{-}witness\text{-}gpv \ A \ C \ R \ R' \ (gpv, gpv')) \ gpv'$ (**is** $?thesis2$)
 $\langle proof \rangle$

lemma $rel\text{-}gpv''\text{-}neg\text{-}distr$:
assumes R : $left\text{-}unique \ R \ right\text{-}total \ R$
and R' : $right\text{-}unique \ R' \ left\text{-}total \ R'$
shows $rel\text{-}gpv'' \ (A \ OO \ A') \ (C \ OO \ C') \ (R \ OO \ R') \leq rel\text{-}gpv'' \ A \ C \ R \ OO \ rel\text{-}gpv'' \ A' \ C' \ R'$
 $\langle proof \rangle$

lemma $rel\text{-}gpv''\text{-}mono'$ [mono]:
assumes $\bigwedge x \ y. A \ x \ y \longrightarrow A' \ x \ y$
and $\bigwedge x \ y. C \ x \ y \longrightarrow C' \ x \ y$
and $\bigwedge x \ y. R' \ x \ y \longrightarrow R \ x \ y$
shows $rel\text{-}gpv'' \ A \ C \ R \ gpv \ gpv' \longrightarrow rel\text{-}gpv'' \ A' \ C' \ R' \ gpv \ gpv'$
 $\langle proof \rangle$

lemma $left\text{-}total\text{-}rel\text{-}gpv'$:
 $\llbracket left\text{-}total \ A; left\text{-}total \ C; left\text{-}unique \ R; right\text{-}total \ R \rrbracket \Longrightarrow left\text{-}total \ (rel\text{-}gpv'' \ A \ C \ R)$
 $\langle proof \rangle$

lemma $right\text{-}total\text{-}rel\text{-}gpv'$:
 $\llbracket right\text{-}total \ A; right\text{-}total \ C; right\text{-}unique \ R; left\text{-}total \ R \rrbracket \Longrightarrow right\text{-}total \ (rel\text{-}gpv'' \ A \ C \ R)$

<proof>

lemma *bi-total-rel-gpv'* [transfer-rule]:

$\llbracket \text{bi-total } A; \text{bi-total } C; \text{bi-unique } R; \text{bi-total } R \rrbracket \implies \text{bi-total } (\text{rel-gpv}'' A C R)$
<proof>

lemma *rel-fun-conversep-grp-grp*:

$\text{rel-fun } (\text{conversep } (\text{BNF-Def.Grp UNIV } f)) (\text{BNF-Def.Grp } B g) = \text{BNF-Def.Grp}$
 $\{x. (x \circ f) \text{ ' UNIV } \subseteq B\} (\text{map-fun } f g)$
<proof>

lemma *Quotient-gpv*:

assumes *Q1: Quotient R1 Abs1 Rep1 T1*
and *Q2: Quotient R2 Abs2 Rep2 T2*
and *Q3: Quotient R3 Abs3 Rep3 T3*
shows *Quotient (rel-gpv'' R1 R2 R3) (map-gpv' Abs1 Abs2 Rep3) (map-gpv'*
Rep1 Rep2 Abs3) (rel-gpv'' T1 T2 T3)
(is *Quotient ?R ?abs ?rep ?T)*
<proof>

lemma *the-gpv-parametric'*:

$(\text{rel-gpv}'' A C R \implies \text{rel-spmf } (\text{rel-generat } A C (R \implies \text{rel-gpv}'' A C R)))$
the-gpv the-gpv
<proof>

lemma *GPV-parametric'*:

$(\text{rel-spmf } (\text{rel-generat } A C (R \implies \text{rel-gpv}'' A C R)) \implies \text{rel-gpv}'' A C R)$
GPV GPV
<proof>

lemma *corec-gpv-parametric'*:

$((S \implies \text{rel-spmf } (\text{rel-generat } A C (R \implies \text{rel-sum } (\text{rel-gpv}'' A C R) S)))$
 $\implies S \implies \text{rel-gpv}'' A C R)$
corec-gpv corec-gpv
<proof>

lemma *map-gpv'-parametric* [transfer-rule]:

$((A \implies A') \implies (C \implies C') \implies (R' \implies R) \implies \text{rel-gpv}''$
 $A C R \implies \text{rel-gpv}'' A' C' R')$ *map-gpv' map-gpv'*
<proof>

lemma *map-gpv-parametric'*: $((A \implies A') \implies (C \implies C') \implies \text{rel-gpv}''$

$A C R \implies \text{rel-gpv}'' A' C' R)$ *map-gpv map-gpv*
<proof>

end

4.4 Simple, derived operations

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

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

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

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

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

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

lemma *Done-inject* [*simp*]: *Done* x = *Done* y \longleftrightarrow x = y
 \langle *proof* \rangle

lemma *Pause-inject* [*simp*]: *Pause* out c = *Pause* out' c' \longleftrightarrow out = out' \wedge c = c'
 \langle *proof* \rangle

lemma [*simp*]:
shows *Done-neq-Pause*: *Done* x \neq *Pause* out c
and *Pause-neq-Done*: *Pause* out c \neq *Done* x
 \langle *proof* \rangle

lemma *outs'-gpv-Done* [*simp*]: *outs'-gpv* (*Done* x) = {}
 \langle *proof* \rangle

lemma *results'-gpv-Done* [*simp*]: *results'-gpv* (*Done* x) = {x}
 \langle *proof* \rangle

lemma *pred-gpv-Done* [*simp*]: *pred-gpv* P Q (*Done* x) = P x
 \langle *proof* \rangle

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

lemma *results'-gpv-Pause* [*simp*]: *results'-gpv* (*Pause* out rpv) = *results'-rpv* rpv
 \langle *proof* \rangle

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

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 \longleftrightarrow 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 \neq Pause out c*
 and *Pause-neq-Fail*: *Pause out c \neq Fail*
 and *Fail-neq-Done*: *Fail \neq Done x*
 and *Done-neq-Fail*: *Done x \neq Fail*
 ⟨*proof*⟩

Add *unit* closure to circumvent SML value restriction

definition *Fail'* :: *unit \Rightarrow ('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]: $\text{Fail} = \text{GPV } f \longleftrightarrow f = \text{return-pmf None}$
 ⟨proof⟩

lemma *outs'-gpv-Fail* [simp]: $\text{outs}'\text{-gpv Fail} = \{\}$
 ⟨proof⟩

lemma *results'-gpv-Fail* [simp]: $\text{results}'\text{-gpv Fail} = \{\}$
 ⟨proof⟩

lemma *pred-gpv-Fail* [simp]: $\text{pred-gpv } P \ Q \ \text{Fail}$
 ⟨proof⟩

lemma *React-inject* [iff]: $\text{React } f = \text{React } f' \longleftrightarrow f = f'$
 ⟨proof⟩

lemma *React-apply* [simp]: $f \text{ input} = (\text{out}, c) \implies \text{React } f \text{ input} = \text{Pause out } c$
 ⟨proof⟩

lemma *rFail-apply* [simp]: $r\text{Fail input} = \text{Fail}$
 ⟨proof⟩

lemma [simp]:
 shows *rFail-neq-React*: $r\text{Fail} \neq \text{React } f$
 and *React-neq-rFail*: $\text{React } f \neq r\text{Fail}$
 ⟨proof⟩

lemma *rel-gpv-FailI* [simp]: $\text{rel-gpv } A \ C \ \text{Fail } \text{Fail}$
 ⟨proof⟩

lemma *rel-gpv-Done* [iff]: $\text{rel-gpv } A \ C \ (\text{Done } x) \ (\text{Done } y) \longleftrightarrow A \ x \ y$
 ⟨proof⟩

lemma *rel-gpv''-Done* [iff]: $\text{rel-gpv}'' \ A \ C \ R \ (\text{Done } x) \ (\text{Done } y) \longleftrightarrow A \ x \ y$
 ⟨proof⟩

lemma *rel-gpv-Pause* [iff]:
 $\text{rel-gpv } A \ C \ (\text{Pause out } c) \ (\text{Pause out}' \ c') \longleftrightarrow C \ \text{out} \ \text{out}' \wedge (\forall x. \text{rel-gpv } A \ C \ (c \ x) \ (c' \ x))$
 ⟨proof⟩

lemma *rel-gpv''-Pause* [iff]:
 $\text{rel-gpv}'' \ A \ C \ R \ (\text{Pause out } c) \ (\text{Pause out}' \ c') \longleftrightarrow C \ \text{out} \ \text{out}' \wedge (\forall x \ x'. R \ x \ x' \longrightarrow \text{rel-gpv}'' \ A \ C \ R \ (c \ x) \ (c' \ x'))$
 ⟨proof⟩

lemma *rel-gpv-lift-spmf* [iff]: $\text{rel-gpv } A \ C \ (\text{lift-spmf } p) \ (\text{lift-spmf } q) \longleftrightarrow \text{rel-spmf } A \ p \ q$
 ⟨proof⟩

lemma *rel-gpv''-lift-spmf* [*iff*]:
 $rel-gpv'' A C R (lift-spmf p) (lift-spmf q) \longleftrightarrow rel-spmf A p q$
 ⟨*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*]: $(A \implies rel-gpv A C) Done Done$
 ⟨*proof*⟩

lemma *Done-parametric'*: $(A \implies rel-gpv'' A C R) Done Done$
 ⟨*proof*⟩

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

lemma *Pause-parametric'*:
 $(C \implies (R \implies rel-gpv'' A C R) \implies rel-gpv'' A C R) Pause Pause$
 ⟨*proof*⟩

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

lemma *lift-spmf-parametric'*:
 $(rel-spmf A \implies rel-gpv'' A C R) lift-spmf lift-spmf$
 ⟨*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*]: $map-gpv f g (Pause x c) = Pause (g x) (map-gpv f g \circ c)$
 ⟨*proof*⟩

lemma *map-gpv'-Pause* [*simp*]: $map-gpv' f g h (Pause x c) = Pause (g x) (map-gpv' f g h \circ c \circ h)$
 ⟨*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 (λr. bind-gpv r f))))
  (the-gpv r ≫≡
   (case-generat
    (λx. map-spmf (map-generat id id ((◦) Inl)) (the-gpv (f x)))
    (λout c. return-spmf (IO out (λinput. Inr (c input))))))
```

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

ad hoc overloading *Monad-Syntax.bind* ⇒ *bind-gpv*

lemma *bind-gpv-unfold* [code]:

```
r ≫≡ f = GPV (
  do {
    generat ← the-gpv r;
    case generat of Pure x ⇒ the-gpv (f x)
    | IO out c ⇒ return-spmf (IO out (λinput. c input ≫≡ f))
  })
⟨proof⟩
```

lemma *bind-gpv-code-cong*: *f = f' ⇒ bind-gpv f g = bind-gpv f' g* ⟨proof⟩
⟨ML⟩

lemma *bind-gpv-sel*:

```
the-gpv (r ≫≡ f) =
  do {
    generat ← the-gpv r;
    case generat of Pure x ⇒ the-gpv (f x)
    | IO out c ⇒ return-spmf (IO out (λinput. 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) (λinput. bind-gpv (continuation generat
input) f))
```

}
 ⟨proof⟩

lemma *Done-bind-gpv* [simp]: $\text{Done } a \ggg f = f a$
 ⟨proof⟩

lemma *bind-gpv-Done* [simp]: $f \ggg \text{Done} = f$
 ⟨proof⟩

lemma *if-distrib-bind-gpv2* [if-distrib]:
 $\text{bind-gpv } gpv (\lambda y. \text{if } b \text{ then } f y \text{ else } g y) = (\text{if } b \text{ then } \text{bind-gpv } gpv f \text{ else } \text{bind-gpv } gpv g)$
 ⟨proof⟩

lemma *lift-spmf-bind*: $\text{lift-spmf } r \ggg f = \text{GPV } (r \ggg \text{the-gpv } \circ f)$
 ⟨proof⟩

lemma *the-gpv-bind-gpv-lift-spmf* [simp]:
 $\text{the-gpv } (\text{bind-gpv } (\text{lift-spmf } p) f) = \text{bind-spmf } p (\text{the-gpv } \circ f)$
 ⟨proof⟩

lemma *lift-spmf-bind-spmf*: $\text{lift-spmf } (p \ggg f) = \text{lift-spmf } p \ggg (\lambda x. \text{lift-spmf } (f x))$
 ⟨proof⟩

lemma *lift-bind-spmf*: $\text{lift-spmf } (\text{bind-spmf } p f) = \text{bind-gpv } (\text{lift-spmf } p) (\text{lift-spmf } \circ f)$
 ⟨proof⟩

lemma *GPV-bind*:
 $\text{GPV } f \ggg g = \text{GPV } (f \ggg (\lambda \text{generat. case generat of Pure } x \Rightarrow \text{the-gpv } (g x) \mid \text{IO out } c \Rightarrow \text{return-spmf } (\text{IO out } (\lambda \text{input. } c \text{ input } \ggg g))))$
 ⟨proof⟩

lemma *GPV-bind'*:
 $\text{GPV } f \ggg g = \text{GPV } (f \ggg (\lambda \text{generat. if is-Pure generat then the-gpv } (g (\text{result generat})) \text{ else return-spmf } (\text{IO } (\text{output generat}) (\lambda \text{input. continuation generat input } \ggg g))))$
 ⟨proof⟩

lemma *bind-gpv-assoc*:
fixes $f :: ('a, 'out, 'in) \text{gpv}$
shows $(f \ggg g) \ggg h = f \ggg (\lambda x. g x \ggg h)$
 ⟨proof⟩

lemma *map-gpv-bind-gpv*: $\text{map-gpv } f g (\text{bind-gpv } gpv h) = \text{bind-gpv } (\text{map-gpv } id g gpv) (\lambda x. \text{map-gpv } f g (h x))$
 ⟨proof⟩

lemma *map-gpv-id-bind-gpv*: $\text{map-gpv } f \text{ id } (\text{bind-gpv } \text{gpv } g) = \text{bind-gpv } \text{gpv } (\text{map-gpv } f \text{ id } \circ g)$
 ⟨proof⟩

lemma *map-gpv-conv-bind*:
 $\text{map-gpv } f (\lambda x. x) x = \text{bind-gpv } x (\lambda x. \text{Done } (f x))$
 ⟨proof⟩

lemma *bind-map-gpv*: $\text{bind-gpv } (\text{map-gpv } f \text{ id } \text{gpv}) g = \text{bind-gpv } \text{gpv } (g \circ f)$
 ⟨proof⟩

lemma *outs-bind-gpv*:
 $\text{outs}'\text{-gpv } (\text{bind-gpv } x f) = \text{outs}'\text{-gpv } x \cup (\bigcup x \in \text{results}'\text{-gpv } x. \text{outs}'\text{-gpv } (f x))$
 (is ?lhs = ?rhs)
 ⟨proof⟩

lemma *bind-gpv-Fail [simp]*: $\text{Fail} \gg= f = \text{Fail}$
 ⟨proof⟩

lemma *bind-gpv-eq-Fail*:
 $\text{bind-gpv } \text{gpv } f = \text{Fail} \iff (\forall x \in \text{set-spmf } (\text{the-gpv } \text{gpv}). \text{is-Pure } x) \wedge (\forall x \in \text{results}'\text{-gpv } \text{gpv}. f x = \text{Fail})$
 (is ?lhs = ?rhs)
 ⟨proof⟩

context includes *lifting-syntax begin*

lemma *bind-gpv-parametric [transfer-rule]*:
 $(\text{rel-gpv } A C \implies (A \implies \text{rel-gpv } B C) \implies \text{rel-gpv } B C) \text{ bind-gpv}$
 ⟨proof⟩

lemma *bind-gpv-parametric'*:
 $(\text{rel-gpv}'' A C R \implies (A \implies \text{rel-gpv}'' B C R) \implies \text{rel-gpv}'' B C R)$
 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*:
 $\llbracket \text{rel-gpv } A C \text{ gpv } \text{gpv}'; \bigwedge x y. A x y \implies \text{rel-gpv } B C (f x) (g y) \rrbracket$
 $\implies \text{rel-gpv } B C (\text{bind-gpv } \text{gpv } f) (\text{bind-gpv } \text{gpv}' g)$

<proof>

lemma *bind-gpv-cong*:

$\llbracket gpv = gpv'; \bigwedge x. x \in results'-gpv\ gpv' \implies f\ x = g\ x \rrbracket \implies bind-gpv\ gpv\ f = bind-gpv\ gpv'\ g$
<proof>

definition *bind-rpv* :: ('a, 'in, 'out) rpv \Rightarrow ('a \Rightarrow ('b, 'in, 'out) gpv) \Rightarrow ('b, 'in, 'out) rpv

where *bind-rpv* rpv f = (*linput*. *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* \equiv *bind-rpv*

lemma *bind-rpv-code-cong*: rpv = rpv' \implies *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) \circ f)

<proof>

lemma *bind-rpv-assoc*: *bind-rpv* (*bind-rpv* rpv f) g = *bind-rpv* rpv ((λ gpv. *bind-gpv* gpv g) \circ 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' \implies S OO S') \leq ((R \implies S) OO (R' \implies 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*:

$\llbracket \text{rel-spmf } A \text{ } p \text{ } q; y \in \text{set-spmf } q \rrbracket \implies \exists x \in \text{set-spmf } p. A \text{ } x \text{ } y$
<proof>

lemma *rel-gpv-lift-spmf1*: $\text{rel-gpv } A \text{ } B \text{ } (\text{lift-spmf } p) \text{ } gpv \longleftrightarrow (\exists q. gpv = \text{lift-spmf } q \wedge \text{rel-spmf } A \text{ } p \text{ } q)$
<proof>

lemma *rel-gpv-lift-spmf2*: $\text{rel-gpv } A \text{ } B \text{ } gpv \text{ } (\text{lift-spmf } q) \longleftrightarrow (\exists p. gpv = \text{lift-spmf } p \wedge \text{rel-spmf } A \text{ } p \text{ } q)$
<proof>

definition *pcr-spmf-gpv* :: $(\text{'a} \Rightarrow \text{'b} \Rightarrow \text{bool}) \Rightarrow \text{'a} \text{ } \text{spmf} \Rightarrow (\text{'b}, \text{'out}, \text{'in}) \text{ } gpv \Rightarrow \text{bool}$

where $\text{pcr-spmf-gpv } A = \text{cr-spmf-gpv } OO \text{ rel-gpv } A (=)$

lemma *pcr-cr-eq-spmf-gpv*: $\text{pcr-spmf-gpv } (=) = \text{cr-spmf-gpv}$
<proof>

lemma *left-unique-cr-spmf-gpv*: *left-unique cr-spmf-gpv*
<proof>

lemma *left-unique-pcr-spmf-gpv* [*transfer-rule*]:
 $\text{left-unique } A \implies \text{left-unique } (\text{pcr-spmf-gpv } A)$
<proof>

lemma *right-unique-cr-spmf-gpv*: *right-unique cr-spmf-gpv*
<proof>

lemma *right-unique-pcr-spmf-gpv* [*transfer-rule*]:
 $\text{right-unique } A \implies \text{right-unique } (\text{pcr-spmf-gpv } A)$
<proof>

lemma *bi-unique-cr-spmf-gpv*: *bi-unique cr-spmf-gpv*
<proof>

lemma *bi-unique-pcr-spmf-gpv* [*transfer-rule*]: $\text{bi-unique } A \implies \text{bi-unique } (\text{pcr-spmf-gpv } A)$
<proof>

lemma *left-total-cr-spmf-gpv*: *left-total cr-spmf-gpv*
<proof>

lemma *left-total-pcr-spmf-gpv* [*transfer-rule*]: $\text{left-total } A \implies \text{left-total } (\text{pcr-spmf-gpv } A)$
<proof>

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

lemma *return-spmf-gpv-transfer'*:

$((=) \implies cr\text{-}spmf\text{-}gpv) \text{ return-spmf Done}$
 $\langle proof \rangle$

lemma *return-spmf-gpv-transfer [transfer-rule]*:

$(A \implies pcr\text{-}spmf\text{-}gpv A) \text{ return-spmf Done}$
 $\langle proof \rangle$

lemma *bind-spmf-gpv-transfer'*:

$(cr\text{-}spmf\text{-}gpv \implies ((=) \implies cr\text{-}spmf\text{-}gpv) \implies cr\text{-}spmf\text{-}gpv) \text{ bind-spmf}$
bind-gpv
 $\langle proof \rangle$

lemma *bind-spmf-gpv-transfer [transfer-rule]*:

$(pcr\text{-}spmf\text{-}gpv A \implies (A \implies pcr\text{-}spmf\text{-}gpv B) \implies pcr\text{-}spmf\text{-}gpv B)$
bind-spmf bind-gpv
 $\langle proof \rangle$

lemma *lift-spmf-gpv-transfer'*:

$((=) \implies cr\text{-}spmf\text{-}gpv) (\lambda x. x) \text{ lift-spmf}$
 $\langle proof \rangle$

lemma *lift-spmf-gpv-transfer [transfer-rule]*:

$(rel\text{-}spmf A \implies pcr\text{-}spmf\text{-}gpv A) (\lambda x. x) \text{ lift-spmf}$
 $\langle proof \rangle$

lemma *fail-spmf-gpv-transfer'*: *cr-spmf-gpv (return-pmf None) Fail*

$\langle proof \rangle$

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

Fail

$\langle proof \rangle$

lemma *map-spmf-gpv-transfer'*:

$((=) \implies R \implies cr\text{-}spmf\text{-}gpv \implies cr\text{-}spmf\text{-}gpv) (\lambda f g. \text{map-spmf } f)$
map-gpv
 $\langle proof \rangle$

lemma *map-spmf-gpv-transfer [transfer-rule]*:

$((A \implies B) \implies R \implies pcr\text{-}spmf\text{-}gpv A \implies pcr\text{-}spmf\text{-}gpv B) (\lambda f g.$
map-spmf f) map-gpv
 $\langle proof \rangle$

end

end

4.7 Embedding 'a option as a monad

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

interpretation *option-to-spmf* $\langle \text{proof} \rangle$

interpretation *spmf-to-gpv* $\langle \text{proof} \rangle$

definition *cr-option-gpv* :: 'a option \Rightarrow ('a, 'out, 'in) gpv \Rightarrow bool

where *cr-option-gpv* x gpv \longleftrightarrow gpv = (lift-spmf \circ return-pmf) x

lemma *cr-option-gpv-conv-OO*:

cr-option-gpv = *cr-spmf-option*⁻¹⁻¹ OO *cr-spmf-gpv*
 $\langle \text{proof} \rangle$

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]:

((=) \implies *cr-option-gpv*) Some Done
 $\langle \text{proof} \rangle$

lemma *bind-option-gpv-transfer* [transfer-rule]:

(*cr-option-gpv* \implies ((=) \implies *cr-option-gpv*) \implies *cr-option-gpv*) Option.bind bind-gpv
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *map-option-gpv-transfer* [transfer-rule]:

((=) \implies $R \implies$ *cr-option-gpv* \implies *cr-option-gpv*) ($\lambda f g. \text{map-option } f$)
map-gpv
 $\langle \text{proof} \rangle$

end

end

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

interpretation *option-le-spmf* $\langle \text{proof} \rangle$

interpretation *spmf-to-gpv* $\langle \text{proof} \rangle$

definition *cr-option-le-gpv* :: 'a option \Rightarrow ('a, 'out, 'in) gpv \Rightarrow bool

where *cr-option-le-gpv* x gpv \longleftrightarrow gpv = (lift-spmf \circ return-pmf) $x \vee x = \text{None}$

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

lemma *return-option-le-gpv-transfer* [transfer-rule]:

((=) ==> cr-option-le-gpv) Some Done
<proof>

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

lemma fail-option-gpv-transfer [transfer-rule]:
cr-option-le-gpv None Fail
<proof>

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

end

end

4.8 Embedding resumptions

primcorec lift-resumption :: ('a, 'out, 'in) resumption ⇒ ('a, 'out, 'in) gpv

where

the-gpv (lift-resumption r) =
(case r of resumption.Done None ⇒ return-pmf None
| resumption.Done (Some x') ⇒ return-spmf (Pure x')
| resumption.Pause out c ⇒ map-spmf (map-generat id id ((◦) lift-resumption))
(return-spmf (IO out c)))

lemma the-gpv-lift-resumption:

the-gpv (lift-resumption r) =
(if is-Done r then if Option.is-none (resumption.result r) then return-pmf None
else return-spmf (Pure (the (resumption.result r))))
else return-spmf (IO (resumption.output r) (lift-resumption ◦ resume r)))
<proof>

declare lift-resumption.simps [simp del]

lemma lift-resumption-Done [code]:

lift-resumption (resumption.Done x) = (case x of None ⇒ Fail | Some x' ⇒ Done x')
<proof>

lemma lift-resumption-DONE [simp]:

lift-resumption (DONE x) = Done x
<proof>

lemma *lift-resumption-ABORT* [simp]:

lift-resumption ABORT = Fail

<proof>

lemma *lift-resumption-Pause* [simp, code]:

lift-resumption (resumption.Pause out c) = Pause out (lift-resumption o 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]:

$\bigwedge A. \text{pred-gpv } A \ C \ (\text{lift-resumption } r) = \text{pred-resumption } A \ C \ r$

<proof>

lemma *lift-resumption-bind*: *lift-resumption (r \gg f) = lift-resumption r \gg lift-resumption o f*

<proof>

4.9 Assertions

definition *assert-gpv* :: *bool* \Rightarrow (*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 \longleftrightarrow b*

and *Done-eq-assert-gpv*: *Done x = assert-gpv b \longleftrightarrow b*

and *Pause-neq-assert-gpv*: *Pause out rpv \neq assert-gpv b*

and *assert-gpv-neq-Pause*: *assert-gpv b \neq Pause out rpv*

and *assert-gpv-eq-Fail*: *assert-gpv b = Fail \longleftrightarrow \neg b*

and *Fail-eq-assert-gpv*: *Fail = assert-gpv b \longleftrightarrow \neg b*

<proof>

lemma *assert-gpv-inject* [simp]: *assert-gpv b = assert-gpv b' \longleftrightarrow b = b'*

<proof>

lemma *assert-gpv-sel* [*simp*]:

$the-gpv (assert-gpv b) = map-spmf Pure (assert-spmf b)$
 $\langle proof \rangle$

lemma *the-gpv-bind-assert* [*simp*]:

$the-gpv (bind-gpv (assert-gpv b) f) =$
 $bind-spmf (assert-spmf b) (the-gpv \circ f)$
 $\langle proof \rangle$

lemma *pred-gpv-assert* [*simp*]: $pred-gpv P Q (assert-gpv b) = (b \longrightarrow P ())$

$\langle proof \rangle$

primcorec *try-gpv* :: $('a, 'call, 'ret) gpv \Rightarrow ('a, 'call, 'ret) gpv \Rightarrow ('a, 'call, 'ret) gpv$
 $\langle TRY - ELSE \rightarrow [0,60] 59 \rangle$

where

$the-gpv (TRY gpv ELSE gpv') =$
 $map-spmf (map-generat id id (\lambda c input. case c input of Inl gpv \Rightarrow try-gpv gpv$
 $gpv' | Inr gpv' \Rightarrow gpv'))$
 $(try-spmf (map-spmf (map-generat id id (map-fun id Inl)) (the-gpv gpv))$
 $(map-spmf (map-generat id id (map-fun id Inr)) (the-gpv gpv')))$

lemma *try-gpv-sel*:

$the-gpv (TRY gpv ELSE gpv') =$
 $TRY map-spmf (map-generat id id (\lambda c input. TRY c input ELSE gpv')) (the-gpv$
 $gpv) ELSE the-gpv gpv'$
 $\langle proof \rangle$

lemma *try-gpv-Done* [*simp*]: $TRY Done x ELSE gpv' = Done x$

$\langle proof \rangle$

lemma *try-gpv-Fail* [*simp*]: $TRY Fail ELSE gpv' = gpv'$

$\langle proof \rangle$

lemma *try-gpv-Pause* [*simp*]: $TRY Pause out c ELSE gpv' = Pause out (\lambda input.$
 $TRY c input ELSE gpv')$

$\langle proof \rangle$

lemma *try-gpv-Fail2* [*simp*]: $TRY gpv ELSE Fail = gpv$

$\langle proof \rangle$

lemma *lift-try-spmf*: $lift-spmf (TRY p ELSE q) = TRY lift-spmf p ELSE lift-spmf$
 q

$\langle proof \rangle$

lemma *try-assert-gpv*: $TRY assert-gpv b ELSE gpv' = (if b then Done () else gpv')$

$\langle proof \rangle$

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

lemma *try-gpv-parametric* [*transfer-rule*]:
 $(rel\text{-}gpv\ A\ C\ ==\Rightarrow\ rel\text{-}gpv\ A\ C\ ==\Rightarrow\ rel\text{-}gpv\ A\ C)\ try\text{-}gpv\ try\text{-}gpv$
 $\langle proof \rangle$

lemma *try-gpv-parametric'*:
 $(rel\text{-}gpv''\ A\ C\ R\ ==\Rightarrow\ rel\text{-}gpv''\ A\ C\ R\ ==\Rightarrow\ rel\text{-}gpv''\ A\ C\ R)\ try\text{-}gpv\ try\text{-}gpv$
 $\langle proof \rangle$
end

lemma *map-try-gpv*: $map\text{-}gpv\ f\ g\ (TRY\ gpv\ ELSE\ gpv') = TRY\ map\text{-}gpv\ f\ g\ gpv$
 $ELSE\ map\text{-}gpv\ f\ g\ gpv'$
 $\langle proof \rangle$

lemma *map'-try-gpv*: $map\text{-}gpv'\ f\ g\ h\ (TRY\ gpv\ ELSE\ gpv') = TRY\ map\text{-}gpv'\ f\ g$
 $h\ gpv\ ELSE\ map\text{-}gpv'\ f\ g\ h\ gpv'$
 $\langle proof \rangle$

lemma *try-bind-assert-gpv*:
 $TRY\ (assert\text{-}gpv\ b\ \gg\ f)\ ELSE\ gpv = (if\ b\ then\ TRY\ (f\ ())\ ELSE\ gpv\ else\ gpv)$
 $\langle proof \rangle$

4.10 Order for $(\text{'}a, \text{'out}, \text{'in})\ gpv$

coinductive *ord-gpv* :: $(\text{'}a, \text{'out}, \text{'in})\ gpv \Rightarrow (\text{'}a, \text{'out}, \text{'in})\ gpv \Rightarrow bool$

where

$ord\text{-}spmf\ (rel\text{-}generat\ (=)\ (=)\ (rel\text{-}fun\ (=)\ ord\text{-}gpv))\ f\ g \Longrightarrow ord\text{-}gpv\ (GPV\ f)$
 $(GPV\ g)$

inductive-simps *ord-gpv-simps* [*simp*]:
 $ord\text{-}gpv\ (GPV\ f)\ (GPV\ g)$

lemma *ord-gpv-coinduct* [*consumes 1, case-names ord-gpv, coinduct pred: ord-gpv*]:
assumes $X\ f\ g$
and step: $\bigwedge f\ g. X\ f\ g \Longrightarrow ord\text{-}spmf\ (rel\text{-}generat\ (=)\ (=)\ (rel\text{-}fun\ (=)\ X))\ (the\text{-}gpv\ f)$
 $(the\text{-}gpv\ g)$
shows $ord\text{-}gpv\ f\ g$
 $\langle proof \rangle$

lemma *ord-gpv-the-gpvD*:
 $ord\text{-}gpv\ f\ g \Longrightarrow ord\text{-}spmf\ (rel\text{-}generat\ (=)\ (=)\ (rel\text{-}fun\ (=)\ ord\text{-}gpv))\ (the\text{-}gpv\ f)$
 $(the\text{-}gpv\ g)$
 $\langle proof \rangle$

lemma *reflp-equality*: $reflp\ (=)$
 $\langle proof \rangle$

lemma *ord-gpv-reflI* [*simp*]: $ord\text{-}gpv\ f\ f$
 $\langle proof \rangle$

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*] $\implies f = g$
 <proof>

lemma *RFail-least [simp]*: *ord-gpv Fail f*
 <proof>

4.11 Bounds on interaction

context

fixes *consider* :: 'out \Rightarrow bool

notes *monotone-SUP*[*partial-function-mono*] [[*function-internals*]]

begin

<ML>

partial-function (*lfp-strong*) *interaction-bound* :: ('a, 'out, 'in) *gpv* \Rightarrow *enat*

where

interaction-bound gpv =

(*SUP generat* \in *set-spmf (the-gpv gpv). case generat of Pure - \Rightarrow 0*

| *IO out c \Rightarrow 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 (\leq)) P*;

P (λ -. 0);

\wedge *interaction-bound'*.

[*P interaction-bound'*;

\wedge *gpv. interaction-bound' gpv \leq interaction-bound gpv*;

\wedge *gpv. interaction-bound' gpv \leq (SUP generat* \in *set-spmf (the-gpv gpv). case*

generat of Pure - \Rightarrow 0

| *IO out c \Rightarrow if consider out then eSuc (SUP input. interaction-bound' (c*

input)) else (SUP input. interaction-bound' (c input)))

]]

$\implies P$ (λ *gpv. [] generat* \in *set-spmf (the-gpv gpv). case generat of Pure x \Rightarrow 0*

$\lfloor IO \text{ out } c \Rightarrow \text{if consider out then } eSuc (\bigsqcup \text{input. interaction-bound}' (c \text{ input})) \text{ else } (\bigsqcup \text{input. interaction-bound}' (c \text{ input})) \rfloor$
 $\Rightarrow P \text{ interaction-bound}$
 <proof>

lemma *interaction-bound-IO*:

$IO \text{ out } c \in \text{set-spmf } (the\text{-gpv } gpv)$
 $\Rightarrow (\text{if consider out then } eSuc (\text{interaction-bound } (c \text{ input})) \text{ else } \text{interaction-bound } (c \text{ input})) \leq \text{interaction-bound } gpv$
 <proof>

lemma *interaction-bound-IO-consider*:

$\llbracket IO \text{ out } c \in \text{set-spmf } (the\text{-gpv } gpv); \text{consider out} \rrbracket$
 $\Rightarrow eSuc (\text{interaction-bound } (c \text{ input})) \leq \text{interaction-bound } gpv$
 <proof>

lemma *interaction-bound-IO-ignore*:

$\llbracket IO \text{ out } c \in \text{set-spmf } (the\text{-gpv } gpv); \neg \text{consider out} \rrbracket$
 $\Rightarrow \text{interaction-bound } (c \text{ input}) \leq \text{interaction-bound } gpv$
 <proof>

lemma *interaction-bound-Done [simp]*: $\text{interaction-bound } (Done \ x) = 0$
 <proof>

lemma *interaction-bound-Fail [simp]*: $\text{interaction-bound } Fail = 0$
 <proof>

lemma *interaction-bound-Pause [simp]*:

$\text{interaction-bound } (Pause \ \text{out } c) =$
 $(\text{if consider out then } eSuc (SUP \ \text{input. interaction-bound } (c \ \text{input})) \text{ else } (SUP \ \text{input. interaction-bound } (c \ \text{input})))$
 <proof>

lemma *interaction-bound-lift-spmf [simp]*: $\text{interaction-bound } (lift\text{-spmf } p) = 0$
 <proof>

lemma *interaction-bound-assert-gpv [simp]*: $\text{interaction-bound } (assert\text{-gpv } b) = 0$
 <proof>

lemma *interaction-bound-bind-step*:

assumes *IH*: $\bigwedge p. \text{interaction-bound}' (p \ggg f) \leq \text{interaction-bound } p + (\bigsqcup x \in \text{results}'\text{-gpv } p. \text{interaction-bound}' (f \ x))$

and *unfold*: $\bigwedge gpv. \text{interaction-bound}' \ gpv \leq (\bigsqcup \text{generat} \in \text{set-spmf } (the\text{-gpv } gpv). \text{case generat of Pure } x \Rightarrow 0$

$\lfloor IO \text{ out } c \Rightarrow \text{if consider out then } eSuc (\bigsqcup \text{input. interaction-bound}' (c \ \text{input})) \text{ else } \bigsqcup \text{input. interaction-bound}' (c \ \text{input}) \rfloor$

shows $(\bigsqcup \text{generat} \in \text{set-spmf } (the\text{-gpv } (p \ggg f)).$

$\text{case generat of Pure } x \Rightarrow 0$

$\lfloor IO \text{ out } c \Rightarrow$

$$\begin{aligned} & \text{if consider out then } e\text{Suc } (\sqcup \text{input. interaction-bound}' (c \text{ input})) \\ & \text{else } \sqcup \text{input. interaction-bound}' (c \text{ input}) \\ \leq & \text{ interaction-bound } p + \\ & (\sqcup x \in \text{results}'\text{-gpv } p. \\ & \quad \sqcup \text{generat} \in \text{set-spmf } (\text{the-gpv } (f x)). \\ & \quad \text{case generat of Pure } x \Rightarrow 0 \\ & \quad | \text{IO out } c \Rightarrow \\ & \quad \quad \text{if consider out then } e\text{Suc } (\sqcup \text{input. interaction-bound}' (c \text{ input})) \\ & \quad \quad \text{else } \sqcup \text{input. interaction-bound}' (c \text{ input})) \\ & (\text{is } (\text{SUP generat}' \in ?\text{bind. } ?g \text{ generat}') \leq ?p + ?f) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *interaction-bound-bind*:

defines $ib1 \equiv \text{interaction-bound}$

shows $\text{interaction-bound } (p \ggg f) \leq ib1 \ p + (\text{SUP } x \in \text{results}'\text{-gpv } p. \text{interaction-bound } (f x))$

$\langle \text{proof} \rangle$

lemma *interaction-bound-bind-lift-spmf* [simp]:

$\text{interaction-bound } (\text{lift-spmf } p \ggg f) = (\text{SUP } x \in \text{set-spmf } p. \text{interaction-bound } (f x))$

$\langle \text{proof} \rangle$

end

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

assumes *surj h*

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

$\langle \text{proof} \rangle$

abbreviation *interaction-any-bound* :: $(\text{'a}, \text{'out}, \text{'in}) \text{ gpv} \Rightarrow \text{enat}$

where $\text{interaction-any-bound} \equiv \text{interaction-bound } (\lambda\text{-}. \text{True})$

lemma *interaction-any-bound-coinduct* [consumes 1, case-names *interaction-bound*]:

assumes $X: X \text{ gpv } n$

and $*$: $\bigwedge \text{gpv } n \text{ out } c \text{ input. } \llbracket X \text{ gpv } n; \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \rrbracket$

$\implies \exists n'. (X (c \text{ input}) n' \vee \text{interaction-any-bound } (c \text{ input}) \leq n') \wedge e\text{Suc } n' \leq$

n

shows $\text{interaction-any-bound } \text{gpv} \leq n$

$\langle \text{proof} \rangle$

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

lemma *interaction-bound-parametric'*:

assumes [transfer-rule]: *bi-total R*

shows $((C \text{ ===== } (=)) \implies \text{rel-gpv'' } A \ C \ R \implies (=)) \text{ interaction-bound } \text{interaction-bound}$

$\langle \text{proof} \rangle$

lemma *interaction-bound-parametric* [transfer-rule]:
 $((C \text{ ===> } (=)) \text{ ===> } \text{rel-gpv } A \ C \text{ ===> } (=))$ *interaction-bound interaction-bound*
 <proof>
end

There is no nice *interaction-bound* equation for (\gg), 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 \Rightarrow bool) \Rightarrow ('a, 'out, 'in) *gpv* \Rightarrow *enat* \Rightarrow bool
for *consider gpv n where*
interaction-bounded-by: $\llbracket \text{interaction-bound consider gpv } \leq n \rrbracket \Longrightarrow \text{interaction-bounded-by consider gpv } n$

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]:
 $((C \text{ ===> } (=)) \text{ ===> } \text{rel-gpv } A \ C \text{ ===> } (=) \text{ ===> } (=))$ *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 $((C \text{ ===> } (=)) \text{ ===> } \text{rel-gpv}'' \ A \ C \ R \text{ ===> } (=) \text{ ===> } (=))$
interaction-bounded-by interaction-bounded-by
 <proof>
end

lemma *interaction-bounded-by-mono*:
 $\llbracket \text{interaction-bounded-by consider gpv } n; n \leq m \rrbracket \Longrightarrow \text{interaction-bounded-by consider gpv } m$
 <proof>

lemma *interaction-bounded-by-contD*:
 $\llbracket \text{interaction-bounded-by consider gpv } n; IO \ \text{out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{consider out} \rrbracket$
 $\Longrightarrow n > 0 \wedge \text{interaction-bounded-by consider } (c \ \text{input}) \ (n - 1)$
 <proof>

lemma *interaction-bounded-by-contD-ignore*:
 $\llbracket \text{interaction-bounded-by consider gpv } n; IO \ \text{out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \rrbracket$
 $\Longrightarrow \text{interaction-bounded-by consider } (c \ \text{input}) \ n$
 <proof>

lemma *interaction-bounded-byI-epred*:

assumes $\bigwedge \text{out } c. \llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{consider out} \rrbracket \implies n \neq 0$
 $\wedge (\forall \text{input}. \text{interaction-bounded-by consider } (c \text{ input}) (n - 1))$
and $\bigwedge \text{out } c \text{ input}. \llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \neg \text{consider out} \rrbracket \implies$
 $\text{interaction-bounded-by consider } (c \text{ input}) n$
shows $\text{interaction-bounded-by consider gpv } n$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-IO:*

$\llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{interaction-bounded-by consider gpv } n; \text{consider out} \rrbracket$
 $\implies n \neq 0 \wedge \text{interaction-bounded-by consider } (c \text{ input}) (n - 1)$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-0: interaction-bounded-by consider gpv 0 \longleftrightarrow interaction-bound consider gpv = 0*
 $\langle \text{proof} \rangle$

abbreviation *interaction-bounded-by'* :: $(\text{'out} \Rightarrow \text{bool}) \Rightarrow (\text{'a}, \text{'out}, \text{'in}) \text{gpv} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where $\text{interaction-bounded-by' consider gpv } n \equiv \text{interaction-bounded-by consider gpv } (\text{enat } n)$

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 =*

$((\text{match conclusion in interaction-bounded-by - - -} \Rightarrow \text{fail} \mid - \Rightarrow \langle \text{solves } \langle \text{clarsimp simp add: simp} \rangle \rangle) \mid \text{rule add interaction-bound})$

method *interaction-bound-rec uses add simp =*

$(\text{interaction-bound-step add: add simp: simp; } (\text{interaction-bound-rec add: add simp: simp})?)$

method *interaction-bound uses add simp =*

$(\text{interaction-bound-start, interaction-bound-rec add: add simp: simp})$

lemma *interaction-bounded-by-Done [simp]: interaction-bounded-by consider (Done x) n*
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-DoneI [interaction-bound]:*

$\text{interaction-bounded-by consider (Done x) 0}$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-Fail [simp]: interaction-bounded-by consider Fail n*
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-FailI [interaction-bound]: interaction-bounded-by consider Fail 0*

$\langle \text{proof} \rangle$

lemma *interaction-bounded-by-lift-spmf* [simp]: *interaction-bounded-by consider (lift-spmf p) n*
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-lift-spmfI* [interaction-bound]:
interaction-bounded-by consider (lift-spmf p) 0
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-assert-gpv* [simp]: *interaction-bounded-by consider (assert-gpv b) n*
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-assert-gpvI* [interaction-bound]:
interaction-bounded-by consider (assert-gpv b) 0
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-Pause* [simp]:
interaction-bounded-by consider (Pause out c) n \longleftrightarrow
(if consider out then $0 < n \wedge (\forall \text{input. interaction-bounded-by consider (c input) (n - 1))$ else $(\forall \text{input. interaction-bounded-by consider (c input) n)$)
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-PauseI* [interaction-bound]:
($\bigwedge \text{input. interaction-bounded-by consider (c input) (n input)$)
 \implies interaction-bounded-by consider (Pause out c) (if consider out then $1 + (\text{SUP input. } n \text{ input})$ else $(\text{SUP input. } n \text{ input})$)
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-bindI* [interaction-bound]:
 $\llbracket \text{interaction-bounded-by consider } \text{gpv } n; \bigwedge x. x \in \text{results}'\text{-gpv } \text{gpv} \implies \text{interaction-bounded-by consider (f x) (m x)} \rrbracket$
 \implies interaction-bounded-by consider (gpv \ggg f) (n + (SUP $x \in \text{results}'\text{-gpv } \text{gpv. } m x$))
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-bind-PauseI* [interaction-bound]:
($\bigwedge \text{input. interaction-bounded-by consider (c input } \ggg \text{ f) (n input)$)
 \implies interaction-bounded-by consider (Pause out c \ggg f) (if consider out then $\text{SUP input. } n \text{ input} + 1$ else $\text{SUP input. } n \text{ input}$)
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-bind-lift-spmf* [simp]:
interaction-bounded-by consider (lift-spmf p \ggg f) n \longleftrightarrow ($\forall x \in \text{set-spmf } p. \text{interaction-bounded-by consider (f x) n}$)
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-bind-lift-spmfI* [interaction-bound]:

$(\bigwedge x. x \in \text{set-spmf } p \implies \text{interaction-bounded-by consider } (f x) (n x))$
 $\implies \text{interaction-bounded-by consider } (\text{lift-spmf } p \ggg f) (\text{SUP } x \in \text{set-spmf } p. n x)$
 <proof>

lemma *interaction-bounded-by-bind-DoneI* [*interaction-bound*]:
 $\text{interaction-bounded-by consider } (f x) n \implies \text{interaction-bounded-by consider } (\text{Done } x \ggg f) n$
 <proof>

lemma *interaction-bounded-by-if* [*interaction-bound*]:
 $\llbracket b \implies \text{interaction-bounded-by consider } \text{gpv1 } n; \neg b \implies \text{interaction-bounded-by consider } \text{gpv2 } m \rrbracket$
 $\implies \text{interaction-bounded-by consider } (\text{if } b \text{ then } \text{gpv1} \text{ else } \text{gpv2}) (\text{if } b \text{ then } n \text{ else } m)$
 <proof>

lemma *interaction-bounded-by-case-bool* [*interaction-bound*]:
 $\llbracket b \implies \text{interaction-bounded-by consider } t \text{ bt}; \neg b \implies \text{interaction-bounded-by consider } f \text{ bf} \rrbracket$
 $\implies \text{interaction-bounded-by consider } (\text{case-bool } t \text{ f } b) (\text{if } b \text{ then } \text{bt} \text{ else } \text{bf})$
 <proof>

lemma *interaction-bounded-by-case-sum* [*interaction-bound*]:
 $\llbracket \bigwedge y. x = \text{Inl } y \implies \text{interaction-bounded-by consider } (l y) (bl y);$
 $\bigwedge y. x = \text{Inr } y \implies \text{interaction-bounded-by consider } (r y) (br y) \rrbracket$
 $\implies \text{interaction-bounded-by consider } (\text{case-sum } l \text{ r } x) (\text{case-sum } bl \text{ br } x)$
 <proof>

lemma *interaction-bounded-by-case-prod* [*interaction-bound*]:
 $(\bigwedge a \text{ b. } x = (a, b) \implies \text{interaction-bounded-by consider } (f a \text{ b}) (n a \text{ b}))$
 $\implies \text{interaction-bounded-by consider } (\text{case-prod } f \text{ x}) (\text{case-prod } n \text{ x})$
 <proof>

lemma *interaction-bounded-by-let* [*interaction-bound*]: — This rule unfolds let's
 $\text{interaction-bounded-by consider } (f t) m \implies \text{interaction-bounded-by consider } (\text{Let } t \text{ f}) m$
 <proof>

lemma *interaction-bounded-by-map-gpv-id* [*interaction-bound*]:
assumes [*interaction-bound*]: $\text{interaction-bounded-by } P \text{ gpv } n$
shows $\text{interaction-bounded-by } P (\text{map-gpv } f \text{ id } \text{gpv}) n$
 <proof>

abbreviation *interaction-any-bounded-by* :: $('a, 'out, 'in) \text{ gpv} \Rightarrow \text{enat} \Rightarrow \text{bool}$
where $\text{interaction-any-bounded-by} \equiv \text{interaction-bounded-by } (\lambda-. \text{True})$

lemma *interaction-any-bounded-by-map-gpv'*:
assumes $\text{interaction-any-bounded-by } \text{gpv } n$
and $\text{surj } h$

shows *interaction-any-bounded-by* (*map-gpv' f g h gpv*) *n*
 ⟨*proof*⟩

4.12 Typing

4.12.1 Interface between gpvs and rpvs / callees

lemma *is-empty-parametric* [*transfer-rule*]: *rel-fun* (*rel-set A*) (=) *Set.is-empty*
Set.is-empty
 ⟨*proof*⟩

typedef (*'call, 'ret*) *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);* $\bigwedge x y. C x y \implies rel-set R (responses-I I1$
x) (responses-I I2 y)]]

$\implies rel-I C R I1 I2$

⟨*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*: $\text{responses-I } \mathcal{I} \text{ out} = \{\}$ \longleftrightarrow $\text{out} \notin \text{outs-I } \mathcal{I}$
including *I.lifting* $\langle \text{proof} \rangle$

lemma *in-outs-I-iff-responses-I*: $\text{out} \in \text{outs-I } \mathcal{I} \longleftrightarrow \text{responses-I } \mathcal{I} \text{ out} \neq \{\}$
 $\langle \text{proof} \rangle$

lift-definition *I-full* :: (*'call*, *'ret*) \mathcal{I} **is** $\lambda\cdot$. *UNIV* $\langle \text{proof} \rangle$

lemma *I-full-sel* [*simp*]:
shows *outs-I-full*: $\text{outs-I } \mathcal{I}\text{-full} = \text{UNIV}$
and *responses-I-full*: $\text{responses-I } \mathcal{I}\text{-full } x = \text{UNIV}$
 $\langle \text{proof} \rangle$

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

lemma *outs-I-parametric* [*transfer-rule*]: $(\text{rel-I } C R \implies \text{rel-set } C) \text{ outs-I}$
 outs-I
 $\langle \text{proof} \rangle$

lemma *responses-I-parametric* [*transfer-rule*]:
 $(\text{rel-I } C R \implies C \implies \text{rel-set } R) \text{ responses-I responses-I}$
 $\langle \text{proof} \rangle$

end

definition *I-trivial* :: (*'out*, *'in*) $\mathcal{I} \Rightarrow \text{bool}$
where *I-trivial* $\mathcal{I} \longleftrightarrow \text{outs-I } \mathcal{I} = \text{UNIV}$

lemma *I-trivialI* [*intro?*]: $(\bigwedge x. x \in \text{outs-I } \mathcal{I}) \implies \text{I-trivial } \mathcal{I}$
 $\langle \text{proof} \rangle$

lemma *I-trivialD*: $\text{I-trivial } \mathcal{I} \implies \text{outs-I } \mathcal{I} = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *I-trivial-I-full* [*simp*]: *I-trivial* $\mathcal{I}\text{-full}$
 $\langle \text{proof} \rangle$

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

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

lift-definition *I-uniform* :: *'out set* \Rightarrow *'in set* \Rightarrow (*'out*, *'in*) \mathcal{I} **is** $\lambda A B x. \text{if } x \in A \text{ then } B \text{ else } \{\}$ $\langle \text{proof} \rangle$

lemma *outs-I-uniform* [*simp*]: $\text{outs-I } (\text{I-uniform } A B) = (\text{if } B = \{\} \text{ then } \{\} \text{ else } A)$
 $\langle \text{proof} \rangle$

lemma *responses- \mathcal{I} -uniform* [simp]: *responses- \mathcal{I} (\mathcal{I} -uniform $A B$) $x = (if\ x \in A$*
then B else $\{\}$)

\langle proof \rangle

lemma *\mathcal{I} -uniform-UNIV* [simp]: *\mathcal{I} -uniform UNIV UNIV = \mathcal{I} -full*

\langle proof \rangle

lift-definition *map- \mathcal{I}* :: *('out' \Rightarrow 'out') \Rightarrow ('in' \Rightarrow 'in') \Rightarrow ('out, 'in) $\mathcal{I} \Rightarrow$ ('out',*
'in') \mathcal{I}

is *$\lambda f\ g\ resp\ x.\ g\ \text{' resp } (f\ x)$* *\langle proof \rangle*

lemma *outs- \mathcal{I} -map- \mathcal{I}* [simp]:

outs- \mathcal{I} (map- \mathcal{I} $f\ g\ \mathcal{I}) = f\ \text{' outs- \mathcal{I} $\mathcal{I}$$

\langle proof \rangle

lemma *responses- \mathcal{I} -map- \mathcal{I}* [simp]:

responses- \mathcal{I} (map- \mathcal{I} $f\ g\ \mathcal{I})\ x = g\ \text{' responses- \mathcal{I} \mathcal{I} (f x)$

\langle proof \rangle

lemma *map- \mathcal{I} - \mathcal{I} -uniform* [simp]:

map- \mathcal{I} $f\ g\ (\mathcal{I}$ -uniform $A B) = \mathcal{I}$ -uniform (f -' A) (g -' B)

\langle proof \rangle

lemma *map- \mathcal{I} -id* [simp]: *map- \mathcal{I} id id $\mathcal{I} = \mathcal{I}$*

\langle proof \rangle

lemma *map- \mathcal{I} -id0*: *map- \mathcal{I} id id = id*

\langle proof \rangle

lemma *map- \mathcal{I} -comp* [simp]: *map- \mathcal{I} $f\ g\ (\text{map- \mathcal{I} } f' g' \mathcal{I}) = \text{map- \mathcal{I} } (f' \circ f)\ (g \circ g')$*
 \mathcal{I}

\langle proof \rangle

lemma *map- \mathcal{I} -cong*: *map- \mathcal{I} $f\ g\ \mathcal{I} = \text{map- \mathcal{I} } f' g' \mathcal{I}'$*

if *$\mathcal{I} = \mathcal{I}'$ and $f: f = f'$ and $\bigwedge x\ y.\ \llbracket x \in \text{outs- \mathcal{I} } \mathcal{I}'; y \in \text{responses- \mathcal{I} } \mathcal{I}'\ x \rrbracket \implies$*
 $g\ y = g'\ y$

\langle proof \rangle

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

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

end

functor *map- \mathcal{I}* *\langle proof \rangle*

lemma *\mathcal{I} -eqI*: *$\llbracket \text{outs- \mathcal{I} } \mathcal{I} = \text{outs- \mathcal{I} } \mathcal{I}'; \bigwedge x.\ x \in \text{outs- \mathcal{I} } \mathcal{I}' \implies \text{responses- \mathcal{I} } \mathcal{I}\ x =$*
 $\text{responses- \mathcal{I} } \mathcal{I}'\ x \rrbracket \implies \mathcal{I} = \mathcal{I}'$

including *\mathcal{I} .lifting* *\langle proof \rangle*

instantiation *\mathcal{I}* :: *(type, type) order begin*

definition $less\text{-}eq\text{-}\mathcal{I} :: ('a, 'b) \mathcal{I} \Rightarrow ('a, 'b) \mathcal{I} \Rightarrow bool$
where $le\text{-}\mathcal{I}\text{-}def: less\text{-}eq\text{-}\mathcal{I} \mathcal{I} \mathcal{I}' \longleftrightarrow outs\text{-}\mathcal{I} \mathcal{I} \subseteq outs\text{-}\mathcal{I} \mathcal{I}' \wedge (\forall x \in outs\text{-}\mathcal{I} \mathcal{I}. responses\text{-}\mathcal{I} \mathcal{I}' x \subseteq responses\text{-}\mathcal{I} \mathcal{I} x)$

definition $less\text{-}\mathcal{I} :: ('a, 'b) \mathcal{I} \Rightarrow ('a, 'b) \mathcal{I} \Rightarrow bool$
where $less\text{-}\mathcal{I} = mk\text{-}less (\leq)$

instance
 $\langle proof \rangle$
end

instantiation $\mathcal{I} :: (type, type) order\text{-}bot$ **begin**
definition $bot\text{-}\mathcal{I} :: ('a, 'b) \mathcal{I}$ **where** $bot\text{-}\mathcal{I} = \mathcal{I}\text{-}uniform \{\}$ *UNIV*
instance $\langle proof \rangle$
end

lemma $outs\text{-}\mathcal{I}\text{-}bot$ [*simp*]: $outs\text{-}\mathcal{I} bot = \{\}$
 $\langle proof \rangle$

lemma $responses\text{-}\mathcal{I}\text{-}bot$ [*simp*]: $responses\text{-}\mathcal{I} bot x = \{\}$
 $\langle proof \rangle$

lemma $outs\text{-}\mathcal{I}\text{-}mono$: $\mathcal{I} \leq \mathcal{I}' \Longrightarrow outs\text{-}\mathcal{I} \mathcal{I} \subseteq outs\text{-}\mathcal{I} \mathcal{I}'$
 $\langle proof \rangle$

lemma $responses\text{-}\mathcal{I}\text{-}mono$: $\llbracket \mathcal{I} \leq \mathcal{I}'; x \in outs\text{-}\mathcal{I} \mathcal{I} \rrbracket \Longrightarrow responses\text{-}\mathcal{I} \mathcal{I}' x \subseteq responses\text{-}\mathcal{I} \mathcal{I} x$
 $\langle proof \rangle$

lemma $\mathcal{I}\text{-}uniform\text{-}empty$ [*simp*]: $\mathcal{I}\text{-}uniform \{\} A = bot$
 $\langle proof \rangle$ **including** $\mathcal{I}\text{-}lifting$ $\langle proof \rangle$

lemma $\mathcal{I}\text{-}uniform\text{-}mono$:
 $\mathcal{I}\text{-}uniform A B \leq \mathcal{I}\text{-}uniform C D$ **if** $A \subseteq C D \subseteq B D = \{\} \longrightarrow B = \{\}$
 $\langle proof \rangle$

context begin

qualified inductive $resultsp\text{-}gpv :: ('out, 'in) \mathcal{I} \Rightarrow 'a \Rightarrow ('a, 'out, 'in) gpv \Rightarrow bool$
for Γx

where

$Pure: Pure x \in set\text{-}spmf (the\text{-}gpv gpv) \Longrightarrow resultsp\text{-}gpv \Gamma x gpv$

| *IO*:

$\llbracket IO out c \in set\text{-}spmf (the\text{-}gpv gpv); input \in responses\text{-}\mathcal{I} \Gamma out; resultsp\text{-}gpv \Gamma x (c input) \rrbracket$

$\Longrightarrow resultsp\text{-}gpv \Gamma x gpv$

definition $results\text{-}gpv :: ('out, 'in) \mathcal{I} \Rightarrow ('a, 'out, 'in) gpv \Rightarrow 'a set$

where $results\text{-}gpv \ \Gamma \ gpv \equiv \{x. \ results\text{-}gpv \ \Gamma \ x \ gpv\}$

lemma $results\text{-}gpv\text{-}results\text{-}gpv\text{-}eq$ [*pred-set-conv*]: $results\text{-}gpv \ \Gamma \ x \ gpv \longleftrightarrow x \in results\text{-}gpv \ \Gamma \ gpv$
(*proof*)

context begin

(*ML*)

lemmas $intros$ [*intro?*] = $results\text{-}gpv.intros[to\text{-}set]$
 and $Pure = Pure[to\text{-}set]$
 and $IO = IO[to\text{-}set]$
 and $induct$ [*consumes 1, case-names Pure IO, induct set: results-gpv*] = $results\text{-}gpv.induct[to\text{-}set]$
 and $cases$ [*consumes 1, case-names Pure IO, cases set: results-gpv*] = $results\text{-}gpv.cases[to\text{-}set]$
 and $simps = results\text{-}gpv.simps[to\text{-}set]$
end

inductive-simps $results\text{-}gpv\text{-}GPV$ [*to-set, simp*]: $results\text{-}gpv \ \Gamma \ x \ (GPV \ gpv)$

end

lemma $results\text{-}gpv\text{-}Done$ [*iff*]: $results\text{-}gpv \ \Gamma \ (Done \ x) = \{x\}$
(*proof*)

lemma $results\text{-}gpv\text{-}Fail$ [*iff*]: $results\text{-}gpv \ \Gamma \ Fail = \{\}$
(*proof*)

lemma $results\text{-}gpv\text{-}Pause$ [*simp*]:
 $results\text{-}gpv \ \Gamma \ (Pause \ out \ c) = (\bigcup input \in responses\text{-}\mathcal{I} \ \Gamma \ out. \ results\text{-}gpv \ \Gamma \ (c \ input))$
(*proof*)

lemma $results\text{-}gpv\text{-}lift\text{-}spm\text{-}f$ [*iff*]: $results\text{-}gpv \ \Gamma \ (lift\text{-}spm\text{-}f \ p) = set\text{-}spm\text{-}f \ p$
(*proof*)

lemma $results\text{-}gpv\text{-}assert\text{-}gpv$ [*simp*]: $results\text{-}gpv \ \Gamma \ (assert\text{-}gpv \ b) = (if \ b \ then \ \{\}\ else \ \{\})$
(*proof*)

lemma $results\text{-}gpv\text{-}bind\text{-}gpv$ [*simp*]:
 $results\text{-}gpv \ \Gamma \ (gpv \ggg \ f) = (\bigcup x \in results\text{-}gpv \ \Gamma \ gpv. \ results\text{-}gpv \ \Gamma \ (f \ x))$
 (**is** ?*lhs* = ?*rhs*)
(*proof*)

lemma $results\text{-}gpv\text{-}\mathcal{I}\text{-full}$: $results\text{-}gpv \ \mathcal{I}\text{-full} = results'\text{-}gpv$
(*proof*)

lemma $results'\text{-}bind\text{-}gpv$ [*simp*]:
 $results'\text{-}gpv \ (bind\text{-}gpv \ gpv \ f) = (\bigcup x \in results'\text{-}gpv \ gpv. \ results'\text{-}gpv \ (f \ x))$

<proof>

lemma *results-gpv-map-gpv-id* [simp]: *results-gpv* \mathcal{I} (*map-gpv* *f id* *gpv*) = *f* ‘ *results-gpv* \mathcal{I} *gpv*
<proof>

lemma *results-gpv-map-gpv-id'* [simp]: *results-gpv* \mathcal{I} (*map-gpv* *f* ($\lambda x. x$) *gpv*) = *f* ‘ *results-gpv* \mathcal{I} *gpv*
<proof>

lemma *pred-gpv-bind* [simp]: *pred-gpv* *P Q* (*bind-gpv* *gpv f*) = *pred-gpv* (*pred-gpv* *P Q* \circ *f*) *Q gpv*
<proof>

lemma *results'-gpv-bind-option* [simp]:
results'-gpv (*monad.bind-option* *Fail x f*) = ($\bigcup_{y \in \text{set-option } x. \text{results'-gpv } (f y)}$)
<proof>

lemma *results'-gpv-map-gpv'*:
assumes *surj h*
shows *results'-gpv* (*map-gpv'* *f g h gpv*) = *f* ‘ *results'-gpv* *gpv* (**is** ?lhs = ?rhs)
<proof>

lemma *bind-gpv-bind-option-assoc*:
bind-gpv (*monad.bind-option* *Fail x f*) *g* = *monad.bind-option* *Fail x* ($\lambda x. \text{bind-gpv } (f x) g$)
<proof>

context begin

qualified inductive *outsp-gpv* :: (*'out, 'in*) $\mathcal{I} \Rightarrow 'out \Rightarrow ('a, 'out, 'in) \text{ gpv} \Rightarrow \text{bool}$
for \mathcal{I} *x* **where**
 IO: $IO\ x\ c \in \text{set-spmf } (the\text{-gpv } gpv) \implies \text{outsp-gpv } \mathcal{I}\ x\ gpv$
 | *Cont*: $\llbracket IO\ out\ rpv \in \text{set-spmf } (the\text{-gpv } gpv); input \in \text{responses-}\mathcal{I}\ \mathcal{I}\ out; \text{outsp-gpv } \mathcal{I}\ x\ (rpv\ input) \rrbracket$
 $\implies \text{outsp-gpv } \mathcal{I}\ x\ gpv$

definition *outs-gpv* :: (*'out, 'in*) $\mathcal{I} \Rightarrow ('a, 'out, 'in) \text{ gpv} \Rightarrow 'out\ \text{set}$
where *outs-gpv* \mathcal{I} *gpv* $\equiv \{x. \text{outsp-gpv } \mathcal{I}\ x\ gpv\}$

lemma *outsp-gpv-outs-gpv-eq* [pred-set-conv]: *outsp-gpv* \mathcal{I} *x* = ($\lambda gpv. x \in \text{outs-gpv } \mathcal{I}\ gpv$)
<proof>

context begin

<ML>

lemmas *intros* [*intro?*] = *outsp-gpv.intros*[*to-set*]
and *IO* = *IO*[*to-set*]
and *Cont* = *Cont*[*to-set*]

and *induct* [*consumes 1, case-names IO Cont, induct set: outs-gpv*] = *outsp-gpv.induct[to-set]*
and *cases* [*consumes 1, case-names IO Cont, cases set: outs-gpv*] = *outsp-gpv.cases[to-set]*
and *simps* = *outsp-gpv.simps[to-set]*
end

inductive-simps *outs-gpv-GPV* [*to-set, simp*]: *outsp-gpv* \mathcal{I} *x* (*GPV gpv*)

end

lemma *outs-gpv-Done* [*iff*]: *outs-gpv* \mathcal{I} (*Done x*) = {}
 ⟨*proof*⟩

lemma *outs-gpv-Fail* [*iff*]: *outs-gpv* \mathcal{I} *Fail* = {}
 ⟨*proof*⟩

lemma *outs-gpv-Pause* [*simp*]:
outs-gpv \mathcal{I} (*Pause out c*) = *insert out* (\bigcup *input* ∈ *responses- \mathcal{I}* \mathcal{I} *out*. *outs-gpv* \mathcal{I} (*c input*))
 ⟨*proof*⟩

lemma *outs-gpv-lift-spmf* [*iff*]: *outs-gpv* \mathcal{I} (*lift-spmf p*) = {}
 ⟨*proof*⟩

lemma *outs-gpv-assert-gpv* [*simp*]: *outs-gpv* \mathcal{I} (*assert-gpv b*) = {}
 ⟨*proof*⟩

lemma *outs-gpv-bind-gpv* [*simp*]:
outs-gpv \mathcal{I} (*gpv* \ggg *f*) = *outs-gpv* \mathcal{I} *gpv* \cup (\bigcup *x* ∈ *results-gpv* \mathcal{I} *gpv*. *outs-gpv* \mathcal{I} (*f x*))
 (**is** ?*lhs* = ?*rhs*)
 ⟨*proof*⟩

lemma *outs-gpv- \mathcal{I} -full*: *outs-gpv* \mathcal{I} -full = *outs'-gpv*
 ⟨*proof*⟩

lemma *outs'-bind-gpv* [*simp*]:
outs'-gpv (*bind-gpv gpv f*) = *outs'-gpv gpv* \cup (\bigcup *x* ∈ *results'-gpv gpv*. *outs'-gpv* (*f x*))
 ⟨*proof*⟩

lemma *outs-gpv-map-gpv-id* [*simp*]: *outs-gpv* \mathcal{I} (*map-gpv f id gpv*) = *outs-gpv* \mathcal{I} *gpv*
 ⟨*proof*⟩

lemma *outs-gpv-map-gpv-id'* [*simp*]: *outs-gpv* \mathcal{I} (*map-gpv f* ($\lambda x. x$) *gpv*) = *outs-gpv* \mathcal{I} *gpv*
 ⟨*proof*⟩

lemma *outs'-gpv-bind-option* [*simp*]:

$outs'-gpv \text{ (monad.bind-option Fail } x f) = (\bigcup_{y \in \text{set-option } x} outs'-gpv (f y))$
 ⟨proof⟩

lemma *rel-gpv''-Grp: includes lifting-syntax shows*
 $rel-gpv'' (BNF-Def.Grp A f) (BNF-Def.Grp B g) (BNF-Def.Grp UNIV h)^{-1-1}$
 =
 $BNF-Def.Grp \{x. results-gpv (\mathcal{I}\text{-uniform } UNIV (range h)) x \subseteq A \wedge outs-gpv$
 $(\mathcal{I}\text{-uniform } UNIV (range h)) x \subseteq B\} (map-gpv' f g h)$
 (is ?lhs = ?rhs)
 ⟨proof⟩

inductive *pred-gpv' :: ('a \Rightarrow bool) \Rightarrow ('out \Rightarrow bool) \Rightarrow 'in set \Rightarrow ('a, 'out, 'in) gpv*
 \Rightarrow bool **for** $P Q X gpv$ **where**
 $pred-gpv' P Q X gpv$
if $\bigwedge x. x \in results-gpv (\mathcal{I}\text{-uniform } UNIV X) gpv \Longrightarrow P x \wedge out. out \in outs-gpv$
 $(\mathcal{I}\text{-uniform } UNIV X) gpv \Longrightarrow Q out$

lemma *pred-gpv-conv-pred-gpv'*: $pred-gpv P Q = pred-gpv' P Q UNIV$
 ⟨proof⟩

lemma *rel-gpv''-map-gpv'1*:
 $rel-gpv'' A C (BNF-Def.Grp UNIV h)^{-1-1} gpv gpv' \Longrightarrow rel-gpv'' A C (=)$
 $(map-gpv' id id h gpv) gpv'$
 ⟨proof⟩

lemma *rel-gpv''-map-gpv'2*:
 $rel-gpv'' A C (eq-on (range h)) gpv gpv' \Longrightarrow rel-gpv'' A C (BNF-Def.Grp UNIV$
 $h)^{-1-1} gpv (map-gpv' id id h gpv')$
 ⟨proof⟩

context

fixes $A :: 'a \Rightarrow 'd \Rightarrow bool$
and $C :: 'c \Rightarrow 'g \Rightarrow bool$
and $R :: 'b \Rightarrow 'e \Rightarrow bool$

begin

private lemma *f11*: $Pure x \in \text{set-spmf } (the-gpv gpv) \Longrightarrow$
 $Domainp (rel-generat A C (rel-fun R (rel-gpv'' A C R))) (Pure x) \Longrightarrow Domainp$
 $A x$

⟨proof⟩ **lemma** *f21*: $IO out c \in \text{set-spmf } (the-gpv gpv) \Longrightarrow$
 $rel-generat A C (rel-fun R (rel-gpv'' A C R)) (IO out c) ba \Longrightarrow Domainp C out$

⟨proof⟩ **lemma** *f12*:

assumes $IO out c \in \text{set-spmf } (the-gpv gpv)$

and $input \in \text{responses-}\mathcal{I} (\mathcal{I}\text{-uniform } UNIV \{x. Domainp R x\}) out$

and $x \in results-gpv (\mathcal{I}\text{-uniform } UNIV \{x. Domainp R x\}) (c input)$

and $Domainp (rel-gpv'' A C R) gpv$

shows $Domainp (rel-gpv'' A C R) (c input)$

⟨proof⟩ **lemma** *f22*:

assumes $IO out' rpv \in \text{set-spmf } (the-gpv gpv)$

and $input \in responses\mathcal{I}$ (\mathcal{I} -uniform UNIV $\{x. Domainp R x\}$) out'
and $out \in outs\text{-}gpv$ (\mathcal{I} -uniform UNIV $\{x. Domainp R x\}$) ($rpv\ input$)
and $Domainp$ ($rel\text{-}gpv'' A C R$) gpv
shows $Domainp$ ($rel\text{-}gpv'' A C R$) ($rpv\ input$)
 $\langle proof \rangle$

lemma $Domainp\text{-}rel\text{-}gpv''\text{-}le$:

$Domainp$ ($rel\text{-}gpv'' A C R$) $\leq pred\text{-}gpv'$ ($Domainp A$) ($Domainp C$) $\{x. Domainp R x\}$
 $\langle proof \rangle$

end

lemma $map\text{-}gpv'\text{-}id12$: $map\text{-}gpv' f g h\ gpv = map\text{-}gpv' id id h$ ($map\text{-}gpv f g\ gpv$)
 $\langle proof \rangle$

lemma $rel\text{-}gpv''\text{-}refl$: $\llbracket (=) \leq A; (=) \leq C; R \leq (=) \rrbracket \implies (=) \leq rel\text{-}gpv'' A C R$
 $\langle proof \rangle$

context

fixes $A A' :: 'a \Rightarrow 'b \Rightarrow bool$
and $C C' :: 'c \Rightarrow 'd \Rightarrow bool$
and $R R' :: 'e \Rightarrow 'f \Rightarrow bool$

begin

private abbreviation foo **where**

$foo \equiv (\lambda fx\ fy\ gpvx\ gpyy\ g1\ g2.$
 $\quad \forall x\ y. x \in fx$ (\mathcal{I} -uniform UNIV ($Collect$ ($Domainp R'$))) $gpvx \longrightarrow$
 $\quad \quad y \in fy$ (\mathcal{I} -uniform UNIV ($Collect$ ($Rangep R'$))) $gpyy \longrightarrow g1\ x\ y$
 $\longrightarrow g2\ x\ y)$

private lemma $f1$: $foo\ results\text{-}gpv\ results\text{-}gpv\ gpv\ gpv' A A' \implies$

$x \in set\text{-}spmf$ ($the\text{-}gpv\ gpv$) $\implies y \in set\text{-}spmf$ ($the\text{-}gpv\ gpv'$) \implies
 $a \in generat\text{-}conts\ x \implies b \in generat\text{-}conts\ y \implies R' a' \alpha \implies R' \beta b' \implies$
 $foo\ results\text{-}gpv\ results\text{-}gpv\ (a\ a')\ (b\ b')\ A\ A'$

$\langle proof \rangle$ **lemma** $f2$: $foo\ outs\text{-}gpv\ outs\text{-}gpv\ gpv\ gpv' C C' \implies$

$x \in set\text{-}spmf$ ($the\text{-}gpv\ gpv$) $\implies y \in set\text{-}spmf$ ($the\text{-}gpv\ gpv'$) \implies
 $a \in generat\text{-}conts\ x \implies b \in generat\text{-}conts\ y \implies R' a' \alpha \implies R' \beta b' \implies$
 $foo\ outs\text{-}gpv\ outs\text{-}gpv\ (a\ a')\ (b\ b')\ C\ C'$

$\langle proof \rangle$

lemma $rel\text{-}gpv''\text{-}mono\text{-}strong$:

$\llbracket rel\text{-}gpv'' A C R\ gpv\ gpv';$
 $\quad \bigwedge x\ y. \llbracket x \in results\text{-}gpv$ (\mathcal{I} -uniform UNIV $\{x. Domainp R' x\}$) $gpv; y \in$
 $results\text{-}gpv$ (\mathcal{I} -uniform UNIV $\{x. Rangep R' x\}$) $gpv'; A\ x\ y \rrbracket \implies A' x\ y;$
 $\quad \bigwedge x\ y. \llbracket x \in outs\text{-}gpv$ (\mathcal{I} -uniform UNIV $\{x. Domainp R' x\}$) $gpv; y \in outs\text{-}gpv$
 $(\mathcal{I}\text{-uniform UNIV } \{x. Rangep R' x\})\ gpv'; C\ x\ y \rrbracket \implies C' x\ y;$

$R' \leq R$]
 $\implies \text{rel-gpv}'' A' C' R' \text{ gpv } \text{gpv}'$
 <proof>

end

lemma *rel-gpv''-reft-strong*:

assumes $\bigwedge x. x \in \text{results-gpv } (\mathcal{I}\text{-uniform UNIV } \{x. \text{Domainp } R \ x\}) \text{ gpv} \implies A$
 $x \ x$
and $\bigwedge x. x \in \text{outs-gpv } (\mathcal{I}\text{-uniform UNIV } \{x. \text{Domainp } R \ x\}) \text{ gpv} \implies C \ x \ x$
and $R \leq (=)$
shows $\text{rel-gpv}'' A \ C \ R \ \text{gpv } \ \text{gpv}$
 <proof>

lemma *rel-gpv''-reft-eq-on*:

$\llbracket \bigwedge x. x \in \text{results-gpv } (\mathcal{I}\text{-uniform UNIV } X) \text{ gpv} \implies A \ x \ x; \bigwedge \text{out}. \text{out} \in \text{outs-gpv}$
 $(\mathcal{I}\text{-uniform UNIV } X) \text{ gpv} \implies B \ \text{out} \ \text{out} \rrbracket$
 $\implies \text{rel-gpv}'' A \ B \ (\text{eq-on } X) \ \text{gpv } \ \text{gpv}$
 <proof>

lemma *pred-gpv'-mono' [mono]*:

$\text{pred-gpv}' A \ C \ R \ \text{gpv} \longrightarrow \text{pred-gpv}' A' \ C' \ R \ \text{gpv}$
if $\bigwedge x. A \ x \longrightarrow A' \ x \ \bigwedge x. C \ x \longrightarrow C' \ x$
 <proof>

4.12.2 Type judgements

coinductive *WT-gpv* :: ('out, 'in) $\mathcal{I} \Rightarrow ('a, 'out, 'in) \text{ gpv} \Rightarrow \text{bool}$ ($\langle\langle(-)/\vdash_g(-)\rangle\rangle$)
 \checkmark)_[100, 0] 99

for Γ

where

$(\bigwedge \text{out } c. \text{IO out } c \in \text{set-spmf } \text{gpv} \implies \text{out} \in \text{outs-}\mathcal{I} \ \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \ \Gamma$
 $\text{out}. \Gamma \vdash_g c \ \text{input} \ \checkmark))$
 $\implies \Gamma \vdash_g \text{GPV } \text{gpv} \ \checkmark$

lemma *WT-gpv-coinduct [consumes 1, case-names WT-gpv, case-conclusion WT-gpv out cont, coinduct pred: WT-gpv]*:

assumes *: $X \ \text{gpv}$
and *step*: $\bigwedge \text{gpv out } c.$
 $\llbracket X \ \text{gpv}; \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \rrbracket$
 $\implies \text{out} \in \text{outs-}\mathcal{I} \ \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \ \Gamma \ \text{out}. X \ (c \ \text{input}) \vee \Gamma \vdash_g c \ \text{input}$
 $\checkmark)$
shows $\Gamma \vdash_g \text{gpv} \ \checkmark$
 <proof>

lemma *WT-gpv-simps*:

$\Gamma \vdash_g \text{GPV } \text{gpv} \ \checkmark \iff$
 $(\forall \text{out } c. \text{IO out } c \in \text{set-spmf } \text{gpv} \longrightarrow \text{out} \in \text{outs-}\mathcal{I} \ \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \ \Gamma$
 $\text{out}. \Gamma \vdash_g c \ \text{input} \ \checkmark))$

$\langle proof \rangle$

lemma *WT-gpvI*:

$(\bigwedge out\ c. IO\ out\ c \in set\ spmf\ (the\ gpv\ gpv) \implies out \in outs\ \mathcal{I}\ \Gamma \wedge (\forall input \in responses\ \mathcal{I}\ \Gamma\ out. \Gamma \vdash_g\ c\ input\ \checkmark))$
 $\implies \Gamma \vdash_g\ gpv\ \checkmark$
 $\langle proof \rangle$

lemma *WT-gpvD*:

assumes $\Gamma \vdash_g\ gpv\ \checkmark$
shows *WT-gpv-OutD*: $IO\ out\ c \in set\ spmf\ (the\ gpv\ gpv) \implies out \in outs\ \mathcal{I}\ \Gamma$
and *WT-gpv-ContD*: $\llbracket IO\ out\ c \in set\ spmf\ (the\ gpv\ gpv); input \in responses\ \mathcal{I}\ \Gamma\ out \rrbracket \implies \Gamma \vdash_g\ c\ input\ \checkmark$
 $\langle proof \rangle$

lemma *WT-gpv-mono*:

assumes *WT*: $\mathcal{I}1 \vdash_g\ gpv\ \checkmark$
and *outs*: $outs\ \mathcal{I}\ \mathcal{I}1 \subseteq outs\ \mathcal{I}\ \mathcal{I}2$
and *responses*: $\bigwedge x. x \in outs\ \mathcal{I}\ \mathcal{I}1 \implies responses\ \mathcal{I}\ \mathcal{I}2\ x \subseteq responses\ \mathcal{I}\ \mathcal{I}1\ x$
shows $\mathcal{I}2 \vdash_g\ gpv\ \checkmark$
 $\langle proof \rangle$

lemma *WT-gpv-Done* [*iff*]: $\Gamma \vdash_g\ Done\ x\ \checkmark$

$\langle proof \rangle$

lemma *WT-gpv-Fail* [*iff*]: $\Gamma \vdash_g\ Fail\ \checkmark$

$\langle proof \rangle$

lemma *WT-gpv-PauseI*:

$\llbracket out \in outs\ \mathcal{I}\ \Gamma; \bigwedge input. input \in responses\ \mathcal{I}\ \Gamma\ out \implies \Gamma \vdash_g\ c\ input\ \checkmark \rrbracket$
 $\implies \Gamma \vdash_g\ Pause\ out\ c\ \checkmark$
 $\langle proof \rangle$

lemma *WT-gpv-Pause* [*iff*]:

$\Gamma \vdash_g\ Pause\ out\ c\ \checkmark \iff out \in outs\ \mathcal{I}\ \Gamma \wedge (\forall input \in responses\ \mathcal{I}\ \Gamma\ out. \Gamma \vdash_g\ c\ input\ \checkmark)$
 $\langle proof \rangle$

lemma *WT-gpv-bindI*:

$\llbracket \Gamma \vdash_g\ gpv\ \checkmark; \bigwedge x. x \in results\ gpv\ \Gamma\ gpv \implies \Gamma \vdash_g\ f\ x\ \checkmark \rrbracket$
 $\implies \Gamma \vdash_g\ gpv\ \ggg\ f\ \checkmark$
 $\langle proof \rangle$

lemma *WT-gpv-bindD2*:

assumes *WT*: $\Gamma \vdash_g\ gpv\ \ggg\ f\ \checkmark$
and *x*: $x \in results\ gpv\ \Gamma\ gpv$
shows $\Gamma \vdash_g\ f\ x\ \checkmark$
 $\langle proof \rangle$

lemma *WT-gpv-bindD1*: $\Gamma \vdash_g \text{gpv} \ggg f \checkmark \implies \Gamma \vdash_g \text{gpv} \checkmark$
 ⟨proof⟩

lemma *WT-gpv-bind [simp]*: $\Gamma \vdash_g \text{gpv} \ggg f \checkmark \iff \Gamma \vdash_g \text{gpv} \checkmark \wedge (\forall x \in \text{results-gpv} \Gamma \text{ gpv}. \Gamma \vdash_g f x \checkmark)$
 ⟨proof⟩

lemma *WT-gpv-full [simp, intro!]*: $\mathcal{I}\text{-full} \vdash_g \text{gpv} \checkmark$
 ⟨proof⟩

lemma *WT-gpv-lift-spmf [simp, intro!]*: $\mathcal{I} \vdash_g \text{lift-spmf } p \checkmark$
 ⟨proof⟩

lemma *WT-gpv-coinduct-bind [consumes 1, case-names WT-gpv, case-conclusion WT-gpv out cont]*:

assumes *: $X \text{ gpv}$
and step: $\bigwedge \text{gpv out } c. \llbracket X \text{ gpv}; IO \text{ out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \rrbracket$
 $\implies \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out.}$
 $\quad X (c \text{ input}) \vee$
 $\quad \mathcal{I} \vdash_g c \text{ input} \checkmark \vee$
 $\quad (\exists (\text{gpv}' :: ('b, 'call, 'ret) \text{gpv}) f. c \text{ input} = \text{gpv}' \ggg f \wedge \mathcal{I} \vdash_g \text{gpv}' \checkmark \wedge$
 $(\forall x \in \text{results-gpv } \mathcal{I} \ \text{gpv}'. X (f x))))$
shows $\mathcal{I} \vdash_g \text{gpv} \checkmark$
 ⟨proof⟩

lemma *I-trivial-WT-gpvD [simp]*: $\mathcal{I}\text{-trivial } \mathcal{I} \implies \mathcal{I} \vdash_g \text{gpv} \checkmark$
 ⟨proof⟩

lemma *I-trivial-WT-gpvI*:
assumes $\bigwedge \text{gpv} :: ('a, 'out, 'in) \text{gpv}. \mathcal{I} \vdash_g \text{gpv} \checkmark$
shows $\mathcal{I}\text{-trivial } \mathcal{I}$
 ⟨proof⟩

lemma *WT-gpv-I-mono*: $\llbracket \mathcal{I} \vdash_g \text{gpv} \checkmark; \mathcal{I} \leq \mathcal{I}' \rrbracket \implies \mathcal{I}' \vdash_g \text{gpv} \checkmark$
 ⟨proof⟩

lemma *results-gpv-mono*:
assumes $le: \mathcal{I}' \leq \mathcal{I}$ **and** $WT: \mathcal{I}' \vdash_g \text{gpv} \checkmark$
shows $\text{results-gpv } \mathcal{I} \ \text{gpv} \subseteq \text{results-gpv } \mathcal{I}' \ \text{gpv}$
 ⟨proof⟩

lemma *WT-gpv-outs-gpv*:
assumes $\mathcal{I} \vdash_g \text{gpv} \checkmark$
shows $\text{outs-gpv } \mathcal{I} \ \text{gpv} \subseteq \text{outs-}\mathcal{I} \ \mathcal{I}$
 ⟨proof⟩

lemma *WT-gpv-map-gpv'*: $\mathcal{I} \vdash_g \text{map-gpv}' f g h \text{ gpv} \checkmark$ **if** $\text{map-}\mathcal{I} \ g \ h \ \mathcal{I} \vdash_g \text{gpv} \checkmark$
 ⟨proof⟩

lemma *WT-gpv-map-gpv*: $\mathcal{I} \vdash g \text{ map-gpv } f g \text{ gpv } \checkmark$ **if** $\text{map-}\mathcal{I} \text{ } g \text{ id } \mathcal{I} \vdash g \text{ gpv } \checkmark$
 ⟨proof⟩

lemma *results-gpv-map-gpv'* [*simp*]:
 $\text{results-gpv } \mathcal{I} (\text{map-gpv}' f g h \text{ gpv}) = f' (\text{results-gpv } (\text{map-}\mathcal{I} g h \mathcal{I}) \text{ gpv})$
 ⟨proof⟩

lemma *WT-gpv-parametric'*: **includes** *lifting-syntax* **shows**
 $\text{bi-unique } C \implies (\text{rel-}\mathcal{I} C R \implies \text{rel-gpv}'' A C R \implies (=)) \text{ WT-gpv } \text{ WT-gpv}$
 ⟨proof⟩

lemma *WT-gpv-map-gpv-id* [*simp*]: $\mathcal{I} \vdash g \text{ map-gpv } f \text{ id } \text{ gpv } \checkmark \iff \mathcal{I} \vdash g \text{ gpv } \checkmark$
 ⟨proof⟩

lemma *WT-gpv-outs-gpvI*:
assumes $\text{outs-gpv } \mathcal{I} \text{ gpv} \subseteq \text{outs-}\mathcal{I} \mathcal{I}$
shows $\mathcal{I} \vdash g \text{ gpv } \checkmark$
 ⟨proof⟩

lemma *WT-gpv-iff-outs-gpv*:
 $\mathcal{I} \vdash g \text{ gpv } \checkmark \iff \text{outs-gpv } \mathcal{I} \text{ gpv} \subseteq \text{outs-}\mathcal{I} \mathcal{I}$
 ⟨proof⟩

4.13 Sub-gpvs

context begin

qualified inductive *sub-gpvsp* :: $(\text{'out}, \text{'in}) \mathcal{I} \Rightarrow (\text{'a}, \text{'out}, \text{'in}) \text{ gpv} \Rightarrow (\text{'a}, \text{'out}, \text{'in}) \text{ gpv} \Rightarrow \text{bool}$

for $\mathcal{I} x$

where

base:

$\llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out}; x = c \text{ input} \rrbracket$
 $\implies \text{sub-gpvsp } \mathcal{I} x \text{ gpv}$

| *cont*:

$\llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out}; \text{sub-gpvsp } \mathcal{I} x (c \text{ input}) \rrbracket$
 $\implies \text{sub-gpvsp } \mathcal{I} x \text{ gpv}$

qualified lemma *sub-gpvsp-base*:

$\llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out} \rrbracket$
 $\implies \text{sub-gpvsp } \mathcal{I} (c \text{ input}) \text{ gpv}$

⟨proof⟩

definition *sub-gpvs* :: $(\text{'out}, \text{'in}) \mathcal{I} \Rightarrow (\text{'a}, \text{'out}, \text{'in}) \text{ gpv} \Rightarrow (\text{'a}, \text{'out}, \text{'in}) \text{ gpv} \text{ set}$
where $\text{sub-gpvs } \mathcal{I} \text{ gpv} \equiv \{x. \text{sub-gpvsp } \mathcal{I} x \text{ gpv}\}$

lemma *sub-gpvsp-sub-gpvs-eq* [*pred-set-conv*]: $\text{sub-gpvsp } \mathcal{I} x \text{ gpv} \iff x \in \text{sub-gpvs } \mathcal{I} \text{ gpv}$
 ⟨proof⟩

context begin

$\langle ML \rangle$

lemmas *intros* [*intro*?] = *sub-gpvsp.intros*[*to-set*]
and *base* = *sub-gpvsp-base*[*to-set*]
and *cont* = *cont*[*to-set*]
and *induct* [*consumes 1*, *case-names Pure IO*, *induct set: sub-gpvs*] = *sub-gpvsp.induct*[*to-set*]
and *cases* [*consumes 1*, *case-names Pure IO*, *cases set: sub-gpvs*] = *sub-gpvsp.cases*[*to-set*]
and *simps* = *sub-gpvsp.simps*[*to-set*]
end
end

lemma *WT-sub-gpvsD*:

assumes $\mathcal{I} \vdash_g \text{gpv} \checkmark$ **and** $\text{gpv}' \in \text{sub-gpvs } \mathcal{I} \text{ gpv}$

shows $\mathcal{I} \vdash_g \text{gpv}' \checkmark$

$\langle \text{proof} \rangle$

lemma *WT-sub-gpvsI*:

$\llbracket \bigwedge \text{out } c. \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \implies \text{out} \in \text{outs-}\mathcal{I} \ \Gamma;$

$\bigwedge \text{gpv}'. \text{gpv}' \in \text{sub-gpvs } \Gamma \ \text{gpv} \implies \Gamma \vdash_g \text{gpv}' \checkmark \rrbracket$

$\implies \Gamma \vdash_g \text{gpv} \checkmark$

$\langle \text{proof} \rangle$

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* :: $(('a, 'out, 'in) \ \text{gpv} \Rightarrow \text{bool}) \Rightarrow (('a, 'out, 'in) \ \text{gpv} \Rightarrow \text{bool})$

and *co'* :: *bool*

defines $F \equiv \lambda \text{gen-lossless-gpv } \text{gpv}. \exists \text{pa}. \text{gpv} = \text{GPV } \text{pa} \wedge$

$\text{lossless-spmf } \text{pa} \wedge (\forall \text{out } c \ \text{input}. \text{IO out } c \in \text{set-spmf } \text{pa} \longrightarrow \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out} \longrightarrow \text{gen-lossless-gpv } (c \ \text{input}))$

and $\text{co}' \equiv \text{co}$ — 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*

$\langle \text{proof} \rangle$

definition $gen\text{-}lossless\text{-}gpv :: ('a, 'out, 'in) gpv \Rightarrow bool$
where $gen\text{-}lossless\text{-}gpv = (if\ co'\ then\ gfp\ else\ lfp)\ F$

lemma $gen\text{-}lossless\text{-}gpv\text{-}unfold: gen\text{-}lossless\text{-}gpv = F\ gen\text{-}lossless\text{-}gpv$
 $\langle proof \rangle$

lemma $gen\text{-}lossless\text{-}gpv\text{-}True: co' = True \Longrightarrow gen\text{-}lossless\text{-}gpv \equiv gfp\ F$
and $gen\text{-}lossless\text{-}gpv\text{-}False: co' = False \Longrightarrow gen\text{-}lossless\text{-}gpv \equiv lfp\ F$
 $\langle proof \rangle$

lemma $gen\text{-}lossless\text{-}gpv\text{-}cases$ [$elim?$, $cases\ pred$]:
assumes $gen\text{-}lossless\text{-}gpv\ gpv$
obtains $(gen\text{-}lossless\text{-}gpv)\ p$ **where** $gpv = GPV\ p\ lossless\text{-}spmf\ p$
 $\bigwedge out\ c\ input. \llbracket IO\ out\ c \in set\text{-}spmf\ p; input \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out \rrbracket \Longrightarrow gen\text{-}lossless\text{-}gpv$
 $(c\ input)$
 $\langle proof \rangle$

lemma $gen\text{-}lossless\text{-}gpvD$:
assumes $gen\text{-}lossless\text{-}gpv\ gpv$
shows $gen\text{-}lossless\text{-}gpv\text{-}lossless\text{-}spmfD: lossless\text{-}spmf\ (the\text{-}gpv\ gpv)$
and $gen\text{-}lossless\text{-}gpv\text{-}continuationD$:
 $\llbracket IO\ out\ c \in set\text{-}spmf\ (the\text{-}gpv\ gpv); input \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out \rrbracket \Longrightarrow gen\text{-}lossless\text{-}gpv$
 $(c\ input)$
 $\langle proof \rangle$

lemma $gen\text{-}lossless\text{-}gpv\text{-}intros$:
 $\llbracket lossless\text{-}spmf\ p;$
 $\bigwedge out\ c\ input. \llbracket IO\ out\ c \in set\text{-}spmf\ p; input \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out \rrbracket \Longrightarrow$
 $gen\text{-}lossless\text{-}gpv\ (c\ input) \rrbracket$
 $\Longrightarrow gen\text{-}lossless\text{-}gpv\ (GPV\ p)$
 $\langle proof \rangle$

lemma $gen\text{-}lossless\text{-}gpvI$ [$intro?$]:
 $\llbracket lossless\text{-}spmf\ (the\text{-}gpv\ gpv);$
 $\bigwedge out\ c\ input. \llbracket IO\ out\ c \in set\text{-}spmf\ (the\text{-}gpv\ gpv); input \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out \rrbracket$
 $\Longrightarrow gen\text{-}lossless\text{-}gpv\ (c\ input) \rrbracket$
 $\Longrightarrow gen\text{-}lossless\text{-}gpv\ gpv$
 $\langle proof \rangle$

lemma $gen\text{-}lossless\text{-}gpv\text{-}simps$:
 $gen\text{-}lossless\text{-}gpv\ gpv \longleftrightarrow$
 $(\exists p. gpv = GPV\ p \wedge lossless\text{-}spmf\ p \wedge (\forall out\ c\ input.$
 $IO\ out\ c \in set\text{-}spmf\ p \longrightarrow input \in responses\text{-}\mathcal{I}\ \mathcal{I}\ out \longrightarrow gen\text{-}lossless\text{-}gpv$
 $(c\ input)))$
 $\langle proof \rangle$

lemma $gen\text{-}lossless\text{-}gpv\text{-}Done$ [iff]: $gen\text{-}lossless\text{-}gpv\ (Done\ x)$
 $\langle proof \rangle$

lemma *gen-lossless-gpv-Fail* [iff]: $\neg \text{gen-lossless-gpv Fail}$
 ⟨proof⟩

lemma *gen-lossless-gpv-Pause* [simp]:
 $\text{gen-lossless-gpv (Pause out } c) \longleftrightarrow (\forall \text{ input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out. } \text{gen-lossless-gpv}$
 $(c \text{ input}))$
 ⟨proof⟩

lemma *gen-lossless-gpv-lift-spmf* [iff]: $\text{gen-lossless-gpv (lift-spmf } p) \longleftrightarrow \text{lossless-spmf}$
 p
 ⟨proof⟩

end

lemma *gen-lossless-gpv-assert-gpv* [iff]: $\text{gen-lossless-gpv co } \mathcal{I} (\text{assert-gpv } b) \longleftrightarrow b$
 ⟨proof⟩

abbreviation *lossless-gpv* :: $('out, 'in) \mathcal{I} \Rightarrow ('a, 'out, 'in) \text{gpv} \Rightarrow \text{bool}$
where $\text{lossless-gpv} \equiv \text{gen-lossless-gpv False}$

abbreviation *colossless-gpv* :: $('out, 'in) \mathcal{I} \Rightarrow ('a, 'out, 'in) \text{gpv} \Rightarrow \text{bool}$
where $\text{colossless-gpv} \equiv \text{gen-lossless-gpv True}$

lemma *lossless-gpv-induct* [consumes 1, case-names *lossless-gpv*, induct *pred*]:
assumes *: $\text{lossless-gpv } \mathcal{I} \text{ gpv}$
and step: $\bigwedge p. \llbracket \text{lossless-spmf } p;$
 $\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } p; \text{ input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} \rrbracket \Longrightarrow \text{lossless-gpv}$
 $\mathcal{I} (c \text{ input});$
 $\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } p; \text{ input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} \rrbracket \Longrightarrow P (c$
 $\text{input}) \rrbracket$
 $\Longrightarrow P (\text{GPV } p)$
shows $P \text{ gpv}$
 ⟨proof⟩

lemma *colossless-gpv-coinduct*
 [consumes 1, case-names *colossless-gpv*, case-conclusion *colossless-gpv lossless-spmf*
continuation, coinduct *pred*]:
assumes *: $X \text{ gpv}$
and step: $\bigwedge \text{gpv. } X \text{ gpv} \Longrightarrow \text{lossless-spmf (the-gpv gpv)} \wedge (\forall \text{ out } c \text{ input.}$
 $\text{IO out } c \in \text{set-spmf (the-gpv gpv)} \longrightarrow \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} \longrightarrow X (c$
 $\text{input}) \vee \text{colossless-gpv } \mathcal{I} (c \text{ input}))$
shows $\text{colossless-gpv } \mathcal{I} \text{ gpv}$
 ⟨proof⟩

lemmas $\text{lossless-gpvI} = \text{gen-lossless-gpvI}[\text{where } \text{co}=\text{False}]$
and $\text{lossless-gpvD} = \text{gen-lossless-gpvD}[\text{where } \text{co}=\text{False}]$
and $\text{lossless-gpv-lossless-spmfD} = \text{gen-lossless-gpv-lossless-spmfD}[\text{where } \text{co}=\text{False}]$
and $\text{lossless-gpv-continuationD} = \text{gen-lossless-gpv-continuationD}[\text{where } \text{co}=\text{False}]$

lemmas $colossless-gpvI = gen-lossless-gpvI$ [**where** $co=True$]
and $colossless-gpvD = gen-lossless-gpvD$ [**where** $co=True$]
and $colossless-gpv-lossless-spmfD = gen-lossless-gpv-lossless-spmfD$ [**where** $co=True$]
and $colossless-gpv-continuationD = gen-lossless-gpv-continuationD$ [**where** $co=True$]

lemma $gen-lossless-bind-gpvI$:
assumes $gen-lossless-gpv\ co\ \mathcal{I}\ gpv \wedge x. x \in results-gpv\ \mathcal{I}\ gpv \implies gen-lossless-gpv\ co\ \mathcal{I}\ (f\ x)$
shows $gen-lossless-gpv\ co\ \mathcal{I}\ (gpv \ggg f)$
 $\langle proof \rangle$

lemmas $lossless-bind-gpvI = gen-lossless-bind-gpvI$ [**where** $co=False$]
and $colossless-bind-gpvI = gen-lossless-bind-gpvI$ [**where** $co=True$]

lemma $gen-lossless-bind-gpvD1$:
assumes $gen-lossless-gpv\ co\ \mathcal{I}\ (gpv \ggg f)$
shows $gen-lossless-gpv\ co\ \mathcal{I}\ gpv$
 $\langle proof \rangle$

lemmas $lossless-bind-gpvD1 = gen-lossless-bind-gpvD1$ [**where** $co=False$]
and $colossless-bind-gpvD1 = gen-lossless-bind-gpvD1$ [**where** $co=True$]

lemma $gen-lossless-bind-gpvD2$:
assumes $gen-lossless-gpv\ co\ \mathcal{I}\ (gpv \ggg f)$
and $x \in results-gpv\ \mathcal{I}\ gpv$
shows $gen-lossless-gpv\ co\ \mathcal{I}\ (f\ x)$
 $\langle proof \rangle$

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*]:
 $gen-lossless-gpv\ co\ \mathcal{I}\ (gpv \ggg f) \longleftrightarrow gen-lossless-gpv\ co\ \mathcal{I}\ gpv \wedge (\forall x \in results-gpv\ \mathcal{I}\ gpv. gen-lossless-gpv\ co\ \mathcal{I}\ (f\ x))$
 $\langle proof \rangle$

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 $lossless-gpv\ \mathcal{I}'\ gpv'$
 $\langle proof \rangle$

lemma $rel-gpv''-lossless-gpvD2$:

$$\llbracket \text{rel-gpv}'' A C R \text{ gpv } \text{gpv}'; \text{lossless-gpv } \mathcal{I}' \text{ gpv}'; \text{rel-}\mathcal{I} C R \mathcal{I} \mathcal{I}' \rrbracket$$

$$\implies \text{lossless-gpv } \mathcal{I} \text{ gpv}$$
 <proof>

lemma *rel-gpv-lossless-gpvD1*:

$$\llbracket \text{rel-gpv } A C \text{ gpv } \text{gpv}'; \text{lossless-gpv } \mathcal{I} \text{ gpv}; \text{rel-}\mathcal{I} C (=) \mathcal{I} \mathcal{I}' \rrbracket \implies \text{lossless-gpv } \mathcal{I}' \text{ gpv}'$$
 <proof>

lemma *rel-gpv-lossless-gpvD2*:

$$\llbracket \text{rel-gpv } A C \text{ gpv } \text{gpv}'; \text{lossless-gpv } \mathcal{I}' \text{ gpv}'; \text{rel-}\mathcal{I} C (=) \mathcal{I} \mathcal{I}' \rrbracket$$

$$\implies \text{lossless-gpv } \mathcal{I} \text{ gpv}$$
 <proof>

lemma *rel-gpv''-colossless-gpvD1*:
assumes *rel*: $\text{rel-gpv}'' A C R \text{ gpv } \text{gpv}'$
and *gpv*: *colossless-gpv* $\mathcal{I} \text{ gpv}$
and [*transfer-rule*]: $\text{rel-}\mathcal{I} C R \mathcal{I} \mathcal{I}'$
shows *colossless-gpv* $\mathcal{I}' \text{ gpv}'$
 <proof>

lemma *rel-gpv''-colossless-gpvD2*:

$$\llbracket \text{rel-gpv}'' A C R \text{ gpv } \text{gpv}'; \text{colossless-gpv } \mathcal{I}' \text{ gpv}'; \text{rel-}\mathcal{I} C R \mathcal{I} \mathcal{I}' \rrbracket$$

$$\implies \text{colossless-gpv } \mathcal{I} \text{ gpv}$$
 <proof>

lemma *rel-gpv-colossless-gpvD1*:

$$\llbracket \text{rel-gpv } A C \text{ gpv } \text{gpv}'; \text{colossless-gpv } \mathcal{I} \text{ gpv}; \text{rel-}\mathcal{I} C (=) \mathcal{I} \mathcal{I}' \rrbracket \implies \text{colossless-gpv } \mathcal{I}' \text{ gpv}'$$
 <proof>

lemma *rel-gpv-colossless-gpvD2*:

$$\llbracket \text{rel-gpv } A C \text{ gpv } \text{gpv}'; \text{colossless-gpv } \mathcal{I}' \text{ gpv}'; \text{rel-}\mathcal{I} C (=) \mathcal{I} \mathcal{I}' \rrbracket$$

$$\implies \text{colossless-gpv } \mathcal{I} \text{ gpv}$$
 <proof>

lemma *gen-lossless-gpv-parametric'*:

$$((=) \implies \text{rel-}\mathcal{I} C R \implies \text{rel-gpv}'' A C R \implies (=))$$

$$\text{gen-lossless-gpv } \text{gen-lossless-gpv}$$
 <proof>

lemma *gen-lossless-gpv-parametric* [*transfer-rule*]:

$$((=) \implies \text{rel-}\mathcal{I} C (=) \implies \text{rel-gpv } A C \implies (=))$$

$$\text{gen-lossless-gpv } \text{gen-lossless-gpv}$$
 <proof>

end

lemma *gen-lossless-gpv-map-full* [*simp*]:

$gen\text{-}lossless\text{-}gpv\ b\ \mathcal{I}\text{-full}\ (map\text{-}gpv\ f\ g\ gpv) = gen\text{-}lossless\text{-}gpv\ b\ \mathcal{I}\text{-full}\ gpv$
 (is ?lhs = ?rhs)
 <proof>

lemma *gen-lossless-gpv-map-id* [simp]:
 $gen\text{-}lossless\text{-}gpv\ b\ \mathcal{I}\ (map\text{-}gpv\ f\ id\ gpv) = gen\text{-}lossless\text{-}gpv\ b\ \mathcal{I}\ gpv$
 <proof>

lemma *results-gpv-try-gpv* [simp]:
 $results\text{-}gpv\ \mathcal{I}\ (TRY\ gpv\ ELSE\ gpv') =$
 $results\text{-}gpv\ \mathcal{I}\ gpv \cup (if\ colossless\text{-}gpv\ \mathcal{I}\ gpv\ then\ \{\}\ else\ results\text{-}gpv\ \mathcal{I}\ gpv')$
 (is ?lhs = ?rhs)
 <proof>

lemma *results'-gpv-try-gpv* [simp]:
 $results'\text{-}gpv\ (TRY\ gpv\ ELSE\ gpv') =$
 $results'\text{-}gpv\ gpv \cup (if\ colossless\text{-}gpv\ \mathcal{I}\text{-full}\ gpv\ then\ \{\}\ else\ results'\text{-}gpv\ gpv')$
 <proof>

lemma *outs'-gpv-try-gpv* [simp]:
 $outs'\text{-}gpv\ (TRY\ gpv\ ELSE\ gpv') =$
 $outs'\text{-}gpv\ gpv \cup (if\ colossless\text{-}gpv\ \mathcal{I}\text{-full}\ gpv\ then\ \{\}\ else\ outs'\text{-}gpv\ gpv')$
 (is ?lhs = ?rhs)
 <proof>

lemma *pred-gpv-try* [simp]:
 $pred\text{-}gpv\ P\ Q\ (try\text{-}gpv\ gpv\ gpv') = (pred\text{-}gpv\ P\ Q\ gpv \wedge (\neg\ colossless\text{-}gpv\ \mathcal{I}\text{-full}\ gpv \longrightarrow pred\text{-}gpv\ P\ Q\ gpv'))$
 <proof>

lemma *lossless-WT-gpv-induct* [consumes 2, case-names lossless-gpv]:
assumes *lossless*: $lossless\text{-}gpv\ \mathcal{I}\ gpv$
and *WT*: $\mathcal{I} \vdash g\ gpv\ \checkmark$
and *step*: $\bigwedge p.\ \llbracket$
 $lossless\text{-}spmf\ p;$
 $\bigwedge out\ c.\ IO\ out\ c \in set\text{-}spmf\ p \implies out \in outs\text{-}\mathcal{I}\ \mathcal{I};$
 $\bigwedge out\ c\ input.\ \llbracket IO\ out\ c \in set\text{-}spmf\ p; out \in outs\text{-}\mathcal{I}\ \mathcal{I} \implies input \in responses\text{-}\mathcal{I}$
 $\mathcal{I}\ out \rrbracket \implies lossless\text{-}gpv\ \mathcal{I}\ (c\ input);$
 $\bigwedge out\ c\ input.\ \llbracket IO\ out\ c \in set\text{-}spmf\ p; out \in outs\text{-}\mathcal{I}\ \mathcal{I} \implies input \in responses\text{-}\mathcal{I}$
 $\mathcal{I}\ out \rrbracket \implies \mathcal{I} \vdash g\ c\ input\ \checkmark;$
 $\bigwedge out\ c\ input.\ \llbracket IO\ out\ c \in set\text{-}spmf\ p; out \in outs\text{-}\mathcal{I}\ \mathcal{I} \implies input \in responses\text{-}\mathcal{I}$
 $\mathcal{I}\ out \rrbracket \implies P\ (c\ input)\rrbracket$
 $\implies P\ (GPV\ p)$
shows $P\ gpv$
 <proof>

lemma *lossless-gpv-induct-strong* [consumes 1, case-names lossless-gpv]:
assumes *gpv*: $lossless\text{-}gpv\ \mathcal{I}\ gpv$
and *step*:

$\bigwedge p. \llbracket \text{lossless-spmf } p; \text{ } \bigwedge gpv. gpv \in \text{sub-gpvs } \mathcal{I} (GPV p) \implies \text{lossless-gpv } \mathcal{I} gpv; \bigwedge gpv. gpv \in \text{sub-gpvs } \mathcal{I} (GPV p) \implies P gpv \rrbracket$
 $\implies P (GPV p)$
shows $P gpv$
 $\langle \text{proof} \rangle$

lemma *lossless-sub-gpvsI*:
assumes *spmf*: *lossless-spmf* (*the-gpv gpv*)
and *sub*: $\bigwedge gpv'. gpv' \in \text{sub-gpvs } \mathcal{I} gpv \implies \text{lossless-gpv } \mathcal{I} gpv'$
shows *lossless-gpv* $\mathcal{I} gpv$
 $\langle \text{proof} \rangle$

lemma *lossless-sub-gpvsD*:
assumes *lossless-gpv* $\mathcal{I} gpv$ $gpv' \in \text{sub-gpvs } \mathcal{I} gpv$
shows *lossless-gpv* $\mathcal{I} gpv'$
 $\langle \text{proof} \rangle$

lemma *lossless-WT-gpv-induct-strong* [*consumes 2, case-names lossless-gpv*]:
assumes *lossless*: *lossless-gpv* $\mathcal{I} gpv$
and *WT*: $\mathcal{I} \vdash_g gpv \checkmark$
and *step*: $\bigwedge p. \llbracket \text{lossless-spmf } p; \bigwedge \text{out } c. IO \text{ out } c \in \text{set-spmf } p \implies \text{out} \in \text{outs-}\mathcal{I} \mathcal{I}; \bigwedge gpv. gpv \in \text{sub-gpvs } \mathcal{I} (GPV p) \implies \text{lossless-gpv } \mathcal{I} gpv; \bigwedge gpv. gpv \in \text{sub-gpvs } \mathcal{I} (GPV p) \implies \mathcal{I} \vdash_g gpv \checkmark; \bigwedge gpv. gpv \in \text{sub-gpvs } \mathcal{I} (GPV p) \implies P gpv \rrbracket$
 $\implies P (GPV p)$
shows $P gpv$
 $\langle \text{proof} \rangle$

lemma *try-gpv-gen-lossless*: — TODO: generalise to arbitrary typings?
 $\text{gen-lossless-gpv } b \mathcal{I}\text{-full } gpv \implies (TRY gpv ELSE gpv') = gpv$
 $\langle \text{proof} \rangle$

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?
 $\text{gen-lossless-gpv } b \mathcal{I}\text{-full } gpv \implies TRY \text{ bind-gpv } gpv f ELSE gpv' = \text{bind-gpv } gpv$
 $(\lambda x. TRY f x ELSE gpv')$
 $\langle \text{proof} \rangle$

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*:
 $\llbracket gpv = gpv''; \neg \text{colossless-gpv } \mathcal{I}\text{-full } gpv'' \implies gpv' = gpv''' \rrbracket$
 $\implies \text{try-gpv } gpv gpv' = \text{try-gpv } gpv'' gpv'''$
 $\langle \text{proof} \rangle$

context fixes $B :: 'b \Rightarrow 'c$ set and $x :: 'a$ begin

primcorec $mk\text{-}lossless\text{-}gpv :: ('a, 'b, 'c) gpv \Rightarrow ('a, 'b, 'c) gpv$ **where**
 $the\text{-}gpv (mk\text{-}lossless\text{-}gpv gpv) =$
 $map\text{-}spmf (\lambda generat. case generat of Pure x \Rightarrow Pure x$
 $| IO out c \Rightarrow IO out (\lambda input. if input \in B out then mk\text{-}lossless\text{-}gpv (c input)$
 $else Done x))$
 $(the\text{-}gpv gpv)$

end

lemma $WT\text{-}gpv\text{-}mk\text{-}lossless\text{-}gpv$:
assumes $\mathcal{I} \vdash g gpv \checkmark$
and $outs: outs\text{-}\mathcal{I} \ \mathcal{I}' = outs\text{-}\mathcal{I} \ \mathcal{I}$
shows $\mathcal{I}' \vdash g mk\text{-}lossless\text{-}gpv (responses\text{-}\mathcal{I} \ \mathcal{I}) x gpv \checkmark$
 $\langle proof \rangle$

4.15 Sequencing with failure handling included

definition $catch\text{-}gpv :: ('a, 'out, 'in) gpv \Rightarrow ('a option, 'out, 'in) gpv$
where $catch\text{-}gpv gpv = TRY map\text{-}gpv Some id gpv ELSE Done None$

lemma $catch\text{-}gpv\text{-}Done [simp]$: $catch\text{-}gpv (Done x) = Done (Some x)$
 $\langle proof \rangle$

lemma $catch\text{-}gpv\text{-}Fail [simp]$: $catch\text{-}gpv Fail = Done None$
 $\langle proof \rangle$

lemma $catch\text{-}gpv\text{-}Pause [simp]$: $catch\text{-}gpv (Pause out rpv) = Pause out (\lambda input. catch\text{-}gpv (rpv input))$
 $\langle proof \rangle$

lemma $catch\text{-}gpv\text{-}lift\text{-}spmf [simp]$: $catch\text{-}gpv (lift\text{-}spmf p) = lift\text{-}spmf (spmf\text{-}of\text{-}pmf p)$
 $\langle proof \rangle$

lemma $catch\text{-}gpv\text{-}assert [simp]$: $catch\text{-}gpv (assert\text{-}gpv b) = Done (assert\text{-}option b)$
 $\langle proof \rangle$

lemma $catch\text{-}gpv\text{-}sel [simp]$:
 $the\text{-}gpv (catch\text{-}gpv gpv) =$
 $TRY map\text{-}spmf (map\text{-}generat Some id (\lambda rpv input. catch\text{-}gpv (rpv input)))$
 $(the\text{-}gpv gpv)$
 $ELSE return\text{-}spmf (Pure None)$
 $\langle proof \rangle$

lemma $catch\text{-}gpv\text{-}bind\text{-}gpv$: $catch\text{-}gpv (bind\text{-}gpv gpv f) = bind\text{-}gpv (catch\text{-}gpv gpv)$
 $(\lambda x. case x of None \Rightarrow Done None | Some x' \Rightarrow catch\text{-}gpv (f x'))$

<proof>

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

lemma *catch-gpv-parametric* [*transfer-rule*]:

$(rel\text{-}gpv\ A\ C\ ==>\ rel\text{-}gpv\ (rel\text{-}option\ A)\ C)\ catch\text{-}gpv\ catch\text{-}gpv$

<proof>

lemma *catch-gpv-parametric'*:

notes [*transfer-rule*] = *try-gpv-parametric' map-gpv-parametric' Done-parametric'*

shows $(rel\text{-}gpv''\ A\ C\ R\ ==>\ rel\text{-}gpv''\ (rel\text{-}option\ A)\ C\ R)\ catch\text{-}gpv\ catch\text{-}gpv$

<proof>

end

lemma *catch-gpv-map'*: $catch\text{-}gpv\ (map\text{-}gpv'\ f\ g\ h\ gpv) = map\text{-}gpv'\ (map\text{-}option\ f)\ g\ h\ (catch\text{-}gpv\ gpv)$

<proof>

lemma *catch-gpv-map*: $catch\text{-}gpv\ (map\text{-}gpv\ f\ g\ gpv) = map\text{-}gpv\ (map\text{-}option\ f)\ g\ (catch\text{-}gpv\ gpv)$

<proof>

lemma *colossless-gpv-catch-gpv* [*simp*]: $colossless\text{-}gpv\ \mathcal{I}\text{-full}\ (catch\text{-}gpv\ gpv)$

<proof>

lemma *colossless-gpv-catch-gpv-conv-map*:

$colossless\text{-}gpv\ \mathcal{I}\text{-full}\ gpv\ ==>\ catch\text{-}gpv\ gpv = map\text{-}gpv\ Some\ id\ gpv$

<proof>

lemma *catch-gpv-catch-gpv* [*simp*]: $catch\text{-}gpv\ (catch\text{-}gpv\ gpv) = map\text{-}gpv\ Some\ id\ (catch\text{-}gpv\ gpv)$

<proof>

lemma *case-map-resumption*:

$case\text{-}resumption\ done\ pause\ (map\text{-}resumption\ f\ g\ r) =$

$case\text{-}resumption\ (done\ \circ\ map\text{-}option\ f)\ (\lambda out\ c.\ pause\ (g\ out)\ (map\text{-}resumption\ f\ g\ \circ\ c))\ r$

<proof>

lemma *catch-gpv-lift-resumption* [*simp*]: $catch\text{-}gpv\ (lift\text{-}resumption\ r) = lift\text{-}resumption\ (map\text{-}resumption\ Some\ id\ r)$

<proof>

lemma *results-gpv-catch-gpv*:

$results\text{-}gpv\ \mathcal{I}\ (catch\text{-}gpv\ gpv) = Some\ ' results\text{-}gpv\ \mathcal{I}\ gpv\ \cup\ (if\ colossless\text{-}gpv\ \mathcal{I}\ gpv\ then\ \{\}\ else\ \{None\})$

<proof>

lemma *Some-in-results-gpv-catch-gpv* [*simp*]:

$Some\ x\ \in\ results\text{-}gpv\ \mathcal{I}\ (catch\text{-}gpv\ gpv)\ \longleftrightarrow\ x\ \in\ results\text{-}gpv\ \mathcal{I}\ gpv$

$\langle \text{proof} \rangle$

lemma *None-in-results-gpv-catch-gpv* [simp]:

$\text{None} \in \text{results-gpv } \mathcal{I} (\text{catch-gpv } \text{gpv}) \longleftrightarrow \neg \text{colossless-gpv } \mathcal{I} \text{ gpv}$

$\langle \text{proof} \rangle$

lemma *results'-gpv-catch-gpv*:

$\text{results}'\text{-gpv } (\text{catch-gpv } \text{gpv}) = \text{Some } \text{'results}'\text{-gpv } \text{gpv} \cup (\text{if } \text{colossless-gpv } \mathcal{I}\text{-full } \text{gpv } \text{ then } \{\} \text{ else } \{\text{None}\})$

$\langle \text{proof} \rangle$

lemma *Some-in-results'-gpv-catch-gpv* [simp]:

$\text{Some } x \in \text{results}'\text{-gpv } (\text{catch-gpv } \text{gpv}) \longleftrightarrow x \in \text{results}'\text{-gpv } \text{gpv}$

$\langle \text{proof} \rangle$

lemma *None-in-results'-gpv-catch-gpv* [simp]:

$\text{None} \in \text{results}'\text{-gpv } (\text{catch-gpv } \text{gpv}) \longleftrightarrow \neg \text{colossless-gpv } \mathcal{I}\text{-full } \text{gpv}$

$\langle \text{proof} \rangle$

lemma *results'-gpv-catch-gpvE*:

assumes $x \in \text{results}'\text{-gpv } (\text{catch-gpv } \text{gpv})$

obtains $(\text{Some}) x'$

where $x = \text{Some } x' \ x' \in \text{results}'\text{-gpv } \text{gpv}$

$| (\text{colossless}) x = \text{None} \ \neg \text{colossless-gpv } \mathcal{I}\text{-full } \text{gpv}$

$\langle \text{proof} \rangle$

lemma *outs'-gpv-catch-gpv* [simp]: $\text{outs}'\text{-gpv } (\text{catch-gpv } \text{gpv}) = \text{outs}'\text{-gpv } \text{gpv}$

$\langle \text{proof} \rangle$

lemma *pred-gpv-catch-gpv* [simp]: $\text{pred-gpv } (\text{pred-option } P) Q (\text{catch-gpv } \text{gpv}) = \text{pred-gpv } P Q \text{ gpv}$

$\langle \text{proof} \rangle$

abbreviation $\text{bind-gpv}' :: ('a, 'call, 'ret) \text{ gpv} \Rightarrow ('a \text{ option} \Rightarrow ('b, 'call, 'ret) \text{ gpv}) \Rightarrow ('b, 'call, 'ret) \text{ gpv}$

where $\text{bind-gpv}' \text{ gpv} \equiv \text{bind-gpv } (\text{catch-gpv } \text{gpv})$

lemma *bind-gpv'-assoc* [simp]: $\text{bind-gpv}' (\text{bind-gpv}' \text{ gpv } f) g = \text{bind-gpv}' \text{ gpv } (\lambda x. \text{bind-gpv}' (f x) g)$

$\langle \text{proof} \rangle$

lemma *bind-gpv'-bind-gpv*: $\text{bind-gpv}' (\text{bind-gpv } \text{gpv } f) g = \text{bind-gpv}' \text{ gpv } (\text{case-option } (g \text{ None}) (\lambda y. \text{bind-gpv}' (f y) g))$

$\langle \text{proof} \rangle$

lemma *bind-gpv'-cong*:

$\llbracket \text{gpv} = \text{gpv}' ; \bigwedge x. x \in \text{Some } \text{'results}'\text{-gpv } \text{gpv}' \vee (\neg \text{colossless-gpv } \mathcal{I}\text{-full } \text{gpv} \wedge x$

$= \text{None}) \implies f x = f' x \]$
 $\implies \text{bind-gpv}' \text{ gpv } f = \text{bind-gpv}' \text{ gpv}' f'$
 <proof>

lemma *bind-gpv'-cong2*:

$\llbracket \text{gpv} = \text{gpv}'; \bigwedge x. x \in \text{results}'\text{-gpv } \text{gpv}' \implies f (\text{Some } x) = f' (\text{Some } x); \neg \text{coloss-}$
 $\text{less-gpv } \mathcal{I}\text{-full } \text{gpv} \implies f \text{ None} = f' \text{ None} \rrbracket$
 $\implies \text{bind-gpv}' \text{ gpv } f = \text{bind-gpv}' \text{ gpv}' f'$
 <proof>

4.16 Inlining

lemma *gpv-coinduct-bind* [*consumes 1, case-names Eq-gpv*]:

fixes $\text{gpv } \text{gpv}' :: ('a, 'call, 'ret) \text{ gpv}$
assumes $*$: $R \text{ gpv } \text{gpv}'$
and step: $\bigwedge \text{gpv } \text{gpv}'. R \text{ gpv } \text{gpv}'$
 $\implies \text{rel-spmf } (\text{rel-generat } (=) (=) (\text{rel-fun } (=) (\lambda \text{gpv } \text{gpv}'. R \text{ gpv } \text{gpv}' \vee \text{gpv} =$
 $\text{gpv}' \vee$
 $(\exists \text{gpv}2 :: ('b, 'call, 'ret) \text{ gpv}. \exists \text{gpv}2' :: ('c, 'call, 'ret) \text{ gpv}. \exists f f'. \text{gpv} =$
 $\text{bind-gpv } \text{gpv}2 f \wedge \text{gpv}' = \text{bind-gpv } \text{gpv}2' f' \wedge$
 $\text{rel-gpv } (\lambda x y. R (f x) (f' y)) (=) \text{gpv}2 \text{ gpv}2'))$
 $(\text{the-gpv } \text{gpv}) (\text{the-gpv } \text{gpv}'))$
shows $\text{gpv} = \text{gpv}'$
 <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 $\text{callee} :: 's \Rightarrow 'call \Rightarrow ('ret \times 's, 'call', 'ret') \text{ gpv}$
notes $\llbracket \text{function-internals} \rrbracket$
begin

partial-function (*spmf*) *inline1*

$:: ('a, 'call, 'ret) \text{ gpv} \Rightarrow 's$
 $\Rightarrow ('a \times 's + 'call' \times ('ret \times 's, 'call', 'ret') \text{ rpv} \times ('a, 'call, 'ret) \text{ rpv}) \text{ spmf}$

where

$\text{inline1 } \text{gpv } s =$
 $\text{the-gpv } \text{gpv} \gg=$
 $\text{case-generat } (\lambda x. \text{return-spmf } (\text{Inl } (x, s)))$
 $(\lambda \text{out } \text{rpv}. \text{the-gpv } (\text{callee } s \text{ out}) \gg=$
 $\text{case-generat } (\lambda(x, y). \text{inline1 } (\text{rpv } x) y)$
 $(\lambda \text{out } \text{rpv}'. \text{return-spmf } (\text{Inr } (\text{out}, \text{rpv}', \text{rpv}))))$

lemma *inline1-unfold*:

$\text{inline1 } \text{gpv } s =$
 $\text{the-gpv } \text{gpv} \gg=$
 $\text{case-generat } (\lambda x. \text{return-spmf } (\text{Inl } (x, s)))$
 $(\lambda \text{out } \text{rpv}. \text{the-gpv } (\text{callee } s \text{ out}) \gg=$

$case_generat (\lambda(x, y). inline1 (rpv\ x)\ y)$
 $(\lambda out\ rpv'. return_spmf (Inr (out, rpv', rpv))))$
 <proof>

lemma *inline1-fixp-induct* [case-names adm bottom step]:

assumes *ccpo.admissible (fun-lub lub-spmf) (fun-ord (ord-spmf (=)))* ($\lambda inline1'$.
 $P (\lambda gpv\ s. inline1' (gpv, s))$)
and $P (\lambda - -. return_pmf\ None)$
and $\bigwedge inline1'. P\ inline1' \implies P (\lambda gpv\ s. the_gpv\ gpv \ggg case_generat (\lambda x.$
 $return_spmf (Inl (x, s))) (\lambda out\ rpv. the_gpv (callee\ s\ out) \ggg case_generat (\lambda(x, y).$
 $inline1' (rpv\ x)\ y) (\lambda out\ rpv'. return_spmf (Inr (out, rpv', rpv))))$)
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 (=)))* ($\lambda inline1'$.
 $P (\lambda gpv\ s. inline1' (gpv, s))$)
and $P (\lambda - -. return_pmf\ None)$
and $\bigwedge inline1'. \llbracket \bigwedge gpv\ s. ord_spmf (=) (inline1' gpv\ s) (inline1\ gpv\ s); P\ inline1' \rrbracket$
 $\implies P (\lambda gpv\ s. the_gpv\ gpv \ggg case_generat (\lambda x. return_spmf (Inl (x, s))) (\lambda out$
 $rpv. the_gpv (callee\ s\ out) \ggg case_generat (\lambda(x, y). inline1' (rpv\ x)\ y) (\lambda out\ rpv'.$
 $return_spmf (Inr (out, rpv', rpv))))$)
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 (=)))* ($\lambda inline1'$.
 $P (\lambda gpv\ s. inline1' (gpv, s))$)
and $P (\lambda - -. return_pmf\ None)$
and $\bigwedge inline1'.$
 $\llbracket \bigwedge gpv\ s. ord_spmf (=) (inline1' gpv\ s) (inline1\ gpv\ s);$
 $\bigwedge gpv\ s. ord_spmf (=) (inline1' gpv\ s) (the_gpv\ gpv \ggg case_generat (\lambda x.$
 $return_spmf (Inl (x, s))) (\lambda out\ rpv. the_gpv (callee\ s\ out) \ggg case_generat (\lambda(x, y).$
 $inline1' (rpv\ x)\ y) (\lambda out\ rpv'. return_spmf (Inr (out, rpv', rpv))))$);
 $P\ inline1' \rrbracket$
 $\implies P (\lambda gpv\ s. the_gpv\ gpv \ggg case_generat (\lambda x. return_spmf (Inl (x, s))) (\lambda out$
 $rpv. the_gpv (callee\ s\ out) \ggg case_generat (\lambda(x, y). inline1' (rpv\ x)\ y) (\lambda out\ rpv'.$
 $return_spmf (Inr (out, rpv', rpv))))$)
shows $P\ inline1$
 <proof>

Iterate *local.inline1* over all interactions. We'd like to use (\ggg) before the recursive call, but *primcorec* does not support this. So we emulate (\ggg) by effectively defining two mutually recursive functions (sum type in the argument) where the second is exactly (\ggg) specialised to call *inline* in the bind.

primcorec *inline-aux*

$:: ('a, 'call, 'ret)\ gpv \times 's + ('ret \Rightarrow ('a, 'call, 'ret)\ gpv) \times ('ret \times 's, 'call', 'ret')$

gpv
 $\Rightarrow ('a \times 's, 'call', 'ret')\ gpv$
where
 $\bigwedge state. the-gpv\ (inline-aux\ state) =$
 $(case\ state\ of\ Inl\ (c, s) \Rightarrow map-spmf\ (\lambda result.$
 $\quad case\ result\ of\ Inl\ (x, s) \Rightarrow Pure\ (x, s)$
 $\quad | Inr\ (out, oracle, rpv) \Rightarrow IO\ out\ (\lambda input. inline-aux\ (Inr\ (rpv, oracle\ input))))$
 $(inline1\ c\ s)$
 $| Inr\ (rpv, c) \Rightarrow$
 $\quad map-spmf\ (\lambda result.$
 $\quad\quad case\ result\ of\ Inl\ (Inl\ (x, s)) \Rightarrow Pure\ (x, s)$
 $\quad\quad | Inl\ (Inr\ (out, oracle, rpv)) \Rightarrow IO\ out\ (\lambda input. inline-aux\ (Inr\ (rpv, oracle$
 $\quad\quad input)))$
 $\quad\quad | Inr\ (out, c) \Rightarrow IO\ out\ (\lambda input. inline-aux\ (Inr\ (rpv, c\ input))))$
 $\quad (bind-spmf\ (the-gpv\ c)\ (\lambda generat. case\ generat\ of\ Pure\ (x, s') \Rightarrow (map-spmf\ Inl$
 $\quad (inline1\ (rpv\ x)\ s'))$
 $\quad\quad | IO\ out\ c \Rightarrow return-spmf\ (Inr\ (out, c)))$
 $\quad))$

declare $inline-aux.simps[simp\ del]$

definition $inline :: ('a, 'call, 'ret)\ gpv \Rightarrow 's \Rightarrow ('a \times 's, 'call', 'ret')\ gpv$
where $inline\ c\ s = inline-aux\ (Inl\ (c, s))$

lemma $inline-aux-Inr$:

$inline-aux\ (Inr\ (rpv, oracl)) = bind-gpv\ oracl\ (\lambda(x, s). inline\ (rpv\ x)\ s)$
 $\langle proof \rangle$

lemma $inline-sel$:

$the-gpv\ (inline\ c\ s) =$
 $map-spmf\ (\lambda result. case\ result\ of\ Inl\ xs \Rightarrow Pure\ xs$
 $\quad | Inr\ (out, oracle, rpv) \Rightarrow IO\ out\ (\lambda input. bind-gpv\ (oracle$
 $\quad input)\ (\lambda(x, s'). inline\ (rpv\ x)\ s'))\ (inline1\ c\ s)$
 $\langle proof \rangle$

lemma $inline1-Fail\ [simp]$: $inline1\ Fail\ s = return-pmf\ None$
 $\langle proof \rangle$

lemma $inline-Fail\ [simp]$: $inline\ Fail\ s = Fail$
 $\langle proof \rangle$

lemma $inline1-Done\ [simp]$: $inline1\ (Done\ x)\ s = return-spmf\ (Inl\ (x, s))$
 $\langle proof \rangle$

lemma $inline-Done\ [simp]$: $inline\ (Done\ x)\ s = Done\ (x, s)$
 $\langle proof \rangle$

lemma $inline1-lift-spmf\ [simp]$: $inline1\ (lift-spmf\ p)\ s = map-spmf\ (\lambda x. Inl\ (x,$
 $s))\ p$

$\langle \text{proof} \rangle$

lemma *inline-lift-spmf* [simp]: $\text{inline} (\text{lift-spmf } p) s = \text{lift-spmf} (\text{map-spmf } (\lambda x. (x, s)) p)$
 $\langle \text{proof} \rangle$

lemma *inline1-Pause*:

$\text{inline1} (\text{Pause out } c) s =$
 $\text{the-gpv} (\text{callee } s \text{ out}) \gg= (\lambda \text{react. case react of Pure } (x, s') \Rightarrow \text{inline1} (c \ x) \ s' \mid$
 $\text{IO out' } c' \Rightarrow \text{return-spmf} (\text{Inr} (\text{out}', c', c)))$
 $\langle \text{proof} \rangle$

lemma *inline-Pause* [simp]:

$\text{inline} (\text{Pause out } c) s = \text{callee } s \text{ out} \gg= (\lambda(x, s'). \text{inline} (c \ x) \ s')$
 $\langle \text{proof} \rangle$

lemma *inline1-bind-gpv*:

fixes $gpv \ f \ s$
defines [simp]: $\text{inline11} \equiv \text{inline1}$ **and** [simp]: $\text{inline12} \equiv \text{inline1}$ **and** [simp]:
 $\text{inline13} \equiv \text{inline1}$
shows $\text{inline11} (\text{bind-gpv } gpv \ f) s = \text{bind-spmf} (\text{inline12 } gpv \ s)$
 $(\lambda \text{res. case res of Inl } (x, s') \Rightarrow \text{inline13} (f \ x) \ s' \mid \text{Inr} (\text{out}, \text{rpv}', \text{rpv}) \Rightarrow$
 $\text{return-spmf} (\text{Inr} (\text{out}, \text{rpv}', \text{bind-rpv } \text{rpv} \ f)))$
(is $?lhs = ?rhs$
 $\langle \text{proof} \rangle$

lemma *inline-bind-gpv* [simp]:

$\text{inline} (\text{bind-gpv } gpv \ f) s = \text{bind-gpv} (\text{inline } gpv \ s) (\lambda(x, s'). \text{inline} (f \ x) \ s')$
 $\langle \text{proof} \rangle$

end

lemma *set-inline1-lift-spmf1*: $\text{set-spmf} (\text{inline1} (\lambda s \ x. \text{lift-spmf} (p \ s \ x)) \ gpv \ s) \subseteq$
 range Inl
 $\langle \text{proof} \rangle$

lemma *in-set-inline1-lift-spmf1*: $y \in \text{set-spmf} (\text{inline1} (\lambda s \ x. \text{lift-spmf} (p \ s \ x))$
 $gpv \ s) \implies \exists r \ s'. y = \text{Inl} (r, s')$
 $\langle \text{proof} \rangle$

lemma *inline-lift-spmf1*:

fixes p **defines** $\text{callee} \equiv \lambda s \ c. \text{lift-spmf} (p \ s \ c)$
shows $\text{inline } \text{callee } gpv \ s = \text{lift-spmf} (\text{map-spmf } \text{projl} (\text{inline1 } \text{callee } gpv \ s))$
 $\langle \text{proof} \rangle$

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

lemma *inline1-parametric'*:

$((S \implies C \implies \text{rel-gpv}'' (\text{rel-prod } R \ S) \ C' \ R') \implies \text{rel-gpv}'' A \ C \ R$
 $\implies S$

$\text{====> rel-spmf (rel-sum (rel-prod A S) (rel-prod C' (rel-prod (R' \text{====>} rel-gpv'' (rel-prod R S) C' R') (R \text{====>} rel-gpv'' A C R))))}$
 inline1 inline1
 $\text{(is (- \text{====>} ?R) - -)}$
 $\langle \text{proof} \rangle$

lemma *inline1-parametric* [*transfer-rule*]:
 $\text{((S \text{====>} C \text{====>} rel-gpv (rel-prod (=) S) C') \text{====>} rel-gpv A C \text{====>} S \text{====>} rel-spmf (rel-sum (rel-prod A S) (rel-prod C' (rel-prod (rel-rpv (rel-prod (=) S) C') (rel-rpv A C))))}$
 inline1 inline1
 $\langle \text{proof} \rangle$

lemma *inline-parametric'*:
notes [*transfer-rule*] = *inline1-parametric' the-gpv-parametric' corec-gpv-parametric'*
shows $\text{((S \text{====>} C \text{====>} rel-gpv'' (rel-prod R S) C' R') \text{====>} rel-gpv'' A C R \text{====>} S \text{====>} rel-gpv'' (rel-prod A S) C' R')}$
 inline inline
 $\langle \text{proof} \rangle$

lemma *inline-parametric* [*transfer-rule*]:
 $\text{((S \text{====>} C \text{====>} rel-gpv (rel-prod (=) S) C') \text{====>} rel-gpv A C \text{====>} S \text{====>} rel-gpv (rel-prod A S) C')}$
 inline inline
 $\langle \text{proof} \rangle$
end

Associativity rule for *inline*

context
fixes *callee1* :: 's1 \Rightarrow 'c1 \Rightarrow ('r1 \times 's1, 'c, 'r) *gpv*
and *callee2* :: 's2 \Rightarrow 'c2 \Rightarrow ('r2 \times 's2, 'c1, 'r1) *gpv*
begin

partial-function (*spmf*) *inline2* :: ('a, 'c2, 'r2) *gpv* \Rightarrow 's2 \Rightarrow 's1
 \Rightarrow ('a \times ('s2 \times 's1) + 'c \times ('r1 \times 's1, 'c, 'r) *rpv* \times ('r2 \times 's2, 'c1, 'r1) *rpv* \times ('a, 'c2, 'r2) *rpv*) *spmf*

where
 $\text{inline2 gpv s2 s1 =}$
 $\text{bind-spmf (the-gpv gpv)}$
 $\text{(case-generat (\lambda x. return-spmf (Inl (x, s2, s1)))}$
 $\text{(\lambda out rpv. bind-spmf (inline1 callee1 (callee2 s2 out) s1)}$
 $\text{(case-sum (\lambda((r2, s2), s1). inline2 (rpv r2) s2 s1)}$
 $\text{(\lambda(x, rpv'', rpv'). return-spmf (Inr (x, rpv'', rpv'))))})}$

lemma *inline2-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) ($\lambda \text{inline2. } P (\lambda \text{gpv s2 s1. inline2 ((gpv, s2), s1))$)
and $P (\lambda - -. \text{return-pmf None})$
and $\bigwedge \text{inline2'. } P \text{ inline2' } \Longrightarrow$

$P (\lambda gpv\ s2\ s1.\ bind\text{-}spmf\ (the\text{-}gpv\ gpv)\ (\lambda generat.\ case\ generat\ of$
 $\quad Pure\ x \Rightarrow return\text{-}spmf\ (Inl\ (x,\ s2,\ s1))$
 $\quad | IO\ out\ rpv \Rightarrow bind\text{-}spmf\ (inline1\ callee1\ (callee2\ s2\ out)\ s1)\ (\lambda lr.\ case\ lr$
of
 $\quad Inl\ ((r2,\ s2),\ c) \Rightarrow inline2'\ (rpv\ r2)\ s2\ c$
 $\quad | Inr\ (x,\ rpv'',\ rpv') \Rightarrow return\text{-}spmf\ (Inr\ (x,\ rpv'',\ rpv',\ rpv))))$
shows $P\ inline2$
 $\langle proof \rangle$

lemma *inline1-inline-conv-inline2*:
fixes $gpv' :: ('r2 \times 's2, 'c1, 'r1)\ gpv$
shows $inline1\ callee1\ (inline\ callee2\ gpv\ s2)\ s1 =$
 $map\text{-}spmf\ (map\text{-}sum\ (\lambda(x,\ (s2,\ s1)).\ ((x,\ s2),\ s1))$
 $\quad (\lambda(x,\ rpv'',\ rpv',\ rpv).\ (x,\ rpv'',\ \lambda r1.\ rpv'\ r1 \gg= (\lambda(r2,\ s2).\ inline\ callee2\ (rpv$
 $\quad r2)\ s2))))$
 $(inline2\ gpv\ s2\ s1)$
(is $?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *inline1-inline-conv-inline2'*:
 $inline1\ (\lambda(s2,\ s1)\ c2.\ map\text{-}gpv\ (\lambda((r,\ s2),\ s1).\ (r,\ s2,\ s1))\ id\ (inline\ callee1$
 $(callee2\ s2\ c2)\ s1))\ gpv\ (s2,\ s1) =$
 $map\text{-}spmf\ (map\text{-}sum\ id\ (\lambda(x,\ rpv'',\ rpv',\ rpv).\ (x,\ \lambda r.\ bind\text{-}gpv\ (rpv''\ r)$
 $\quad (\lambda(r1,\ s1).\ map\text{-}gpv\ (\lambda((r2,\ s2),\ s1).\ (r2,\ s2,\ s1))\ id\ (inline\ callee1\ (rpv'$
 $\quad r1)\ s1)),\ rpv)))$
 $(inline2\ gpv\ s2\ s1)$
(is $?lhs = ?rhs)$
 $\langle proof \rangle$

lemma *inline-assoc*:
 $inline\ callee1\ (inline\ callee2\ gpv\ s2)\ s1 =$
 $map\text{-}gpv\ (\lambda(r,\ s2,\ s1).\ ((r,\ s2),\ s1))\ id\ (inline\ (\lambda(s2,\ s1)\ c2.\ map\text{-}gpv\ (\lambda((r,$
 $s2),\ s1).\ (r,\ s2,\ s1))\ id\ (inline\ callee1\ (callee2\ s2\ c2)\ s1))\ gpv\ (s2,\ s1))$
 $\langle proof \rangle$

end

lemma *set-inline2-lift-spmf1*: $set\text{-}spmf\ (inline2\ (\lambda s\ x.\ lift\text{-}spmf\ (p\ s\ x))\ callee\ gpv$
 $s\ s') \subseteq range\ Inl$
 $\langle proof \rangle$

lemma *in-set-inline2-lift-spmf1*: $y \in set\text{-}spmf\ (inline2\ (\lambda s\ x.\ lift\text{-}spmf\ (p\ s\ x))$
 $callee\ gpv\ s\ s') \implies \exists r\ s\ s'. y = Inl\ (r,\ s,\ s')$
 $\langle proof \rangle$

context

fixes $consider' :: 'call \Rightarrow bool$
and $consider :: 'call' \Rightarrow bool$
and $callee :: 's \Rightarrow 'call \Rightarrow ('ret \times 's,\ 'call', 'ret')\ gpv$

notes $[[\text{function-internals}]]$
begin

private partial-function (*spmf*) *inline1'*
 $:: ('a, 'call, 'ret) \text{ gpv} \Rightarrow 's$
 $\Rightarrow ('a \times 's + 'call \times 'call' \times ('ret \times 's, 'call', 'ret') \text{ rpv} \times ('a, 'call, 'ret) \text{ rpv})$
spmf
where
inline1' gpv s =
the-gpv gpv \ggg
case-generat ($\lambda x. \text{return-spmf (Inl (x, s))}$)
 $(\lambda \text{out rpv. the-gpv (callee s out)} \ggg$
 $\text{case-generat } (\lambda(x, y). \text{inline1'} (\text{rpv } x) \text{ } y)$
 $(\lambda \text{out}' \text{ rpv}'. \text{return-spmf (Inr (out, out}', \text{ rpv}', \text{ rpv}))))$)

private lemma *inline1'-fixp-induct* [*case-names adm bottom step*]:
assumes *ccpo.admissible (fun-lub lub-spmf) (fun-ord (ord-spmf (=)))* ($\lambda \text{inline1}'.$
 $P (\lambda \text{gpv } s. \text{inline1}' (\text{gpv}, s))$)
and $P (\lambda -. \text{return-pmf None})$
and $\bigwedge \text{inline1}'. P \text{ inline1}' \Longrightarrow P (\lambda \text{gpv } s. \text{the-gpv gpv} \ggg \text{case-generat } (\lambda x.$
 $\text{return-spmf (Inl (x, s))} (\lambda \text{out rpv. the-gpv (callee s out)} \ggg \text{case-generat } (\lambda(x,$
 $y). \text{inline1}' (\text{rpv } x) \text{ } y) (\lambda \text{out}' \text{ rpv}'. \text{return-spmf (Inr (out, out}', \text{ rpv}', \text{ rpv}))))$)
shows $P \text{ inline1}'$
 $\langle \text{proof} \rangle$ **lemma** *inline1-conv-inline1'*: *inline1 callee gpv s = map-spmf (map-sum*
 $\text{id snd) (inline1' gpv s)$
 $\langle \text{proof} \rangle$

context
fixes $q :: \text{enat}$
assumes $q: \bigwedge s x. \text{consider}' x \Longrightarrow \text{interaction-bound consider (callee s } x) \leq q$
and $\text{ignore}: \bigwedge s x. \neg \text{consider}' x \Longrightarrow \text{interaction-bound consider (callee s } x) = 0$
begin

private lemma *interaction-bound-inline1'-aux*:
 $\text{interaction-bound consider}' \text{ gpv} \leq p$
 $\Longrightarrow \text{set-spmf (inline1' gpv s)} \subseteq \{\text{Inr (out}', \text{ out, } c', \text{ rpv)} \mid \text{out}' \text{ out } c' \text{ rpv.}$
 $\text{if consider}' \text{ out}'$
 $\text{then } (\forall \text{input. (if consider out then eSuc (interaction-bound consider (c'}$
 $\text{input})) \text{ else interaction-bound consider (c' input))} \leq q) \wedge$
 $(\forall x. \text{eSuc (interaction-bound consider}' (\text{rpv } x))} \leq p)$
 $\text{else } \neg \text{consider out} \wedge (\forall \text{input. interaction-bound consider (c' input)} = 0) \wedge$
 $(\forall x. \text{interaction-bound consider}' (\text{rpv } x) \leq p)\}$
 $\cup \text{range Inl}$
 $\langle \text{proof} \rangle$

lemma *interaction-bound-inline1'*:
 $\llbracket \text{Inr (out}', \text{ out, } c', \text{ rpv)} \in \text{set-spmf (inline1' gpv s); interaction-bound consider}'$
 $\text{gpv} \leq p \rrbracket$
 $\Longrightarrow \text{if consider}' \text{ out}' \text{ then}$

(if consider out then eSuc (interaction-bound consider (c' input)) else interaction-bound consider (c' input)) $\leq q \wedge$
 eSuc (interaction-bound consider' (rpv x)) $\leq p$
 else \neg consider out \wedge interaction-bound consider (c' input) = 0 \wedge interaction-bound consider' (rpv x) $\leq p$
 ⟨proof⟩

end

lemma *interaction-bounded-by-inline1*:

[[Inr (out', out, c', rpv) \in set-spmf (inline1' gpv s);
 interaction-bounded-by consider' gpv p;
 $\bigwedge s x$. consider' x \implies interaction-bounded-by consider (callee s x) q;
 $\bigwedge s x$. \neg consider' x \implies interaction-bounded-by consider (callee s x) 0]]
 \implies if consider' out' then
 (if consider out then $q \neq 0 \wedge$ interaction-bounded-by consider (c' input) (q - 1) else interaction-bounded-by consider (c' input) q) \wedge
 $p \neq 0 \wedge$ interaction-bounded-by consider' (rpv x) (p - 1)
 else \neg consider out \wedge interaction-bounded-by consider (c' input) 0 \wedge 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: $\bigwedge s x$. consider' x \implies interaction-bounded-by consider (callee s x) q
and ignore: $\bigwedge s x$. \neg consider' x \implies 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 \implies bool
and consider :: 'call' \implies bool
and callee :: 's \implies 'call \implies ('ret \times 's, 'call', 'ret') gpv
and gpv :: ('a, 'call, 'ret) gpv
assumes p: interaction-bounded-by consider' gpv p
and q: $\bigwedge s x$. [[I s; consider' x] \implies interaction-bounded-by consider (callee s x) q
and ignore: $\bigwedge s x$. [[I s; \neg consider' x] \implies interaction-bounded-by consider (callee s x) 0
and I: I s
and invariant: $\bigwedge s x y s'$. [[(y, s') \in results'-gpv (callee s x); I s] \implies 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 $callee :: 's \Rightarrow 'call \Rightarrow ('ret \times 's, 'call', 'ret') gpv$
assumes $results: \bigwedge s x. x \in outs\text{-}\mathcal{I} \mathcal{I} \Longrightarrow results\text{-}gpv \mathcal{I}' (callee\ s\ x) \subseteq responses\text{-}\mathcal{I} \mathcal{I} \times UNIV$
begin

lemma *inline1-in-sub-gpvs-callee*:
assumes $Inr (out, callee', rpv') \in set\text{-}spmf (inline1\ callee\ gpv\ s)$
and $WT: \mathcal{I} \vdash g\ gpv\ \checkmark$
shows $\exists call \in outs\text{-}\mathcal{I} \mathcal{I}. \exists s. \forall x \in responses\text{-}\mathcal{I} \mathcal{I}'\ out. callee'\ x \in sub\text{-}gpvs \mathcal{I}' (callee\ s\ call)$
 $\langle proof \rangle$

lemma *inline1-in-sub-gpvs*:
assumes $Inr (out, callee', rpv') \in set\text{-}spmf (inline1\ callee\ gpv\ s)$
and $(x, s') \in results\text{-}gpv \mathcal{I}' (callee'\ input)$
and $input \in responses\text{-}\mathcal{I} \mathcal{I}'\ out$
and $\mathcal{I} \vdash g\ gpv\ \checkmark$
shows $rpv'\ x \in sub\text{-}gpvs \mathcal{I} gpv$
 $\langle proof \rangle$

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

lemma *WT-gpv-inline1*:
assumes $Inr (out, rpv, rpv') \in set\text{-}spmf (inline1\ callee\ gpv\ s)$
and $\mathcal{I} \vdash g\ gpv\ \checkmark$
shows $out \in outs\text{-}\mathcal{I} \mathcal{I}'$ (**is** *?thesis1*)
and $input \in responses\text{-}\mathcal{I} \mathcal{I}'\ out \Longrightarrow \mathcal{I}' \vdash g\ rpv\ input\ \checkmark$ (**is** *PROP ?thesis2*)
and $\llbracket input \in responses\text{-}\mathcal{I} \mathcal{I}'\ out; (x, s') \in results\text{-}gpv \mathcal{I}' (rpv\ input) \rrbracket \Longrightarrow \mathcal{I} \vdash g\ rpv'\ x\ \checkmark$ (**is** *PROP ?thesis3*)
 $\langle proof \rangle$

lemma *WT-gpv-inline*:
assumes $\mathcal{I} \vdash g\ gpv\ \checkmark$
shows $\mathcal{I}' \vdash g\ inline\ callee\ gpv\ s\ \checkmark$
 $\langle proof \rangle$

end

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

lemma *lossless-spmf-inline1*:
assumes *lossless*: $\bigwedge s x. x \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{lossless-spmf} (\text{the-gpv} (\text{callee } s \ x))$
shows *lossless-spmf* (*inline1 callee gpv s*)
 $\langle \text{proof} \rangle$

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

lemma *lossless-results-inline1*:
assumes *Inr* (*out*, *rpv*, *rpv'*) $\in \text{set-spmf} (\text{inline1 callee gpv } s)$
and (*x*, *s'*) $\in \text{results-gpv } \mathcal{I}' (\text{rpv input})$
and *input* $\in \text{responses-}\mathcal{I} \ \mathcal{I}' \ \text{out}$
shows *lossless-gpv* \mathcal{I} (*rpv' x*)
 $\langle \text{proof} \rangle$

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* $\mathcal{I} \ \text{gpv } \mathcal{I} \vdash_g \ \text{gpv} \ \checkmark$
and *lossless*: $\bigwedge s x. x \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{lossless-gpv } \mathcal{I}' (\text{callee } s \ x)$
shows *lossless-gpv* \mathcal{I}' (*inline callee gpv s*)
 $\langle \text{proof} \rangle$

end

definition *id-oracle* :: 's \Rightarrow 'call \Rightarrow ('ret \times 's, 'call, 'ret) gpv
where *id-oracle* *s x* = *Pause x* ($\lambda x. \text{Done } (x, s)$)

lemma *inline1-id-oracle*:
inline1 id-oracle gpv s =
map-spmf ($\lambda \text{generat. case generat of Pure } x \Rightarrow \text{Inl } (x, s) \mid \text{IO out } c \Rightarrow \text{Inr } (\text{out}, \lambda x. \text{Done } (x, s), c)$) (*the-gpv gpv*)
 $\langle \text{proof} \rangle$

lemma *inline-id-oracle* [*simp*]: *inline id-oracle gpv s* = *map-gpv* ($\lambda x. (x, s)$) *id gpv*
 $\langle \text{proof} \rangle$

locale *raw-converter-invariant* =
fixes \mathcal{I} :: ('call, 'ret) \mathcal{I}
and \mathcal{I}' :: ('call', 'ret') \mathcal{I}
and *callee* :: 's \Rightarrow 'call \Rightarrow ('ret \times 's, 'call', 'ret') gpv

and $I :: 's \Rightarrow \text{bool}$
assumes $\text{results-callee}: \bigwedge s x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I \ s \rrbracket \Longrightarrow \text{results-gpv} \ \mathcal{I}' \ (\text{callee} \ s \ x)$
 $\subseteq \text{responses-}\mathcal{I} \ \mathcal{I} \ x \times \{s. I \ s\}$
and $\text{WT-callee}: \bigwedge x s. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I \ s \rrbracket \Longrightarrow \mathcal{I}' \vdash_g \text{callee} \ s \ x \ \checkmark$
begin

context begin

private lemma aux:

$\text{set-spmf} \ (\text{inline1} \ \text{callee} \ \text{gpv} \ s) \subseteq \{ \text{Inr} \ (out, \text{callee}', \text{rpv}') \mid out \ \text{callee}' \ \text{rpv}'.$
 $\exists \text{call} \in \text{outs-}\mathcal{I} \ \mathcal{I}. \exists s. I \ s \wedge (\forall x \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ out. \text{callee}' \ x \in \text{sub-gpvs} \ \mathcal{I}'$
 $(\text{callee} \ s \ \text{call})) \} \cup$
 $\{ \text{Inl} \ (x, s') \mid x \ s'. x \in \text{results-gpv} \ \mathcal{I} \ \text{gpv} \wedge I \ s' \}$
(is $?concl \ (\text{inline1} \ \text{callee}) \ \text{gpv} \ s$ **is** $- \subseteq ?rhs1 \cup ?rhs2 \ \text{gpv}$)
if $\mathcal{I} \vdash_g \ \text{gpv} \ \checkmark \ I \ s$
 $\langle \text{proof} \rangle$

lemma inline1-in-sub-gpvs-callee:

assumes $\text{Inr} \ (out, \text{callee}', \text{rpv}') \in \text{set-spmf} \ (\text{inline1} \ \text{callee} \ \text{gpv} \ s)$
and $\text{WT}: \mathcal{I} \vdash_g \ \text{gpv} \ \checkmark$
and $s: I \ s$
shows $\exists \text{call} \in \text{outs-}\mathcal{I} \ \mathcal{I}. \exists s. I \ s \wedge (\forall x \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ out. \text{callee}' \ x \in \text{sub-gpvs}$
 $\mathcal{I}' \ (\text{callee} \ s \ \text{call}))$
 $\langle \text{proof} \rangle$

lemma inline1-Inl-results-gpv:

assumes $\text{Inl} \ (x, s') \in \text{set-spmf} \ (\text{inline1} \ \text{callee} \ \text{gpv} \ s)$
and $\text{WT}: \mathcal{I} \vdash_g \ \text{gpv} \ \checkmark$
and $s: I \ s$
shows $x \in \text{results-gpv} \ \mathcal{I} \ \text{gpv} \wedge I \ s'$
 $\langle \text{proof} \rangle$

end

lemma inline1-in-sub-gpvs:

assumes $\text{Inr} \ (out, \text{callee}', \text{rpv}') \in \text{set-spmf} \ (\text{inline1} \ \text{callee} \ \text{gpv} \ s)$
and $(x, s') \in \text{results-gpv} \ \mathcal{I}' \ (\text{callee}' \ \text{input})$
and $\text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ out$
and $\mathcal{I} \vdash_g \ \text{gpv} \ \checkmark$
and $I \ s$
shows $\text{rpv}' \ x \in \text{sub-gpvs} \ \mathcal{I} \ \text{gpv} \wedge I \ s'$
 $\langle \text{proof} \rangle$

lemma WT-gpv-inline1:

assumes $\text{Inr} \ (out, \text{rpv}, \text{rpv}') \in \text{set-spmf} \ (\text{inline1} \ \text{callee} \ \text{gpv} \ s)$
and $\mathcal{I} \vdash_g \ \text{gpv} \ \checkmark$
and $I \ s$
shows $out \in \text{outs-}\mathcal{I} \ \mathcal{I}'$ **(is** $?thesis1$)
and $\text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ out \Longrightarrow \mathcal{I}' \vdash_g \ \text{rpv} \ \text{input} \ \checkmark$ **(is** $\text{PROP} \ ?thesis2$)
and $\llbracket \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I}' \ out; (x, s') \in \text{results-gpv} \ \mathcal{I}' \ (\text{rpv} \ \text{input}) \rrbracket \Longrightarrow \mathcal{I}$
 $\vdash_g \ \text{rpv}' \ x \ \checkmark \wedge I \ s'$ **(is** $\text{PROP} \ ?thesis3$)

<proof>

lemma *WT-gpv-inline-invar*:

assumes $\mathcal{I} \vdash g \text{ gpv } \checkmark$

and $I \ s$

shows $\mathcal{I}' \vdash g \text{ inline callee gpv } s \checkmark$

<proof>

end

lemma *WT-gpv-inline'*:

assumes $\bigwedge s \ x. x \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{results-gpv } \mathcal{I}' \ (\text{callee } s \ x) \subseteq \text{responses-}\mathcal{I} \ \mathcal{I} \ x \times \text{UNIV}$

and $\bigwedge x \ s. x \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \mathcal{I}' \vdash g \text{ callee } s \ x \checkmark$

and $\mathcal{I} \vdash g \text{ gpv } \checkmark$

shows $\mathcal{I}' \vdash g \text{ inline callee gpv } s \checkmark$

<proof>

lemma *results-gpv-sub-gvps*: $\text{gpv}' \in \text{sub-gvps } \mathcal{I} \ \text{gpv} \implies \text{results-gpv } \mathcal{I} \ \text{gpv}' \subseteq \text{results-gpv } \mathcal{I} \ \text{gpv}$

<proof>

lemma *in-results-gpv-sub-gvps*: $\llbracket x \in \text{results-gpv } \mathcal{I} \ \text{gpv}'; \text{gpv}' \in \text{sub-gvps } \mathcal{I} \ \text{gpv} \rrbracket \implies x \in \text{results-gpv } \mathcal{I} \ \text{gpv}$

<proof>

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

lemma *results-gpv-inline-aux*:

assumes $(x, s') \in \text{results-gpv } \mathcal{I}' \ (\text{inline-aux callee } y)$

shows $\llbracket y = \text{Inl } (\text{gpv}, s); \mathcal{I} \vdash g \text{ gpv } \checkmark; I \ s \rrbracket \implies x \in \text{results-gpv } \mathcal{I} \ \text{gpv} \wedge I \ s'$

and $\llbracket y = \text{Inr } (\text{rpv}, \text{callee}'); \forall (z, s') \in \text{results-gpv } \mathcal{I}' \ \text{callee}'. \mathcal{I} \vdash g \ \text{rpv } z \checkmark \wedge I \ s' \rrbracket$

$\implies \exists (z, s'') \in \text{results-gpv } \mathcal{I}' \ \text{callee}'. x \in \text{results-gpv } \mathcal{I} \ (\text{rpv } z) \wedge I \ s'' \wedge I \ s'$

<proof>

lemma *results-gpv-inline*:

$\llbracket (x, s') \in \text{results-gpv } \mathcal{I}' \ (\text{inline callee gpv } s); \mathcal{I} \vdash g \text{ gpv } \checkmark; I \ s \rrbracket \implies x \in \text{results-gpv } \mathcal{I} \ \text{gpv} \wedge I \ s'$

<proof>

end

lemma *inline-map-gpv*:

$\text{inline callee } (\text{map-gpv } f \ g \ \text{gpv}) \ s = \text{map-gpv } (\text{apfst } f) \ \text{id } (\text{inline } (\lambda s \ x. \text{callee } s \ (g \ x)) \ \text{gpv } s)$

<proof>

4.17 Running GPVs

type-synonym ('call, 'ret, 's) callee = 's \Rightarrow 'call \Rightarrow ('ret \times 's) spmf

context fixes callee :: ('call, 'ret, 's) callee **notes** [[function-internals]] **begin**

partial-function (spmf) exec-gpv :: ('a, 'call, 'ret) gpv \Rightarrow 's \Rightarrow ('a \times 's) spmf
where

exec-gpv c s =
 the-gpv c \ggg
 case-generat (λx . return-spmf (x, s))
 (λout c. callee s out \ggg ($\lambda(x, y)$. exec-gpv (c x) y))

abbreviation run-gpv :: ('a, 'call, 'ret) gpv \Rightarrow 's \Rightarrow 'a spmf

where run-gpv gpv s \equiv map-spmf fst (exec-gpv gpv s)

lemma exec-gpv-fixp-induct [case-names adm bottom step]:

assumes ccpo.admissible (fun-lub lub-spmf) (fun-ord (ord-spmf (=))) (λf . P (λc s. f (c, s)))
and P (λ - -. return-pmf None)
and \bigwedge exec-gpv. P exec-gpv \implies
 P (λc s. the-gpv c \ggg case-generat (λx . return-spmf (x, s)) (λout c. callee s out \ggg ($\lambda(x, y)$. 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 (=))) (λf . P (λc s. f (c, s)))
and P (λ - -. return-pmf None)
and \bigwedge exec-gpv'. $\llbracket \bigwedge c$ s. ord-spmf (=) (exec-gpv' c s) (exec-gpv c s); P exec-gpv' \rrbracket
 \implies P (λc s. the-gpv c \ggg case-generat (λx . return-spmf (x, s)) (λout c. callee s out \ggg ($\lambda(x, y)$. 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 (=))) (λf . P (λc s. f (c, s)))
and P (λ - -. return-pmf None)
and \bigwedge exec-gpv'.
 $\llbracket \bigwedge c$ s. ord-spmf (=) (exec-gpv' c s) (exec-gpv c s);
 $\bigwedge c$ s. ord-spmf (=) (exec-gpv' c s) (the-gpv c \ggg case-generat (λx . return-spmf (x, s)) (λout c. callee s out \ggg ($\lambda(x, y)$. exec-gpv' (c x) y)));
 P exec-gpv' \rrbracket
 \implies P (λc s. the-gpv c \ggg case-generat (λx . return-spmf (x, s)) (λout c. callee s out \ggg ($\lambda(x, y)$. 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-distrib]*:

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*: $exec\text{-}gpv\ callee\ (c \gg= f)\ s = exec\text{-}gpv\ callee\ c\ s \gg= (\lambda(x, s') \Rightarrow exec\text{-}gpv\ callee\ (f\ x)\ s')$
 $\langle proof \rangle$

lemma *exec-gpv-map-gpv-id*:
 $exec\text{-}gpv\ oracle\ (map\text{-}gpv\ f\ id\ gpv)\ \sigma = map\text{-}spmf\ (apfst\ f)\ (exec\text{-}gpv\ oracle\ gpv\ \sigma)$
 $\langle proof \rangle$

lemma *exec-gpv-Pause* [simp]:
 $exec\text{-}gpv\ callee\ (Pause\ out\ f)\ s = callee\ s\ out \gg= (\lambda(x, s'). exec\text{-}gpv\ callee\ (f\ x)\ s')$
 $\langle proof \rangle$

lemma *exec-gpv-bind-lift-spmf*:
 $exec\text{-}gpv\ callee\ (bind\text{-}gpv\ (lift\text{-}spmf\ p)\ f)\ s = bind\text{-}spmf\ p\ (\lambda x. exec\text{-}gpv\ callee\ (f\ x)\ s)$
 $\langle proof \rangle$

lemma *exec-gpv-bind-option* [simp]:
 $exec\text{-}gpv\ oracle\ (monad.\text{bind-option}\ Fail\ x\ f)\ s = monad.\text{bind-option}\ (return\text{-}pmf\ None)\ x\ (\lambda a. exec\text{-}gpv\ oracle\ (f\ a)\ s)$
 $\langle proof \rangle$

lemma *pred-spmf-exec-gpv*:
— We don't get an equivalence here because states are threaded through in *exec-gpv*.
 $\llbracket pred\text{-}gpv\ A\ C\ gpv; pred\text{-}fun\ S\ (pred\text{-}fun\ C\ (pred\text{-}spmf\ (pred\text{-}prod\ (\lambda_. True)\ S))\ callee; S\ s \rrbracket$
 $\implies pred\text{-}spmf\ (pred\text{-}prod\ A\ S)\ (exec\text{-}gpv\ callee\ gpv\ s)$
 $\langle proof \rangle$

lemma *exec-gpv-inline*:
fixes $callee :: ('c, 'r, 's)\ callee$
and $gpv :: 's' \Rightarrow 'c' \Rightarrow ('r' \times 's', 'c, 'r)\ gpv$
shows $exec\text{-}gpv\ callee\ (inline\ gpv\ c'\ s')\ s =$
 $map\text{-}spmf\ (\lambda(x, s', s). ((x, s'), s))\ (exec\text{-}gpv\ (\lambda(s', s)\ y. map\text{-}spmf\ (\lambda((x, s'), s). (x, s', s))\ (exec\text{-}gpv\ callee\ (gpv\ s'\ y)\ s))\ c'\ (s', s))$
(is $?lhs = ?rhs$ **)
 $\langle proof \rangle$**

lemma *ord-spmf-exec-gpv*:
assumes $callee: \bigwedge s\ x. ord\text{-}spmf\ (=)\ (callee1\ s\ x)\ (callee2\ s\ x)$
shows $ord\text{-}spmf\ (=)\ (exec\text{-}gpv\ callee1\ gpv\ s)\ (exec\text{-}gpv\ callee2\ gpv\ s)$
 $\langle proof \rangle$

context **fixes** $callee :: ('call, 'ret, 's)\ callee$ **notes** $[[function\text{-}internals]]$ **begin**

partial-function (*spmf*) *excep-resumption* :: ('a, 'call, 'ret) *resumption* \Rightarrow 's \Rightarrow ('a \times 's) *spmf*

where

excep-resumption *r s* = (case *r* of *resumption.Done* *x* \Rightarrow *return-pmf* (*map-option* ($\lambda a. (a, s)$) *x*)
| *resumption.Pause* *out c* \Rightarrow *bind-spmf* (*callee* *s out*) ($\lambda(input, s').$ *excep-resumption* (*c input*) *s'*))

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

lemma *excep-resumption-ABORT* [*simp*]: *excep-resumption ABORT s* = *return-pmf None*
<proof>

lemma *excep-resumption-DONE* [*simp*]: *excep-resumption (DONE x) s* = *return-spmf (x, s)*
<proof>

lemma *exec-gpv-lift-resumption*: *exec-gpv callee (lift-resumption r) s* = *excep-resumption r s*
<proof>

lemma *mcont2mcont-excep-resumption* [*THEN* *spmf.mcont2mcont, cont-intro, simp*]:
shows *mcont-excep-resumption*:
mcont resumption-lub resumption-ord lub-spmf (ord-spmf (=)) ($\lambda r. \textit{excep-resumption r s}$)
<proof>

lemma *excep-resumption-bind* [*simp*]:
excep-resumption (r \ggg f) s = *excep-resumption r s \ggg ($\lambda(x, s'). \textit{excep-resumption (f x) s'}$)*
<proof>

lemma *pred-spmf-excep-resumption*:
 $\bigwedge A. \llbracket \textit{pred-resumption A C r; pred-fun S (pred-fun C (pred-spmf (pred-prod (\lambda-. True) S))) callee; S s} \rrbracket$
 $\implies \textit{pred-spmf (pred-prod A S) (excep-resumption r s)}$
<proof>

end

inductive *WT-callee* :: ('call, 'ret) *I* \Rightarrow ('call \Rightarrow ('ret \times 's) *spmf*) \Rightarrow *bool* ($\iota(-)$)
 $\vdash c/ (-) \surd [100, 0] 99$

for *I callee*

where

WT-callee:

$\llbracket \bigwedge call \textit{ret s.} \llbracket call \in \textit{outs-I I; (ret, s) \in set-spmf (callee call)} \rrbracket \implies \textit{ret} \in \textit{responses-I I call} \rrbracket$

$\implies \mathcal{I} \vdash c \text{ callee } \checkmark$

lemmas $WT\text{-callee}I = WT\text{-callee}$

hide-fact $WT\text{-callee}$

lemma $WT\text{-callee}D$: $\llbracket \mathcal{I} \vdash c \text{ callee } \checkmark; (ret, s) \in \text{set-spmf } (callee \text{ out}); out \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies ret \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out}$

$\langle \text{proof} \rangle$

lemma $WT\text{-callee-full}$ $[intro!, simp]$: $\mathcal{I}\text{-full} \vdash c \text{ callee } \checkmark$

$\langle \text{proof} \rangle$

lemma $WT\text{-callee-parametric}$ $[transfer\text{-rule}]$:

includes $lifting\text{-syntax}$

assumes $[transfer\text{-rule}]$: $bi\text{-unique } R$

shows $(rel\text{-}\mathcal{I} \ C \ R \implies (C \implies rel\text{-spmfs } (rel\text{-prod } R \ S)) \implies (=))$

$WT\text{-callee } WT\text{-callee}$

$\langle \text{proof} \rangle$

locale $callee\text{-invariant-on-base} =$

fixes $callee :: 's \Rightarrow 'a \Rightarrow ('b \times 's) \text{ spmf}$

and $I :: 's \Rightarrow \text{bool}$

and $\mathcal{I} :: ('a, 'b) \mathcal{I}$

locale $callee\text{-invariant-on} = callee\text{-invariant-on-base } callee \ I \ \mathcal{I}$

for $callee :: 's \Rightarrow 'a \Rightarrow ('b \times 's) \text{ spmf}$

and $I :: 's \Rightarrow \text{bool}$

and $\mathcal{I} :: ('a, 'b) \mathcal{I}$

+

assumes $callee\text{-invariant}$: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (callee \ s \ x); I \ s; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies I \ s'$

and $WT\text{-callee}$: $\bigwedge s. I \ s \implies \mathcal{I} \vdash c \text{ callee } s \ \checkmark$

begin

lemma $callee\text{-invariant}'$: $\llbracket (y, s') \in \text{set-spmf } (callee \ s \ x); I \ s; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies I \ s' \wedge y \in \text{responses-}\mathcal{I} \mathcal{I} \ x$

$\langle \text{proof} \rangle$

lemma $exec\text{-gpv-invariant}'$:

$\llbracket I \ s; \mathcal{I} \vdash g \text{ gpv } \checkmark \rrbracket \implies \text{set-spmf } (exec\text{-gpv } callee \ gpv \ s) \subseteq \{(x, s'). I \ s'\}$

$\langle \text{proof} \rangle$

lemma $exec\text{-gpv-invariant}$:

$\llbracket (x, s') \in \text{set-spmf } (exec\text{-gpv } callee \ gpv \ s); I \ s; \mathcal{I} \vdash g \text{ gpv } \checkmark \rrbracket \implies I \ s'$

$\langle \text{proof} \rangle$

lemma $interaction\text{-bounded-by-exec-gpv-count}'$:

fixes $count$

assumes $bound$: $interaction\text{-bounded-by consider gpv } n$

and *count*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf} (\text{callee } s x); I s; \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{eSuc} (\text{count } s)$
and *ignore*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf} (\text{callee } s x); I s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and *WT*: $\mathcal{I} \vdash_g \text{gpv} \checkmark$
and *I*: $I s$
shows $\text{set-spmf} (\text{exec-gpv callee gpv } s) \subseteq \{(x, s'). \text{count } s' \leq n + \text{count } s\}$
<proof>

lemma *interaction-bounded-by-exec-gpv-count*:

fixes *count*
assumes *bound*: *interaction-bounded-by consider gpv n*
and *xs'*: $(x, s') \in \text{set-spmf} (\text{exec-gpv callee gpv } s)$
and *count*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf} (\text{callee } s x); I s; \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{eSuc} (\text{count } s)$
and *ignore*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf} (\text{callee } s x); I s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and *WT*: $\mathcal{I} \vdash_g \text{gpv} \checkmark$
and *I*: $I s$
shows $\text{count } s' \leq n + \text{count } s$
<proof>

lemma *interaction-bounded-by'-exec-gpv-count*:

fixes *count*
assumes *bound*: *interaction-bounded-by' consider gpv n*
and *xs'*: $(x, s') \in \text{set-spmf} (\text{exec-gpv callee gpv } s)$
and *count*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf} (\text{callee } s x); I s; \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{Suc} (\text{count } s)$
and *ignore*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf} (\text{callee } s x); I s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and *outs*: $\mathcal{I} \vdash_g \text{gpv} \checkmark$
and *I*: $I s$
shows $\text{count } s' \leq n + \text{count } s$
<proof>

lemma *pred-spmf-calleeI*: $\llbracket I s; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{pred-spmf} (\text{pred-prod} (\lambda-. \text{True}) I) (\text{callee } s x)$
<proof>

lemma *lossless-exec-gpv*:

assumes *gpv*: *lossless-gpv I gpv*
and *callee*: $\bigwedge s \text{out}. \llbracket \text{out} \in \text{outs-}\mathcal{I} \rrbracket; I s \implies \text{lossless-spmf} (\text{callee } s \text{out})$
and *WT-gpv*: $\mathcal{I} \vdash_g \text{gpv} \checkmark$
and *I*: $I s$
shows $\text{lossless-spmf} (\text{exec-gpv callee gpv } s)$
<proof>

lemma *in-set-spmf-exec-gpv-into-results-gpv*:

assumes *: $(x, s') \in \text{set-spmf} (\text{exec-gpv callee gpv } s)$

and $WT\text{-}gpv : \mathcal{I} \vdash g\text{-}gpv \checkmark$
and $I : I\ s$
shows $x \in \text{results-gpv } \mathcal{I} \text{ } gpv$
 $\langle \text{proof} \rangle$

end

lemma *callee-invariant-on-alt-def*:

$\text{callee-invariant-on} = (\lambda \text{callee } I \mathcal{I}.$
 $\quad (\forall s \in \text{Collect } I. \forall x \in \text{outs-}\mathcal{I} \mathcal{I}. \forall (y, s') \in \text{set-spmf } (\text{callee } s\ x). I\ s') \wedge$
 $\quad (\forall s \in \text{Collect } I. \mathcal{I} \vdash c \text{ callee } s \checkmark))$
 $\langle \text{proof} \rangle$

lemma *callee-invariant-on-parametric* [transfer-rule]: **includes** *lifting-syntax*

assumes [transfer-rule]: *bi-unique R bi-total S*
shows $((S \text{====>} C \text{====>} \text{rel-spmf } (\text{rel-prod } R\ S)) \text{====>} (S \text{====>} (=))$
 $\text{====>} \text{rel-}\mathcal{I} \ C\ R \text{====>} (=))$
 $\text{callee-invariant-on callee-invariant-on}$
 $\langle \text{proof} \rangle$

lemma *callee-invariant-on-cong*:

$\llbracket I = I'; \text{outs-}\mathcal{I} \ \mathcal{I} = \text{outs-}\mathcal{I} \ \mathcal{I}' \rrbracket$
 $\quad \wedge s\ x. \llbracket I' s; x \in \text{outs-}\mathcal{I} \ \mathcal{I}' \rrbracket \implies \text{set-spmf } (\text{callee } s\ x) \subseteq \text{responses-}\mathcal{I} \ \mathcal{I} \ x \times$
 $\text{Collect } I' \longleftrightarrow \text{set-spmf } (\text{callee}' s\ x) \subseteq \text{responses-}\mathcal{I} \ \mathcal{I}' \ x \times \text{Collect } I'$
 $\implies \text{callee-invariant-on callee } I \ \mathcal{I} = \text{callee-invariant-on callee}' I' \ \mathcal{I}'$
 $\langle \text{proof} \rangle$

abbreviation *callee-invariant* :: $(s \Rightarrow a \Rightarrow (b \times s) \text{ spmf}) \Rightarrow (s \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where *callee-invariant callee I* $\equiv \text{callee-invariant-on callee } I \ \mathcal{I}\text{-full}$

interpretation *oi-True*: *callee-invariant-on callee* $\lambda\cdot$. *True* \mathcal{I} -full **for** *callee*

$\langle \text{proof} \rangle$

lemma *callee-invariant-on-return-spmf* [simp]:

$\text{callee-invariant-on } (\lambda s\ x. \text{return-spmf } (f\ s\ x)) \ I \ \mathcal{I} \longleftrightarrow (\forall s. \forall x \in \text{outs-}\mathcal{I} \ \mathcal{I}. I\ s$
 $\longrightarrow I \ (\text{snd } (f\ s\ x)) \wedge \text{fst } (f\ s\ x) \in \text{responses-}\mathcal{I} \ \mathcal{I} \ x)$
 $\langle \text{proof} \rangle$

lemma *callee-invariant-return-spmf* [simp]:

$\text{callee-invariant } (\lambda s\ x. \text{return-spmf } (f\ s\ x)) \ I \longleftrightarrow (\forall s\ x. I\ s \longrightarrow I \ (\text{snd } (f\ s\ x)))$
 $\langle \text{proof} \rangle$

lemma *callee-invariant-restrict-relp*:

includes *lifting-syntax*
assumes $(S \text{====>} C \text{====>} \text{rel-spmf } (\text{rel-prod } R\ S)) \ \text{callee1} \ \text{callee2}$
and *callee-invariant callee1 I1*
and *callee-invariant callee2 I2*
shows $((S \upharpoonright I1 \otimes I2) \text{====>} C \text{====>} \text{rel-spmf } (\text{rel-prod } R \ (S \upharpoonright I1 \otimes I2)))$
 $\text{callee1} \ \text{callee2}$

$\langle \text{proof} \rangle$

lemma *callee-invariant-on-True* [simp]: *callee-invariant-on callee* $(\lambda-. \text{True}) \mathcal{I} \longleftrightarrow$
 $(\forall s. \mathcal{I} \vdash c \text{ callee } s \checkmark)$
 $\langle \text{proof} \rangle$

lemma *lossless-exec-gpv*:
[[*lossless-gpv* \mathcal{I} *gpv*; $\bigwedge s \text{ out}. \text{out} \in \text{outs-}\mathcal{I} \mathcal{I} \implies \text{lossless-spmf} (\text{callee } s \text{ out});$
 $\mathcal{I} \vdash g \text{ gpv } \checkmark$; $\bigwedge s. \mathcal{I} \vdash c \text{ callee } s \checkmark$]]
 $\implies \text{lossless-spmf} (\text{exec-gpv callee gpv } s)$
 $\langle \text{proof} \rangle$

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

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

primcorec *restrict-gpv* :: $('a, 'out, 'in) \text{gpv} \Rightarrow ('a, 'out, 'in) \text{gpv}$
where

restrict-gpv gpv = *GPV* (
map-pmf (*case-option* *None* (*case-generat* (*Some* \circ *Pure*)
 $(\lambda \text{out } c. \text{if } \text{out} \in \text{outs-}\mathcal{I} \mathcal{I} \text{ then } \text{Some} (\text{IO } \text{out} (\lambda \text{input}. \text{if } \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out then } \text{restrict-gpv} (c \text{ input}) \text{ else } \text{Fail}))$
else *None*)))
the-gpv gpv))

lemma *restrict-gpv-Done* [simp]: *restrict-gpv* (*Done* x) = *Done* x
 $\langle \text{proof} \rangle$

lemma *restrict-gpv-Fail* [simp]: *restrict-gpv* *Fail* = *Fail*
 $\langle \text{proof} \rangle$

lemma *restrict-gpv-Pause* [simp]: *restrict-gpv* (*Pause* $\text{out } c$) = (*if* $\text{out} \in \text{outs-}\mathcal{I} \mathcal{I}$
then *Pause* $\text{out} (\lambda \text{input}. \text{if } \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out then } \text{restrict-gpv} (c \text{ input})$
else *Fail*) *else* *Fail*)
 $\langle \text{proof} \rangle$

lemma *restrict-gpv-bind* [simp]: *restrict-gpv* (*bind-gpv* $\text{gpv } f$) = *bind-gpv* (*restrict-gpv*
 gpv) $(\lambda x. \text{restrict-gpv} (f x))$
 $\langle \text{proof} \rangle$

lemma *WT-restrict-gpv* [simp]: $\mathcal{I} \vdash g \text{ restrict-gpv gpv } \checkmark$
 $\langle \text{proof} \rangle$

lemma *exec-gpv-restrict-gpv*:
assumes $\mathcal{I} \vdash g \text{ gpv } \checkmark$ **and** *WT-callee*: $\bigwedge s. \mathcal{I} \vdash c \text{ callee } s \checkmark$

shows $\text{exec-gpv callee (restrict-gpv gpv) s} = \text{exec-gpv callee gpv s}$
(proof)

lemma $\text{in-outs}'\text{-restrict-gpvD}$: $x \in \text{outs}'\text{-gpv (restrict-gpv gpv)} \implies x \in \text{outs-}\mathcal{I} \mathcal{I}$
(proof)

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

lemma $\text{lossless-restrict-gpvI}$: $\llbracket \text{lossless-gpv } \mathcal{I} \text{ gpv}; \mathcal{I} \vdash_g \text{ gpv } \checkmark \rrbracket \implies \text{lossless-gpv } \mathcal{I} \text{ (restrict-gpv gpv)}$
(proof)

lemma $\text{lossless-restrict-gpvD}$: $\llbracket \text{lossless-gpv } \mathcal{I} \text{ (restrict-gpv gpv)}; \mathcal{I} \vdash_g \text{ gpv } \checkmark \rrbracket \implies \text{lossless-gpv } \mathcal{I} \text{ gpv}$
(proof)

lemma $\text{colossless-restrict-gpvD}$:
 $\llbracket \text{colossless-gpv } \mathcal{I} \text{ (restrict-gpv gpv)}; \mathcal{I} \vdash_g \text{ gpv } \checkmark \rrbracket \implies \text{colossless-gpv } \mathcal{I} \text{ gpv}$
(proof)

lemma $\text{colossless-restrict-gpvI}$:
 $\llbracket \text{colossless-gpv } \mathcal{I} \text{ gpv}; \mathcal{I} \vdash_g \text{ gpv } \checkmark \rrbracket \implies \text{colossless-gpv } \mathcal{I} \text{ (restrict-gpv gpv)}$
(proof)

lemma $\text{gen-colossless-restrict-gpv}$ [simp]:
 $\mathcal{I} \vdash_g \text{ gpv } \checkmark \implies \text{gen-lossless-gpv } b \mathcal{I} \text{ (restrict-gpv gpv)} \longleftrightarrow \text{gen-lossless-gpv } b \mathcal{I} \text{ gpv}$
(proof)

lemma $\text{interaction-bound-restrict-gpv}$:
 $\text{interaction-bound consider (restrict-gpv gpv)} \leq \text{interaction-bound consider gpv}$
(proof)

lemma $\text{interaction-bounded-by-restrict-gpvI}$ [interaction-bound, simp]:
 $\text{interaction-bounded-by consider gpv } n \implies \text{interaction-bounded-by consider (restrict-gpv gpv) } n$
(proof)

end

lemma $\text{restrict-gpv-parametric}'$:
includes lifting-syntax
notes [transfer-rule] = $\text{the-gpv-parametric}' \text{ Fail-parametric}' \text{ corec-gpv-parametric}'$
assumes [transfer-rule]: $\text{bi-unique } C \text{ bi-unique } R$
shows $(\text{rel-}\mathcal{I} \ C \ R \implies \text{rel-gpv}'' \ A \ C \ R \implies \text{rel-gpv}'' \ A \ C \ R) \text{ restrict-gpv}$
 restrict-gpv
(proof)

lemma $\text{restrict-gpv-parametric}$ [transfer-rule]: **includes** lifting-syntax **shows**

bi-unique $C \implies (\text{rel-}\mathcal{I} \ C (=) \implies \text{rel-gpv} \ A \ C \implies \text{rel-gpv} \ A \ C) \text{ restrict-gpv}$
restrict-gpv
 $\langle \text{proof} \rangle$

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

lemma (*in callee-invariant-on*) *exec-gpv-restrict-gpv-invariant*:
assumes $\mathcal{I} \vdash g \ gpv \ \checkmark$ **and** $I \ s$
shows $\text{exec-gpv} \ \text{callee} \ (\text{restrict-gpv} \ \mathcal{I} \ gpv) \ s = \text{exec-gpv} \ \text{callee} \ gpv \ s$
 $\langle \text{proof} \rangle$

lemma *in-results-gpv-restrict-gpvD*:
assumes $x \in \text{results-gpv} \ \mathcal{I} \ (\text{restrict-gpv} \ \mathcal{I}' \ gpv)$
shows $x \in \text{results-gpv} \ \mathcal{I} \ gpv$
 $\langle \text{proof} \rangle$

lemma *results-gpv-restrict-gpv*:
 $\text{results-gpv} \ \mathcal{I} \ (\text{restrict-gpv} \ \mathcal{I}' \ gpv) \subseteq \text{results-gpv} \ \mathcal{I} \ gpv$
 $\langle \text{proof} \rangle$

lemma *in-results'-gpv-restrict-gpvD*:
 $x \in \text{results}'\text{-gpv} \ (\text{restrict-gpv} \ \mathcal{I}' \ gpv) \implies x \in \text{results}'\text{-gpv} \ gpv$
 $\langle \text{proof} \rangle$

primcorec *enforce- \mathcal{I} -gpv* :: $('out, 'in) \ \mathcal{I} \Rightarrow ('a, 'out, 'in) \ gpv \Rightarrow ('a, 'out, 'in) \ gpv$
where
 $\text{enforce-}\mathcal{I}\text{-gpv} \ \mathcal{I} \ gpv = GPV$
 $(\text{map-spmf} \ (\text{map-generat} \ id \ id \ ((\circ) \ (\text{enforce-}\mathcal{I}\text{-gpv} \ \mathcal{I}))))$
 $(\text{map-spmf} \ (\lambda \text{generat. case generat of Pure } x \Rightarrow \text{Pure } x \mid IO \ out \ rpv \Rightarrow IO \ out$
 $(\lambda \text{input. if input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ out \ \text{then } rpv \ \text{input} \ \text{else Fail}))$
 $(\text{enforce-spmf} \ (\text{pred-generat} \ \top \ (\lambda x. x \in \text{outs-}\mathcal{I} \ \mathcal{I}) \ \top) \ (\text{the-gpv} \ gpv))))$

lemma *enforce- \mathcal{I} -gpv-Done [simp]*: $\text{enforce-}\mathcal{I}\text{-gpv} \ \mathcal{I} \ (\text{Done} \ x) = \text{Done} \ x$
 $\langle \text{proof} \rangle$

lemma *enforce- \mathcal{I} -gpv-Fail [simp]*: $\text{enforce-}\mathcal{I}\text{-gpv} \ \mathcal{I} \ \text{Fail} = \text{Fail}$
 $\langle \text{proof} \rangle$

lemma *enforce- \mathcal{I} -gpv-Pause [simp]*:
 $\text{enforce-}\mathcal{I}\text{-gpv} \ \mathcal{I} \ (\text{Pause} \ out \ rpv) =$
 $(\text{if } out \in \text{outs-}\mathcal{I} \ \mathcal{I} \ \text{then } \text{Pause} \ out \ (\lambda \text{input. if input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ out \ \text{then}$
 $\text{enforce-}\mathcal{I}\text{-gpv} \ \mathcal{I} \ (rpv \ \text{input}) \ \text{else Fail}) \ \text{else Fail})$
 $\langle \text{proof} \rangle$

lemma *enforce- \mathcal{I} -gpv-lift-spmf [simp]*: $\text{enforce-}\mathcal{I}\text{-gpv} \ \mathcal{I} \ (\text{lift-spmf} \ p) = \text{lift-spmf} \ p$
 $\langle \text{proof} \rangle$

lemma *enforce- \mathcal{I} -gpv-bind-gpv* [simp]:
 $enforce\text{-}\mathcal{I}\text{-gpv } \mathcal{I} (bind\text{-gpv } gpv f) = bind\text{-gpv } (enforce\text{-}\mathcal{I}\text{-gpv } \mathcal{I} gpv) (enforce\text{-}\mathcal{I}\text{-gpv } \mathcal{I} \circ f)$
 ⟨proof⟩

lemma *enforce- \mathcal{I} -gpv-parametric'*:
includes *lifting-syntax*
notes [transfer-rule] = *corec-gpv-parametric' the-gpv-parametric' Fail-parametric'*
assumes [transfer-rule]: *bi-unique C bi-unique R*
shows ($rel\text{-}\mathcal{I} C R \implies rel\text{-gpv}'' A C R \implies rel\text{-gpv}'' A C R$) *enforce- \mathcal{I} -gpv*
enforce- \mathcal{I} -gpv
 ⟨proof⟩

lemma *enforce- \mathcal{I} -gpv-parametric* [transfer-rule]: **includes** *lifting-syntax* **shows**
 $bi\text{-unique } C \implies (rel\text{-}\mathcal{I} C (=) \implies rel\text{-gpv } A C \implies rel\text{-gpv } A C)$ *enforce- \mathcal{I} -gpv*
enforce- \mathcal{I} -gpv
 ⟨proof⟩

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

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

inductive *finite-gpv* :: $('a, 'out, 'in) gpv \Rightarrow bool$
where

finite-gpvI:
 $(\bigwedge out\ c\ input. \llbracket IO\ out\ c \in set\text{-spmf } (the\text{-gpv } gpv); input \in responses\text{-}\mathcal{I} \ \mathcal{I}\ out \rrbracket \implies finite\text{-gpv } (c\ input)) \implies finite\text{-gpv } gpv$

lemmas *finite-gpv-induct*[consumes 1, case-names *finite-gpv*, induct *pred*] = *finite-gpv.induct*

lemma *finite-gpvD*: $\llbracket finite\text{-gpv } gpv; IO\ out\ c \in set\text{-spmf } (the\text{-gpv } gpv); input \in responses\text{-}\mathcal{I} \ \mathcal{I}\ out \rrbracket \implies finite\text{-gpv } (c\ input)$
 ⟨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) $\longleftrightarrow (\forall input \in responses\text{-}\mathcal{I} \ \mathcal{I} \ x. finite\text{-gpv } (c\ input))$
 ⟨proof⟩

lemma *finite-gpv-lift-spmf* [simp]: *finite-gpv* (*lift-spmf* p)
 ⟨proof⟩

lemma *finite-gpv-bind* [*simp*]:
 $finite-gpv (gpv \gg= f) \longleftrightarrow finite-gpv\ gpv \wedge (\forall x \in results-gpv\ \mathcal{I}\ gpv.\ finite-gpv\ (f\ x))$
(is ?lhs = ?rhs)
<proof>

end

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

lemma *finite-gpv-rel''D1*:
assumes *rel-gpv'' A C R gpv gpv'* and *finite-gpv I gpv* and $\mathcal{I}: rel\text{-}\mathcal{I}\ C\ R\ \mathcal{I}\ \mathcal{I}'$
shows *finite-gpv I' gpv'*
<proof>

lemma *finite-gpv-relD1*: $\llbracket rel-gpv\ A\ C\ gpv\ gpv';\ finite-gpv\ \mathcal{I}\ gpv;\ rel\text{-}\mathcal{I}\ C\ (=)\ \mathcal{I}\ \mathcal{I}' \rrbracket \Longrightarrow finite-gpv\ \mathcal{I}'\ gpv'$
<proof>

lemma *finite-gpv-rel''D2*: $\llbracket rel-gpv''\ A\ C\ R\ gpv\ gpv';\ finite-gpv\ \mathcal{I}\ gpv';\ rel\text{-}\mathcal{I}\ C\ R\ \mathcal{I}'\ \mathcal{I}' \rrbracket \Longrightarrow finite-gpv\ \mathcal{I}'\ gpv$
<proof>

lemma *finite-gpv-relD2*: $\llbracket rel-gpv\ A\ C\ gpv\ gpv';\ finite-gpv\ \mathcal{I}\ gpv';\ rel\text{-}\mathcal{I}\ C\ (=)\ \mathcal{I}\ \mathcal{I}' \rrbracket \Longrightarrow finite-gpv\ \mathcal{I}\ gpv$
<proof>

lemma *finite-gpv-parametric'*: $(rel\text{-}\mathcal{I}\ C\ R \implies rel-gpv''\ A\ C\ R \implies (=))\ finite-gpv\ finite-gpv$
<proof>

lemma *finite-gpv-parametric* [*transfer-rule*]: $(rel\text{-}\mathcal{I}\ C\ (=) \implies rel-gpv\ A\ C \implies (=))\ finite-gpv\ finite-gpv$
<proof>

end

lemma *finite-gpv-map* [*simp*]: $finite-gpv\ \mathcal{I}\ (map-gpv\ f\ id\ gpv) = finite-gpv\ \mathcal{I}\ gpv$
<proof>

lemma *finite-gpv-assert* [*simp*]: $finite-gpv\ \mathcal{I}\ (assert-gpv\ b)$
<proof>

lemma *finite-gpv-try* [*simp*]:
 $finite-gpv\ \mathcal{I}\ (TRY\ gpv\ ELSE\ gpv') \longleftrightarrow finite-gpv\ \mathcal{I}\ gpv \wedge (colossless-gpv\ \mathcal{I}\ gpv \vee finite-gpv\ \mathcal{I}\ gpv')$
(is ?lhs = -)
<proof>

lemma *lossless-gpv-conv-finite*:

lossless-gpv \mathcal{I} *gpv* \longleftrightarrow *finite-gpv* \mathcal{I} *gpv* \wedge *colossless-gpv* \mathcal{I} *gpv*
 (is ?*loss* \longleftrightarrow ?*fin* \wedge ?*co*)

\langle *proof* \rangle

lemma *colossless-gpv-try* [*simp*]:

colossless-gpv \mathcal{I} (*TRY* *gpv* *ELSE* *gpv'*) \longleftrightarrow *colossless-gpv* \mathcal{I} *gpv* \vee *colossless-gpv*
 \mathcal{I} *gpv'*

(is ?*lhs* \longleftrightarrow ?*gpv* \vee ?*gpv'*)

\langle *proof* \rangle

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'*)

\langle *proof* \rangle

lemma *interaction-any-bounded-by-imp-finite*:

assumes *interaction-any-bounded-by* *gpv* (*enat* *n*)

shows *finite-gpv* \mathcal{I} -full *gpv*

\langle *proof* \rangle

lemma *finite-restrict-gpvI* [*simp*]: *finite-gpv* \mathcal{I}' *gpv* \implies *finite-gpv* \mathcal{I}' (*restrict-gpv*
 \mathcal{I} *gpv*)

\langle *proof* \rangle

lemma *interaction-bounded-by-exec-gpv-bad-count*:

fixes *count* **and** *bad* **and** *n* :: *enat* **and** *k* :: *real*

assumes *bound*: *interaction-bounded-by* *consider* *gpv* *n*

and *good*: \neg *bad* *s*

and *count*: $\bigwedge s$ *x y s'*. $\llbracket (y, s') \in \text{set-spmf} (\text{callee } s \ x); \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$
 \implies *count* *s'* \leq *Suc* (*count* *s*)

and *ignore*: $\bigwedge s$ *x y s'*. $\llbracket (y, s') \in \text{set-spmf} (\text{callee } s \ x); \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$
 \implies *count* *s'* \leq *count* *s*

and *bad*: $\bigwedge s' *x*. $\llbracket \neg \text{bad } s'; \text{count } s' < n + \text{count } s; \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$
 \implies *spmf* (*map-spmf* (*bad* \circ *snd*) (*callee* *s'* *x*)) *True* \leq *k*$

and *consider*: $\bigwedge s$ *x y s'*. $\llbracket (y, s') \in \text{set-spmf} (\text{callee } s \ x); \neg \text{bad } s; \text{bad } s'; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket$
 \implies *consider* *x*

and *k-nonneg*: *k* \geq 0

and *WT-gpv*: $\mathcal{I} \vdash_g$ *gpv* \checkmark

and *WT-callee*: $\bigwedge s$. $\mathcal{I} \vdash_c$ *callee* *s* \checkmark

shows *spmf* (*map-spmf* (*bad* \circ *snd*) (*exec-gpv* *callee* *gpv* *s*)) *True* \leq *ennreal* *k* *
n

\langle *proof* \rangle

context *callee-invariant-on* **begin**

lemma *interaction-bounded-by-exec-gpv-bad-count*:

includes *lifting-syntax*

fixes *count* **and** *bad* **and** *n* :: *enat*
assumes *bound*: *interaction-bounded-by consider gpv n*
and *I*: *I s*
and *good*: \neg *bad s*
and *count*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s x); I s; \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{Suc } (\text{count } s)$
and *ignore*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s x); I s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and *bad*: $\bigwedge s' x. \llbracket I s'; \neg \text{bad } s'; \text{count } s' < n + \text{count } s; \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{callee } s' x)) \text{ True} \leq k$
and *consider*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s x); I s; \neg \text{bad } s; \text{bad } s'; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{consider } x$
and *k-nonneg*: $k \geq 0$
and *WT-gpv*: $\mathcal{I} \vdash g \text{ gpv} \checkmark$
shows $\text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{exec-gpv callee gpv s})) \text{ True} \leq \text{ennreal } k * n$
<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*: \neg *bad s*
and *count*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s x); I s; \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{Suc } (\text{count } s)$
and *ignore*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s x); I s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and *bad*: $\bigwedge s' x. \llbracket I s'; \neg \text{bad } s'; \text{count } s' < n + \text{count } s; \text{consider } x; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{callee } s' x)) \text{ True} \leq k$
and *consider*: $\bigwedge s x y s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s x); I s; \neg \text{bad } s; \text{bad } s'; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{consider } x$
and *k-nonneg*: $k \geq 0$
and *WT-gpv*: $\mathcal{I} \vdash g \text{ gpv} \checkmark$
shows $\text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{exec-gpv callee gpv s})) \text{ True} \leq k * n$
<proof>

lemma *interaction-bounded-by-exec-gpv-bad*:

assumes *interaction-any-bounded-by gpv n*
and *I s* \neg *bad s*
and *bad*: $\bigwedge s x. \llbracket I s; \neg \text{bad } s; x \in \text{outs-}\mathcal{I} \rrbracket \implies \text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{callee } s x)) \text{ True} \leq k$
and *k-nonneg*: $0 \leq k$
and *WT-gpv*: $\mathcal{I} \vdash g \text{ gpv} \checkmark$
shows $\text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{exec-gpv callee gpv s})) \text{ True} \leq k * n$
<proof>

end

end

5 Oracle combinators

theory *Computational-Model* **imports**

Generative-Probabilistic-Value

begin

type-synonym *security* = *nat*

type-synonym *advantage* = *security* \Rightarrow *real*

type-synonym (*' σ* , *'call*, *'ret*) *oracle'* = *' σ* \Rightarrow *'call* \Rightarrow (*'ret* \times *' σ*) *spmf*

type-synonym (*' σ* , *'call*, *'ret*) *oracle* = *security* \Rightarrow (*' σ* , *'call*, *'ret*) *oracle'* \times *' σ*

$\langle ML \rangle$

typ (*' σ* , *'call*, *'ret*) *oracle*

5.1 Shared state

context includes *\mathcal{I} .lifting* and *lifting-syntax* **begin**

lift-definition *plus- \mathcal{I}* :: (*'out*, *'ret*) *\mathcal{I}* \Rightarrow (*'out'*, *'ret'*) *\mathcal{I}* \Rightarrow (*'out* + *'out'*, *'ret* + *'ret'*) *\mathcal{I}* (**infix** $\langle \oplus_{\mathcal{I}} \rangle$ 500)

is λ *resp1 resp2*. λ *out*. *case out of* *Inl out'* \Rightarrow *Inl ' resp1 out'* | *Inr out'* \Rightarrow *Inr ' resp2 out'* \langle *proof* \rangle

lemma *plus- \mathcal{I} -sel* [*simp*]:

shows *outs-plus- \mathcal{I}* : *outs- \mathcal{I}* (*plus- \mathcal{I}* *$\mathcal{I}l$* *$\mathcal{I}r$*) = *outs- \mathcal{I}* *$\mathcal{I}l$* $\langle + \rangle$ *outs- \mathcal{I}* *$\mathcal{I}r$*

and *responses-plus- \mathcal{I} -Inl*: *responses- \mathcal{I}* (*plus- \mathcal{I}* *$\mathcal{I}l$* *$\mathcal{I}r$*) (*Inl x*) = *Inl ' responses- \mathcal{I}* *$\mathcal{I}l$* *x*

and *responses-plus- \mathcal{I} -Inr*: *responses- \mathcal{I}* (*plus- \mathcal{I}* *$\mathcal{I}l$* *$\mathcal{I}r$*) (*Inr y*) = *Inr ' responses- \mathcal{I}* *$\mathcal{I}r$* *y*

\langle *proof* \rangle

lemma *vimage-Inl-Plus* [*simp*]: *Inl - ' (A* $\langle + \rangle$ *B) = A*

and *vimage-Inr-Plus* [*simp*]: *Inr - ' (A* $\langle + \rangle$ *B) = B*

\langle *proof* \rangle

lemma *vimage-Inl-image-Inr*: *Inl - ' Inr ' A = {}*

and *vimage-Inr-image-Inl*: *Inr - ' Inl ' A = {}*

\langle *proof* \rangle

lemma *plus- \mathcal{I} -parametric* [*transfer-rule*]:

(*rel- \mathcal{I}* *C R* $====>$ *rel- \mathcal{I}* *C' R'* $====>$ *rel- \mathcal{I}* (*rel-sum C C'*) (*rel-sum R R'*)) *plus- \mathcal{I}*

\langle *proof* \rangle

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

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

lemma *\mathcal{I} -trivial-plus- \mathcal{I}* [*simp*]: *\mathcal{I} -trivial* (*$\mathcal{I}_1 \oplus_{\mathcal{I}} \mathcal{I}_2$*) \longleftrightarrow *\mathcal{I} -trivial* *\mathcal{I}_1* \wedge *\mathcal{I} -trivial* *\mathcal{I}_2*

<proof>

end

lemma *map-I-plus-I* [*simp*]:

$map-I (map-sum f1 f2) (map-sum g1 g2) (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) = map-I f1 g1 \mathcal{I}1 \oplus_{\mathcal{I}} map-I f2 g2 \mathcal{I}2$

<proof>

lemma *le-plus-I-iff* [*simp*]:

$\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \leq \mathcal{I}1' \oplus_{\mathcal{I}} \mathcal{I}2' \iff \mathcal{I}1 \leq \mathcal{I}1' \wedge \mathcal{I}2 \leq \mathcal{I}2'$

<proof>

lemma *I-full-le-plus-I*: $\mathcal{I}\text{-full} \leq plus-I \mathcal{I}1 \mathcal{I}2$ **if** $\mathcal{I}\text{-full} \leq \mathcal{I}1$ $\mathcal{I}\text{-full} \leq \mathcal{I}2$

<proof>

lemma *plus-I-mono*: $plus-I \mathcal{I}1 \mathcal{I}2 \leq plus-I \mathcal{I}1' \mathcal{I}2'$ **if** $\mathcal{I}1 \leq \mathcal{I}1'$ $\mathcal{I}2 \leq \mathcal{I}2'$

<proof>

context

fixes *left* :: ('s, 'a, 'b) oracle'

and *right* :: ('s, 'c, 'd) oracle'

and *s* :: 's

begin

primrec *plus-oracle* :: 'a + 'c \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*]:

$\llbracket \bigwedge a. x = Inl a \implies lossless-spmf (left s a);$

$\bigwedge b. x = Inr b \implies lossless-spmf (right s b) \rrbracket$

$\implies lossless-spmf (plus-oracle x)$

<proof>

lemma *plus-oracle-split*:

$P (plus-oracle lr) \iff$

$(\forall x. lr = Inl x \longrightarrow P (map-spmf (apfst Inl) (left s x))) \wedge$

$(\forall y. lr = Inr y \longrightarrow P (map-spmf (apfst Inr) (right s y)))$

<proof>

lemma *plus-oracle-split-asm*:

$P (plus-oracle lr) \iff$

$\neg ((\exists x. lr = Inl x \wedge \neg P (map-spmf (apfst Inl) (left s x))) \vee$

$(\exists y. lr = Inr y \wedge \neg P (map-spmf (apfst Inr) (right s y))))$

<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!*]:

$\llbracket \mathcal{I}l \vdash c \text{ left } s \checkmark; \mathcal{I}r \vdash c \text{ right } s \checkmark \rrbracket \implies \mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c \text{ (left } \oplus_O \text{ right) } s \checkmark$
<proof>

lemma *WT-plus-oracleD1*:

assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c \text{ (left } \oplus_O \text{ right) } s \checkmark$ (**is** $?\mathcal{I} \vdash c \text{ ?callee } s \checkmark$)

shows $\mathcal{I}l \vdash c \text{ left } s \checkmark$

<proof>

lemma *WT-plus-oracleD2*:

assumes $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c \text{ (left } \oplus_O \text{ right) } s \checkmark$ (**is** $?\mathcal{I} \vdash c \text{ ?callee } s \checkmark$)

shows $\mathcal{I}r \vdash c \text{ right } s \checkmark$

<proof>

lemma *WT-plus-oracle-iff* [*simp*]: $\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r \vdash c \text{ (left } \oplus_O \text{ right) } s \checkmark \iff \mathcal{I}l \vdash c \text{ left } s \checkmark \wedge \mathcal{I}r \vdash c \text{ right } s \checkmark$

<proof>

lemma *callee-invariant-on-plus-oracle* [*simp*]:

callee-invariant-on (left \oplus_O right) $I \iff$

callee-invariant-on left $I \mathcal{I}l \wedge$ *callee-invariant-on* right $I \mathcal{I}r$

(**is** $?\text{lhs} \iff ?\text{rhs}$)

<proof>

lemma *callee-invariant-plus-oracle* [*simp*]:

callee-invariant (left \oplus_O right) $I \iff$

callee-invariant left $I \wedge$ *callee-invariant* right I

(**is** $?\text{lhs} \iff ?\text{rhs}$)

<proof>

lemma *plus-oracle-parametric* [*transfer-rule*]:

includes *lifting-syntax* **shows**

$((S \implies A \implies \text{rel-spmf (rel-prod B S)})$

$\implies (S \implies C \implies \text{rel-spmf (rel-prod D S)})$

$\implies S \implies \text{rel-sum A C} \implies \text{rel-spmf (rel-prod (rel-sum B D) S)})$

plus-oracle plus-oracle

<proof>

lemma *rel-spmf-plus-oracle*:

$\llbracket \wedge q1' q2'. \llbracket q1 = \text{Inl } q1'; q2 = \text{Inl } q2' \rrbracket \implies \text{rel-spmf (rel-prod B S) (left1 s1 } q1') \text{ (left2 s2 } q2') \rrbracket$

$\bigwedge q1' q2'. \llbracket q1 = \text{Inr } q1'; q2 = \text{Inr } q2' \rrbracket \implies \text{rel-spmf } (\text{rel-prod } D \ S) \ (\text{right1 } s1 \ q1') \ (\text{right2 } s2 \ q2')$;
 $S \ s1 \ s2; \text{rel-sum } A \ C \ q1 \ q2 \llbracket$
 $\implies \text{rel-spmf } (\text{rel-prod } (\text{rel-sum } B \ D) \ S) \ ((\text{left1 } \oplus_O \ \text{right1}) \ s1 \ q1) \ ((\text{left2 } \oplus_O \ \text{right2}) \ s2 \ q2)$
 $\langle \text{proof} \rangle$

end

5.2 Shared state with aborts

context

fixes $\text{left} :: ('s, 'a, 'b \ \text{option}) \ \text{oracle}'$
and $\text{right} :: ('s, 'c, 'd \ \text{option}) \ \text{oracle}'$
and $s :: 's$

begin

primrec $\text{plus-oracle-stop} :: 'a + 'c \Rightarrow (('b + 'd) \ \text{option} \times 's) \ \text{spmf}$

where

$\text{plus-oracle-stop } (\text{Inl } a) = \text{map-spmf } (\text{apfst } (\text{map-option } \text{Inl})) \ (\text{left } s \ a)$
 $|\ \text{plus-oracle-stop } (\text{Inr } b) = \text{map-spmf } (\text{apfst } (\text{map-option } \text{Inr})) \ (\text{right } s \ b)$

lemma $\text{lossless-plus-oracle-stopI}$ [*intro, simp*]:

$\llbracket \bigwedge a. x = \text{Inl } a \implies \text{lossless-spmf } (\text{left } s \ a);$
 $\bigwedge b. x = \text{Inr } b \implies \text{lossless-spmf } (\text{right } s \ b) \rrbracket$
 $\implies \text{lossless-spmf } (\text{plus-oracle-stop } x)$
 $\langle \text{proof} \rangle$

lemma $\text{plus-oracle-stop-split}$:

$P \ (\text{plus-oracle-stop } lr) \longleftrightarrow$
 $(\forall x. lr = \text{Inl } x \longrightarrow P \ (\text{map-spmf } (\text{apfst } (\text{map-option } \text{Inl})) \ (\text{left } s \ x))) \wedge$
 $(\forall y. lr = \text{Inr } y \longrightarrow P \ (\text{map-spmf } (\text{apfst } (\text{map-option } \text{Inr})) \ (\text{right } s \ y)))$
 $\langle \text{proof} \rangle$

lemma $\text{plus-oracle-stop-split-asm}$:

$P \ (\text{plus-oracle-stop } lr) \longleftrightarrow$
 $\neg ((\exists x. lr = \text{Inl } x \wedge \neg P \ (\text{map-spmf } (\text{apfst } (\text{map-option } \text{Inl})) \ (\text{left } s \ x))) \vee$
 $(\exists y. lr = \text{Inr } y \wedge \neg P \ (\text{map-spmf } (\text{apfst } (\text{map-option } \text{Inr})) \ (\text{right } s \ y))))$
 $\langle \text{proof} \rangle$

end

notation plus-oracle-stop (**infix** $\langle \oplus_O^S \rangle$ 500)

5.3 Disjoint state

context

fixes $\text{left} :: ('s1, 'a, 'b) \ \text{oracle}'$
and $\text{right} :: ('s2, 'c, 'd) \ \text{oracle}'$

begin

fun *parallel-oracle* :: ('s1 × 's2, 'a + 'c, 'b + 'd) oracle'

where

parallel-oracle (s1, s2) (Inl a) = map-spmf (map-prod Inl (λs1'. (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 (λa. map-spmf (map-prod Inl (λs1'. (s1', s2))) (left s1 a)) (λb. map-spmf (map-prod Inr (Pair s1)) (right s2 b)))
⟨proof⟩

lemma *lossless-parallel-oracle* [simp]:

lossless-spmf (*parallel-oracle* s1s2 xy) ↔
(∀ x. xy = Inl x → *lossless-spmf* (left (fst s1s2) x)) ∧
(∀ y. xy = Inr y → *lossless-spmf* (right (snd s1s2) 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 (λs1'. (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 (λs1'. (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]:

[[*I*l ⊢_c left sl √; *I*r ⊢_c right sr √]] ⇒ plus-*I* *I*l *I*r ⊢_c *parallel-oracle* (sl, sr)
√
⟨proof⟩

lemma *callee-invariant-parallel-oracleI* [simp, intro]:

assumes *callee-invariant-on* left *I*l *I*l *callee-invariant-on* right *I*r *I*r
shows *callee-invariant-on* *parallel-oracle* (pred-prod *I*l *I*r) (*I*l ⊕_{*I*} *I*r)
⟨proof⟩

end

lemma *parallel-oracle-parametric*:

includes *lifting-syntax* **shows**

$((S1 \text{ ===> } CALL1 \text{ ===> } \text{rel-spmf } (\text{rel-prod } (=) S1))$
 $\text{ ===> } (S2 \text{ ===> } CALL2 \text{ ===> } \text{rel-spmf } (\text{rel-prod } (=) S2))$
 $\text{ ===> } \text{rel-prod } S1 S2 \text{ ===> } \text{rel-sum } CALL1 CALL2 \text{ ===> } \text{rel-spmf } (\text{rel-prod}$
 $(=) (\text{rel-prod } S1 S2))$
parallel-oracle parallel-oracle
 <proof>

5.4 Indexed oracles

definition *family-oracle* :: ('i \Rightarrow ('s, 'a, 'b) oracle') \Rightarrow ('i \Rightarrow 's, 'i \times 'a, 'b) oracle'
where *family-oracle* f s = ($\lambda(i, x). \text{map-spmf } (\lambda(y, s'). (y, s(i := s')))$ (f i (s i) x))

lemma *family-oracle-apply* [simp]:
family-oracle f s (i, x) = $\text{map-spmf } (\text{apsnd } (\text{fun-upd } s i))$ (f i (s i) x)
 <proof>

lemma *lossless-family-oracle*:
 $\text{lossless-spmf } (\text{family-oracle } f s ix) \iff \text{lossless-spmf } (f (\text{fst } ix) (s (\text{fst } ix))) (\text{snd } ix)$
 <proof>

5.5 State extension

definition *extend-state-oracle* :: ('call, 'ret, 's) callee \Rightarrow ('call, 'ret, 's' \times 's) callee
 ($\dagger \rightarrow [1000] 1000$)
where *extend-state-oracle* callee = ($\lambda(s', s) x. \text{map-spmf } (\lambda(y, s). (y, (s', s)))$ (callee s x))

lemma *extend-state-oracle-simps* [simp]:
 $\text{extend-state-oracle } \text{callee } (s', s) x = \text{map-spmf } (\lambda(y, s). (y, (s', s)))$ (callee s x)
 <proof>

context includes *lifting-syntax begin*

lemma *extend-state-oracle-parametric* [transfer-rule]:
 $((S \text{ ===> } C \text{ ===> } \text{rel-spmf } (\text{rel-prod } R S)) \text{ ===> } \text{rel-prod } S' S \text{ ===> } C$
 $\text{ ===> } \text{rel-spmf } (\text{rel-prod } R (\text{rel-prod } S' S)))$
extend-state-oracle extend-state-oracle
 <proof>

lemma *extend-state-oracle-transfer*:
 $((S \text{ ===> } C \text{ ===> } \text{rel-spmf } (\text{rel-prod } R S))$
 $\text{ ===> } \text{rel-prod2 } S \text{ ===> } C \text{ ===> } \text{rel-spmf } (\text{rel-prod } R (\text{rel-prod2 } S)))$
 $(\lambda \text{oracle. oracle}) \text{ extend-state-oracle}$
 <proof>

end

lemma *callee-invariant-extend-state-oracle-const* [simp]:
 $\text{callee-invariant } \dagger \text{oracle } (\lambda(s', s). I s')$
 <proof>

lemma *callee-invariant-extend-state-oracle-const'*:

callee-invariant †oracle (λs. 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)))
(λx. x) *lift-stop-oracle*
⟨proof⟩

end

definition *extend-state-oracle2* :: ('call, 'ret, 's) callee ⇒ ('call, 'ret, 's × 's) callee (λs. s) [1000] 1000
where *extend-state-oracle2* callee = (λ(s, s') x. map-spmf (λ(y, s). (y, (s, s')))) (callee s x)

lemma *extend-state-oracle2-simps* [simp]:
extend-state-oracle2 callee (s, s') x = map-spmf (λ(y, s). (y, (s, s')))) (callee s x)
⟨proof⟩

lemma *extend-state-oracle2-parametric* [transfer-rule]: **includes** *lifting-syntax* **shows**
((S ==> C ==> rel-spmf (rel-prod R S)) ==> rel-prod S S' ==> C ==> rel-spmf (rel-prod R (rel-prod S S'))))
extend-state-oracle2 *extend-state-oracle2*
⟨proof⟩

lemma *callee-invariant-extend-state-oracle2-const* [simp]:
callee-invariant oracle† (λ(s, s'). I s')
⟨proof⟩

lemma *callee-invariant-extend-state-oracle2-const'*:
callee-invariant oracle† (λs. I (snd s))
⟨proof⟩

lemma *extend-state-oracle2-plus-oracle*:
extend-state-oracle2 (plus-oracle oracle1 oracle2) = plus-oracle (*extend-state-oracle2* oracle1) (*extend-state-oracle2* oracle2)
⟨proof⟩

lemma *parallel-oracle-conv-plus-oracle*:

$parallel-oracle\ oracle1\ oracle2 = plus-oracle\ (oracle1\ \dagger)\ (\dagger\ oracle2)$
 $\langle proof \rangle$

lemma *map-sum-parallel-oracle*: **includes** *lifting-syntax* **shows**

$(id\ \text{----}\>\ map-sum\ f\ g\ \text{----}\>\ map-spmf\ (map-prod\ (map-sum\ h\ k)\ id))\ (parallel-oracle\ oracle1\ oracle2)$
 $=\ parallel-oracle\ ((id\ \text{----}\>\ f\ \text{----}\>\ map-spmf\ (map-prod\ h\ id))\ oracle1)\ ((id\ \text{----}\>\ g\ \text{----}\>\ map-spmf\ (map-prod\ k\ id))\ oracle2)$
 $\langle proof \rangle$

lemma *map-sum-plus-oracle*: **includes** *lifting-syntax* **shows**

$(id\ \text{----}\>\ map-sum\ f\ g\ \text{----}\>\ map-spmf\ (map-prod\ (map-sum\ h\ k)\ id))\ (plus-oracle\ oracle1\ oracle2)$
 $=\ plus-oracle\ ((id\ \text{----}\>\ f\ \text{----}\>\ map-spmf\ (map-prod\ h\ id))\ oracle1)\ ((id\ \text{----}\>\ g\ \text{----}\>\ map-spmf\ (map-prod\ k\ id))\ oracle2)$
 $\langle proof \rangle$

lemma *map-rsuml-plus-oracle*: **includes** *lifting-syntax* **shows**

$(id\ \text{----}\>\ rsuml\ \text{----}\>\ (map-spmf\ (map-prod\ lsumr\ id)))\ (oracle1\ \oplus_O\ (oracle2\ \oplus_O\ oracle3)) =$
 $((oracle1\ \oplus_O\ oracle2)\ \oplus_O\ oracle3)$
 $\langle proof \rangle$

lemma *map-lsumr-plus-oracle*: **includes** *lifting-syntax* **shows**

$(id\ \text{----}\>\ lsumr\ \text{----}\>\ (map-spmf\ (map-prod\ rsuml\ id)))\ ((oracle1\ \oplus_O\ oracle2)\ \oplus_O\ oracle3) =$
 $(oracle1\ \oplus_O\ (oracle2\ \oplus_O\ oracle3))$
 $\langle proof \rangle$

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

definition *lift-state-oracle*

$::\ (('s\ \Rightarrow\ 'a\ \Rightarrow\ (('b\ \times\ 't)\ \times\ 's)\ spmf)\ \Rightarrow\ ('s'\ \Rightarrow\ 'a\ \Rightarrow\ (('b\ \times\ 't)\ \times\ 's')\ spmf))$
 $\Rightarrow\ ('t\ \times\ 's\ \Rightarrow\ 'a\ \Rightarrow\ ('b\ \times\ 't\ \times\ 's)\ spmf)\ \Rightarrow\ ('t\ \times\ 's'\ \Rightarrow\ 'a\ \Rightarrow\ ('b\ \times\ 't\ \times\ 's')\ spmf)$ **where**

$lift-state-oracle\ F\ oracle =$
 $(\lambda(t,\ s')\ a.\ map-spmf\ rprodl\ (F\ ((Pair\ t\ \text{----}\>\ id\ \text{----}\>\ map-spmf\ lprodr)\ oracle)\ s'\ a))$

lemma *lift-state-oracle-simps* [*simp*]:

$lift-state-oracle\ F\ oracle\ (t,\ s')\ a = map-spmf\ rprodl\ (F\ ((Pair\ t\ \text{----}\>\ id\ \text{----}\>\ map-spmf\ lprodr)\ oracle)\ s'\ a)$
 $\langle proof \rangle$

lemma *lift-state-oracle-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**

$((S\ \text{====}\>\ A\ \text{====}\>\ rel-spmf\ (rel-prod\ (rel-prod\ B\ T)\ S))\ \text{====}\>\ S'\ \text{====}\>\ A\ \text{====}\>\ rel-spmf\ (rel-prod\ (rel-prod\ B\ T)\ S'))$

$\implies (rel\text{-}prod\ T\ S \implies A \implies rel\text{-}spmf\ (rel\text{-}prod\ B\ (rel\text{-}prod\ T\ S)))$
 $\implies rel\text{-}prod\ T\ S' \implies A \implies rel\text{-}spmf\ (rel\text{-}prod\ B\ (rel\text{-}prod\ T\ S'))$
lift-state-oracle lift-state-oracle
<proof>

lemma *lift-state-oracle-extend-state-oracle*:

includes *lifting-syntax*
assumes $\bigwedge B. Transfer.Rel\ ((=) \implies (=) \implies rel\text{-}spmf\ (rel\text{-}prod\ B\ (=)))$
 $\implies (=) \implies (=) \implies rel\text{-}spmf\ (rel\text{-}prod\ B\ (=))\ G\ F$

shows $lift\text{-}state\text{-}oracle\ F\ (extend\text{-}state\text{-}oracle\ oracle) = extend\text{-}state\text{-}oracle\ (G\ oracle)$
<proof>

lemma *lift-state-oracle-compose*:

$lift\text{-}state\text{-}oracle\ F\ (lift\text{-}state\text{-}oracle\ G\ oracle) = lift\text{-}state\text{-}oracle\ (F \circ G)\ oracle$
<proof>

lemma *lift-state-oracle-id [simp]*: $lift\text{-}state\text{-}oracle\ id = id$

<proof>

lemma *rprodl-extend-state-oracle*: **includes** *lifting-syntax* **shows**

$(rprodl\ \text{----}\>\ id\ \text{----}\>\ map\text{-}spmf\ (map\text{-}prod\ id\ lprodr))\ (extend\text{-}state\text{-}oracle\ (extend\text{-}state\text{-}oracle\ oracle)) =$
 $extend\text{-}state\text{-}oracle\ oracle$
<proof>

end

6 Combining GPVs

6.1 Shared state without interrupts

context

fixes $left :: 's \Rightarrow 'x1 \Rightarrow ('y1 \times 's, 'call, 'ret)\ gpv$

and $right :: 's \Rightarrow 'x2 \Rightarrow ('y2 \times 's, 'call, 'ret)\ gpv$

begin

primrec $plus\text{-}intercept :: 's \Rightarrow 'x1 + 'x2 \Rightarrow (('y1 + 'y2) \times 's, 'call, 'ret)\ gpv$

where

$plus\text{-}intercept\ s\ (Inl\ x) = map\text{-}gpv\ (apfst\ Inl)\ id\ (left\ s\ x)$

| $plus\text{-}intercept\ s\ (Inr\ x) = map\text{-}gpv\ (apfst\ Inr)\ id\ (right\ s\ x)$

end

lemma *plus-intercept-parametric [transfer-rule]*:

includes *lifting-syntax* **shows**

$((S \implies X1 \implies rel\text{-}gpv\ (rel\text{-}prod\ Y1\ S)\ C)$

$\implies (S \implies X2 \implies rel\text{-}gpv\ (rel\text{-}prod\ Y2\ S)\ C)$

$====> S$ $====> \text{rel-sum } X1 \ X2$ $====> \text{rel-gpv } (\text{rel-prod } (\text{rel-sum } Y1 \ Y2) \ S)$
C)
plus-intercept plus-intercept
 <proof>

lemma *interaction-bounded-by-plus-intercept* [*interaction-bound*]:

fixes *left right*
shows $\llbracket \bigwedge x'. x = \text{Inl } x' \implies \text{interaction-bounded-by } P (\text{left } s \ x') (n \ x');$
 $\bigwedge y. x = \text{Inr } y \implies \text{interaction-bounded-by } P (\text{right } s \ y) (m \ y) \rrbracket$
 $\implies \text{interaction-bounded-by } P (\text{plus-intercept } \text{left } \text{right } s \ x) (\text{case } x \ \text{of } \text{Inl } x \Rightarrow n$
 $x \mid \text{Inr } y \Rightarrow m \ y)$
 <proof>

6.2 Shared state with interrupts

context

fixes *left* :: $'s \Rightarrow 'x1 \Rightarrow ('y1 \ \text{option} \times 's, 'call, 'ret) \ \text{gpv}$
and *right* :: $'s \Rightarrow 'x2 \Rightarrow ('y2 \ \text{option} \times 's, 'call, 'ret) \ \text{gpv}$
begin

primrec *plus-intercept-stop* :: $'s \Rightarrow 'x1 + 'x2 \Rightarrow (('y1 + 'y2) \ \text{option} \times 's, 'call,$
'ret) gpv

where

$\text{plus-intercept-stop } s (\text{Inl } x) = \text{map-gpv } (\text{apfst } (\text{map-option } \text{Inl})) \ \text{id } (\text{left } s \ x)$
 $\mid \text{plus-intercept-stop } s (\text{Inr } x) = \text{map-gpv } (\text{apfst } (\text{map-option } \text{Inr})) \ \text{id } (\text{right } s \ x)$

end

lemma *plus-intercept-stop-parametric* [*transfer-rule*]:

includes *lifting-syntax* **shows**
 $((S \ \text{====>} \ X1 \ \text{====>} \ \text{rel-gpv } (\text{rel-prod } (\text{rel-option } Y1) \ S) \ C)$
 $\text{====>} \ (S \ \text{====>} \ X2 \ \text{====>} \ \text{rel-gpv } (\text{rel-prod } (\text{rel-option } Y2) \ S) \ C)$
 $\text{====>} \ S \ \text{====>} \ \text{rel-sum } X1 \ X2 \ \text{====>} \ \text{rel-gpv } (\text{rel-prod } (\text{rel-option } (\text{rel-sum } Y1$
 $Y2)) \ S) \ C)$
plus-intercept-stop plus-intercept-stop
 <proof>

6.3 One-sided shifts

primcorec (*transfer*) *left-gpv* :: $('a, 'out, 'in) \ \text{gpv} \Rightarrow ('a, 'out + 'out', 'in + 'in')$
gpv **where**

$\text{the-gpv } (\text{left-gpv } \text{gpv}) =$
 $\text{map-spmf } (\text{map-generat } \text{id } \text{Inl } (\lambda \text{rpv } \text{input}. \ \text{case } \text{input} \ \text{of } \ \text{Inl } \ \text{input}' \Rightarrow \text{left-gpv}$
 $(\text{rpv } \ \text{input}') \mid - \Rightarrow \text{Fail})) \ (\text{the-gpv } \ \text{gpv})$

abbreviation *left-rpv* :: $('a, 'out, 'in) \ \text{rpv} \Rightarrow ('a, 'out + 'out', 'in + 'in')$ *rpv*
where

$\text{left-rpv } \ \text{rpv} \equiv \lambda \text{input}. \ \text{case } \text{input} \ \text{of } \ \text{Inl } \ \text{input}' \Rightarrow \text{left-gpv } (\text{rpv } \ \text{input}') \mid - \Rightarrow \text{Fail}$

primcorec (*transfer*) *right-gpv* :: ('a, 'out, 'in) *gpv* ⇒ ('a, 'out' + 'out, 'in' + 'in) *gpv* **where**

the-gpv (*right-gpv gpv*) =
map-spmf (*map-generat id Inr* (λ*rpv input. case input of Inr input' ⇒ right-gpv*
(*rpv input'*) | - ⇒ *Fail*)) (*the-gpv gpv*)

abbreviation *right-rpv* :: ('a, 'out, 'in) *rpv* ⇒ ('a, 'out' + 'out, 'in' + 'in) *rpv* **where**

right-rpv rpv ≡ λ*input. case input of Inr input' ⇒ right-gpv* (*rpv input'*) | - ⇒ *Fail*

context

includes *lifting-syntax*

notes [*transfer-rule*] = *corec-gpv-parametric' Fail-parametric' the-gpv-parametric'*
begin

lemmas *left-gpv-parametric* = *left-gpv.transfer*

lemma *left-gpv-parametric'*:

(*rel-gpv'' A C R* ==> *rel-gpv'' A (rel-sum C C') (rel-sum R R')*) *left-gpv left-gpv*
⟨*proof*⟩

lemmas *right-gpv-parametric* = *right-gpv.transfer*

lemma *right-gpv-parametric'*:

(*rel-gpv'' A C' R'* ==> *rel-gpv'' A (rel-sum C C') (rel-sum R R')*) *right-gpv*
right-gpv
⟨*proof*⟩

end

lemma *left-gpv-Done* [*simp*]: *left-gpv* (*Done x*) = *Done x*

⟨*proof*⟩

lemma *right-gpv-Done* [*simp*]: *right-gpv* (*Done x*) = *Done x*

⟨*proof*⟩

lemma *left-gpv-Pause* [*simp*]:

left-gpv (*Pause x rpv*) = *Pause* (*Inl x*) (λ*input. case input of Inl input' ⇒ left-gpv*
(*rpv input'*) | - ⇒ *Fail*)

⟨*proof*⟩

lemma *right-gpv-Pause* [*simp*]:

right-gpv (*Pause x rpv*) = *Pause* (*Inr x*) (λ*input. case input of Inr input' ⇒*
right-gpv (*rpv input'*) | - ⇒ *Fail*)

⟨*proof*⟩

lemma *left-gpv-map*: *left-gpv* (*map-gpv f g gpv*) = *map-gpv f* (*map-sum g h*)
(*left-gpv gpv*)

<proof>

lemma *right-gpv-map*: $\text{right-gpv } (\text{map-gpv } f \ g \ \text{gpv}) = \text{map-gpv } f \ (\text{map-sum } h \ g)$
(*right-gpv gpv*)
<proof>

lemma *results'-gpv-left-gpv* [*simp*]:
 $\text{results'-gpv } (\text{left-gpv } \text{gpv} :: ('a, 'out + 'out', 'in + 'in') \ \text{gpv}) = \text{results'-gpv } \text{gpv}$
(**is** ?*lhs* = ?*rhs*)
<proof>

lemma *results'-gpv-right-gpv* [*simp*]:
 $\text{results'-gpv } (\text{right-gpv } \text{gpv} :: ('a, 'out' + 'out, 'in' + 'in) \ \text{gpv}) = \text{results'-gpv } \text{gpv}$
(**is** ?*lhs* = ?*rhs*)
<proof>

lemma *left-gpv-Inl-transfer*: $\text{rel-gpv}'' (=) (\lambda l \ r. l = \text{Inl } r) (\lambda l \ r. l = \text{Inl } r) (\text{left-gpv } \text{gpv}) \ \text{gpv}$
<proof>

lemma *right-gpv-Inr-transfer*: $\text{rel-gpv}'' (=) (\lambda l \ r. l = \text{Inr } r) (\lambda l \ r. l = \text{Inr } r) (\text{right-gpv } \text{gpv}) \ \text{gpv}$
<proof>

lemma *exec-gpv-plus-oracle-left*: $\text{exec-gpv } (\text{plus-oracle } \text{oracle1 } \text{oracle2}) (\text{left-gpv } \text{gpv}) \ s = \text{exec-gpv } \text{oracle1 } \text{gpv } s$
<proof>

lemma *exec-gpv-plus-oracle-right*: $\text{exec-gpv } (\text{plus-oracle } \text{oracle1 } \text{oracle2}) (\text{right-gpv } \text{gpv}) \ s = \text{exec-gpv } \text{oracle2 } \text{gpv } s$
<proof>

lemma *left-gpv-bind-gpv*: $\text{left-gpv } (\text{bind-gpv } \text{gpv } f) = \text{bind-gpv } (\text{left-gpv } \text{gpv}) (\text{left-gpv } \circ f)$
<proof>

lemma *inline1-left-gpv*:
 $\text{inline1 } (\lambda s \ q. \text{left-gpv } (\text{callee } s \ q)) \ \text{gpv } s =$
 $\text{map-spmf } (\text{map-sum } \text{id } (\text{map-prod } \text{Inl } (\text{map-prod } \text{left-rpv } \text{id}))) (\text{inline1 } \text{callee } \text{gpv } s)$
<proof>

lemma *left-gpv-inline*: $\text{left-gpv } (\text{inline } \text{callee } \text{gpv } s) = \text{inline } (\lambda s \ q. \text{left-gpv } (\text{callee } s \ q)) \ \text{gpv } s$
<proof>

lemma *right-gpv-bind-gpv*: $\text{right-gpv } (\text{bind-gpv } \text{gpv } f) = \text{bind-gpv } (\text{right-gpv } \text{gpv}) (\text{right-gpv } \circ f)$
<proof>

lemma *inline1-right-gpv*:

$inline1 (\lambda s q. right-gpv (callee s q))\ gpv s =$
 $map-spmf (map-sum id (map-prod Inr (map-prod right-rpv id))) (inline1 callee$
 $gpv s)$
<proof>

lemma *right-gpv-inline*: $right-gpv (inline callee gpv s) = inline (\lambda s q. right-gpv$
 $(callee s q))\ gpv s$
<proof>

lemma *WT-gpv-left-gpv*: $\mathcal{I}1 \vdash_g gpv \checkmark \implies \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_g left-gpv gpv \checkmark$
<proof>

lemma *WT-gpv-right-gpv*: $\mathcal{I}2 \vdash_g gpv \checkmark \implies \mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2 \vdash_g right-gpv gpv \checkmark$
<proof>

lemma *results-gpv-left-gpv [simp]*: $results-gpv (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) (left-gpv gpv) = re-$
 $sults-gpv \mathcal{I}1 gpv$
(is ?lhs = ?rhs)
<proof>

lemma *results-gpv-right-gpv [simp]*: $results-gpv (\mathcal{I}1 \oplus_{\mathcal{I}} \mathcal{I}2) (right-gpv gpv) = re-$
 $sults-gpv \mathcal{I}2 gpv$
(is ?lhs = ?rhs)
<proof>

lemma *left-gpv-Fail [simp]*: $left-gpv Fail = Fail$
<proof>

lemma *right-gpv-Fail [simp]*: $right-gpv Fail = Fail$
<proof>

lemma *rsuml-lsumr-left-gpv-left-gpv*: $map-gpv' id rsuml lsumr (left-gpv (left-gpv$
 $gpv)) = left-gpv gpv$
<proof>

lemma *rsuml-lsumr-left-gpv-right-gpv*: $map-gpv' id rsuml lsumr (left-gpv (right-gpv$
 $gpv)) = right-gpv (left-gpv gpv)$
<proof>

lemma *rsuml-lsumr-right-gpv*: $map-gpv' id rsuml lsumr (right-gpv gpv) = right-gpv$
 $(right-gpv gpv)$
<proof>

lemma *map-gpv'-map-gpv-swap*:

$map-gpv' f g h (map-gpv f' id gpv) = map-gpv (f \circ f') id (map-gpv' id g h gpv)$
<proof>

lemma *lsumr-rsuml-left-gpv*: $\text{map-gpv}' \text{ id } \text{lsumr } \text{rsuml } (\text{left-gpv } \text{gpv}) = \text{left-gpv } (\text{left-gpv } \text{gpv})$
 ⟨proof⟩

lemma *lsumr-rsuml-right-gpv-left-gpv*:
 $\text{map-gpv}' \text{ id } \text{lsumr } \text{rsuml } (\text{right-gpv } (\text{left-gpv } \text{gpv})) = \text{left-gpv } (\text{right-gpv } \text{gpv})$
 ⟨proof⟩

lemma *lsumr-rsuml-right-gpv-right-gpv*:
 $\text{map-gpv}' \text{ id } \text{lsumr } \text{rsuml } (\text{right-gpv } (\text{right-gpv } \text{gpv})) = \text{right-gpv } \text{gpv}$
 ⟨proof⟩

lemma *in-set-spmf-extend-state-oracle* [simp]:
 $x \in \text{set-spmf } (\text{extend-state-oracle } \text{oracle } s \ y) \longleftrightarrow$
 $\text{fst } (\text{snd } x) = \text{fst } s \wedge (\text{fst } x, \text{snd } (\text{snd } x)) \in \text{set-spmf } (\text{oracle } (\text{snd } s) \ y)$
 ⟨proof⟩

lemma *extend-state-oracle-plus-oracle*:
 $\text{extend-state-oracle } (\text{plus-oracle } \text{oracle1 } \text{oracle2}) = \text{plus-oracle } (\text{extend-state-oracle } \text{oracle1}) (\text{extend-state-oracle } \text{oracle2})$
 ⟨proof⟩

definition *stateless-callee* :: $('a \Rightarrow ('b, 'out, 'in) \text{gpv}) \Rightarrow ('s \Rightarrow 'a \Rightarrow ('b \times 's, 'out, 'in) \text{gpv})$ **where**
 $\text{stateless-callee } \text{callee } s = \text{map-gpv } (\lambda b. (b, s)) \text{ id } \circ \text{callee}$

lemma *stateless-callee-parametric'*:
includes *lifting-syntax notes* [transfer-rule] = *map-gpv-parametric'* **shows**
 $((A \text{ ===> } \text{rel-gpv}'' B \ C \ R) \text{ ===> } S \text{ ===> } A \text{ ===> } (\text{rel-gpv}'' (\text{rel-prod } B \ S) \ C \ R))$
 $\text{stateless-callee } \text{stateless-callee}$
 ⟨proof⟩

lemma *id-oracle-alt-def*: $\text{id-oracle} = \text{stateless-callee } (\lambda x. \text{Pause } x \ \text{Done})$
 ⟨proof⟩

context

fixes *left* :: $'s1 \Rightarrow 'x1 \Rightarrow ('y1 \times 's1, 'call1, 'ret1) \text{gpv}$
and *right* :: $'s2 \Rightarrow 'x2 \Rightarrow ('y2 \times 's2, 'call2, 'ret2) \text{gpv}$
begin

fun *parallel-intercept* :: $'s1 \times 's2 \Rightarrow 'x1 + 'x2 \Rightarrow (('y1 + 'y2) \times ('s1 \times 's2), 'call1 + 'call2, 'ret1 + 'ret2) \text{gpv}$
where
 $\text{parallel-intercept } (s1, s2) (\text{Inl } a) = \text{left-gpv } (\text{map-gpv } (\text{map-prod } \text{Inl } (\lambda s1'. (s1', s2)))) \text{ id } (\text{left } s1 \ a)$
 $\mid \text{parallel-intercept } (s1, s2) (\text{Inr } b) = \text{right-gpv } (\text{map-gpv } (\text{map-prod } \text{Inr } (\text{Pair } s1' \ s2)))) \text{ id } (\text{right } s2 \ b)$

$s1$) id ($right$ $s2$ b)

end

end

6.4 Expectation transformer semantics

theory *GPV-Expectation* **imports**

Computational-Model

begin

lemma *le-enn2realI*: $\llbracket ennreal\ x \leq y; y = \top \implies x \leq 0 \rrbracket \implies x \leq enn2real\ y$
<proof>

lemma *enn2real-leD*: $\llbracket enn2real\ x < y; x \neq \top \rrbracket \implies x < ennreal\ y$
<proof>

lemma *ennreal-mult-le-self2I*: $\llbracket y > 0 \implies x \leq 1 \rrbracket \implies x * y \leq y$ **for** $x\ y :: ennreal$
<proof>

lemma *ennreal-leI*: $x \leq enn2real\ y \implies ennreal\ x \leq y$
<proof>

lemma *enn2real-INF*: $\llbracket A \neq \{\}; \forall x \in A. f\ x < \top \rrbracket \implies enn2real\ (INF\ x \in A. f\ x)$
 $= (INF\ x \in A. enn2real\ (f\ x))$
<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*: $monotone\ (rel\text{-}prod\ (\leq)\ (\leq))\ (\leq)\ (\lambda(x, y). x * y :: ennreal)$
<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*]:
 $\llbracket mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. f\ x);$
 $mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. g\ x) \rrbracket$
 $\implies mcont\ lub\ ord\ Sup\ (\leq)\ (\lambda x. f\ x * g\ x :: ennreal)$

<proof>

lemma *ereal-INF-cmult*: $0 < c \implies (\text{INF } i \in I. c * f i) = \text{ereal } c * (\text{INF } i \in I. f i)$
<proof>

lemma *ereal-INF-multc*: $0 < c \implies (\text{INF } i \in I. f i * c) = (\text{INF } i \in I. f i) * \text{ereal } c$
<proof>

lemma *INF-mult-left-ennreal*:

assumes $I = \{\}$ $\implies c \neq 0$

and $\llbracket c = \top; \exists i \in I. f i > 0 \rrbracket \implies \exists p > 0. \forall i \in I. f i \geq p$

shows $c * (\text{INF } i \in I. f i) = (\text{INF } i \in I. c * f i :: \text{ennreal})$

<proof> **including** *ennreal.lifting*

<proof>

lemma *pmf-map-spmf-None*: $\text{pmf } (\text{map-spmf } f p) \text{ None} = \text{pmf } p \text{ None}$

<proof>

lemma *nn-integral-try-spmf*:

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

<proof>

lemma *INF-UNION*: $(\text{INF } z \in \bigcup x \in A. B x. f z) = (\text{INF } x \in A. \text{INF } z \in B x. f z)$

for $f :: - \Rightarrow 'b :: \text{complete-lattice}$

<proof>

definition *nn-integral-spmf* :: $'a \text{ spmf} \Rightarrow ('a \Rightarrow \text{ennreal}) \Rightarrow \text{ennreal}$ **where**

$\text{nn-integral-spmf } p = \text{nn-integral } (\text{measure-spmf } p)$

lemma *nn-integral-spmf-parametric* [*transfer-rule*]:

includes *lifting-syntax*

shows $(\text{rel-spmf } A \text{ ===== } (A \text{ ===== } (=)) \text{ ===== } (=)) \text{ nn-integral-spmf nn-integral-spmf}$

<proof>

lemma *weight-spmf-mcont2mcont* [*THEN* *lfp.mcont2mcont*, *cont-intro*]:

shows *weight-spmf-mcont*: $\text{mcont } (\text{lub-spmf}) (\text{ord-spmf } (=)) \text{ Sup } (\leq) (\lambda p. \text{ennreal } (\text{weight-spmf } p))$

<proof>

lemma *mono2mono-nn-integral-spmf* [*THEN* *lfp.mono2mono*, *cont-intro*]:

shows *monotone-nn-integral-spmf*: $\text{monotone } (\text{ord-spmf } (=)) (\leq) (\lambda p. \text{integral}^N (\text{measure-spmf } p) f)$

<proof>

lemma *cont-nn-integral-spmf*:

$\text{cont } \text{lub-spmf } (\text{ord-spmf } (=)) \text{ Sup } (\leq) (\lambda p :: 'a \text{ spmf}. \text{nn-integral } (\text{measure-spmf } p) f)$

<proof>

lemma *mcont2mcont-nn-integral-spmf* [*THEN lfp.mcont2mcont, cont-intro*]:
 shows *mcont-nn-integral-spmf*:
 mcont lub-spmf (ord-spmf (=)) Sup (≤) (λp :: 'a spmf. nn-integral (measure-spmf p) f)
<proof>

lemma *nn-integral-mono2mono*:
 assumes $\bigwedge x. x \in \text{space } M \implies \text{monotone ord } (\leq) (\lambda f. F f x)$
 shows *monotone ord (≤) (λf. 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)
 $(\bigwedge x. \text{lfp.mono-body } (\lambda f. F f x)) \implies \text{lfp.mono-body } (\lambda f. \text{nn-integral } M (F f))$
<proof>

lemma *INF-mono-lfp* [*partial-function-mono*]:
 $(\bigwedge x. \text{lfp.mono-body } (\lambda f. F f x)) \implies \text{lfp.mono-body } (\lambda f. \text{INF } x \in M. F f x)$
<proof>

lemmas *parallel-fixp-induct-1-2 = parallel-fixp-induct-uc*
 of - - - λx. x - λx. x case-prod - curry,
 where $P = \lambda f g. P f (\text{curry } g),$
 unfolded case-prod-curry curry-case-prod curry-K,
 $OF - - - - - \text{refl refl}$
 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*]:
 $\llbracket \text{monotone (fun-ord } (\leq)) (\leq) f; \text{monotone (fun-ord } (\leq)) (\leq) g \rrbracket$
 $\implies \text{monotone (fun-ord } (\leq)) (\leq) (\lambda x. f x + g x :: \text{ennreal})$
<proof>

context
 fixes *fail :: ennreal*

```

and  $\mathcal{I} :: ('out, 'ret) \mathcal{I}$ 
and  $f :: 'a \Rightarrow ennreal$ 
notes [[function-internals]]
begin

partial-function (lfp-strong) expectation-gpv :: ('a, 'out, 'ret) gpv  $\Rightarrow$  ennreal where
  expectation-gpv gpv =
    ( $\int^+$  generat. (case generat of Pure  $x \Rightarrow f x$ 
      | IO out  $c \Rightarrow INF r \in responses-\mathcal{I} \ \mathcal{I} \ out.$  expectation-gpv ( $c \ r$ ))
     $\partial measure\text{-}spmf$  (the-gpv gpv))
    + fail * pmf (the-gpv gpv) None

lemma expectation-gpv-fixp-induct [case-names adm bottom step]:
  assumes lfp.admissible P
  and  $P (\lambda-. \ 0)$ 
  and  $\bigwedge expectation\text{-}gpv'. \ [\bigwedge gpv. \ expectation\text{-}gpv' \ gpv \leq \ expectation\text{-}gpv \ gpv; \ P$ 
expectation-gpv'  $\Longrightarrow$ 
     $P (\lambda gpv. (\int^+$  generat. (case generat of Pure  $x \Rightarrow f x$  | IO out  $c \Rightarrow INF$ 
 $r \in responses-\mathcal{I} \ \mathcal{I} \ out.$  expectation-gpv' ( $c \ r$ ))  $\partial measure\text{-}spmf$  (the-gpv gpv)) + fail
  * pmf (the-gpv gpv) None)
  shows  $P \ expectation\text{-}gpv$ 
   $\langle proof \rangle$ 

lemma expectation-gpv-Done [simp]: expectation-gpv (Done  $x$ ) =  $f \ x$ 
   $\langle proof \rangle$ 

lemma expectation-gpv-Fail [simp]: expectation-gpv Fail = fail
   $\langle proof \rangle$ 

lemma expectation-gpv-lift-spmf [simp]:
  expectation-gpv (lift-spmf  $p$ ) = ( $\int^+$   $x. \ f \ x \ \partial measure\text{-}spmf \ p$ ) + fail * pmf  $p \ None$ 
   $\langle proof \rangle$ 

lemma expectation-gpv-Pause [simp]:
  expectation-gpv (Pause out  $c$ ) = ( $INF \ r \in responses-\mathcal{I} \ \mathcal{I} \ out.$  expectation-gpv ( $c \ r$ ))
   $\langle proof \rangle$ 

end

context begin
private definition weight-spmf'  $p = \ weight\text{-}spmf \ p$ 
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  $((=) \Longrightarrow rel-\mathcal{I} \ C \ R \Longrightarrow (A \Longrightarrow (=)) \Longrightarrow rel\text{-}gpv'' \ A \ C \ R$ 
 $\Longrightarrow (=)) \ expectation\text{-}gpv \ expectation\text{-}gpv$ 
   $\langle proof \rangle$ 
end

```

lemma *expectation-gpv-parametric* [*transfer-rule*]:
includes *lifting-syntax*
shows $((=) \text{====>} \text{rel-}\mathcal{I} \ C \ (=) \text{====>} (A \text{====>} (=)) \text{====>} \text{rel-gpv} \ A \ C$
 $\text{====>} (=)) \text{expectation-gpv} \ \text{expectation-gpv}$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-cong*:
fixes *fail fail'*
assumes *fail: fail = fail'*
and $\mathcal{I}: \mathcal{I} = \mathcal{I}'$
and *gpv: gpv = gpv'*
and $f: \bigwedge x. x \in \text{results-gpv} \ \mathcal{I}' \ gpv' \implies f \ x = g \ x$
shows $\text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f \ gpv = \text{expectation-gpv} \ \text{fail}' \ \mathcal{I}' \ g \ gpv'$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-cong-fail*:
 $\text{colossless-gpv} \ \mathcal{I} \ gpv \implies \text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f \ gpv = \text{expectation-gpv} \ \text{fail}' \ \mathcal{I} \ f$
 $gpv \ \text{for} \ \text{fail}$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-mono*:
fixes *fail fail'*
assumes *fail: fail \leq fail'*
and $fg: f \leq g$
shows $\text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f \ gpv \leq \text{expectation-gpv} \ \text{fail}' \ \mathcal{I} \ g \ gpv$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-mono-strong*:
fixes *fail fail'*
assumes *fail: $\neg \text{colossless-gpv} \ \mathcal{I} \ gpv \implies \text{fail} \leq \text{fail}'$*
and $fg: \bigwedge x. x \in \text{results-gpv} \ \mathcal{I} \ gpv \implies f \ x \leq g \ x$
shows $\text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f \ gpv \leq \text{expectation-gpv} \ \text{fail}' \ \mathcal{I} \ g \ gpv$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-bind* [*simp*]:
fixes $\mathcal{I} \ f \ g \ \text{fail}$
defines $\text{expectation-gpv1} \equiv \text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f$
and $\text{expectation-gpv2} \equiv \text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ (\text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f \circ g)$
shows $\text{expectation-gpv1} \ (\text{bind-gpv} \ gpv \ g) = \text{expectation-gpv2} \ gpv \ (\text{is} \ ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-try-gpv* [*simp*]:
fixes $\text{fail} \ \mathcal{I} \ f \ gpv'$
defines $\text{expectation-gpv1} \equiv \text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f$
and $\text{expectation-gpv2} \equiv \text{expectation-gpv} \ (\text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f \ gpv') \ \mathcal{I} \ f$
shows $\text{expectation-gpv1} \ (\text{try-gpv} \ gpv \ gpv') = \text{expectation-gpv2} \ gpv$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-restrict-gpv*:

$\mathcal{I} \vdash_g \text{gpv } \checkmark \implies \text{expectation-gpv fail } \mathcal{I} f (\text{restrict-gpv } \mathcal{I} \text{ gpv}) = \text{expectation-gpv fail } \mathcal{I} f \text{ gpv}$ **for** *fail*
 ⟨proof⟩

lemma *expectation-gpv-const-le*: $\mathcal{I} \vdash_g \text{gpv } \checkmark \implies \text{expectation-gpv fail } \mathcal{I} (\lambda-. c) \text{ gpv} \leq \max c \text{ fail}$ **for** *fail*
 ⟨proof⟩

lemma *expectation-gpv-no-results*:

$\llbracket \text{results-gpv } \mathcal{I} \text{ gpv} = \{\}; \mathcal{I} \vdash_g \text{gpv } \checkmark \rrbracket \implies \text{expectation-gpv } 0 \mathcal{I} f \text{ gpv} = 0$
 ⟨proof⟩

lemma *expectation-gpv-cmult*:

fixes *fail*
assumes $0 < c$ **and** $c \neq \top$
shows $c * \text{expectation-gpv fail } \mathcal{I} f \text{ gpv} = \text{expectation-gpv } (c * \text{fail}) \mathcal{I} (\lambda x. c * f x) \text{ gpv}$
 ⟨proof⟩

lemma *expectation-gpv-le-exec-gpv*:

assumes *callee*: $\bigwedge s x. x \in \text{outs-}\mathcal{I} \mathcal{I} \implies \text{lossless-spmf } (\text{callee } s x)$
and *WT-gpv*: $\mathcal{I} \vdash_g \text{gpv } \checkmark$
and *WT-callee*: $\bigwedge s. \mathcal{I} \vdash_c \text{callee } s \checkmark$
shows $\text{expectation-gpv } 0 \mathcal{I} f \text{ gpv} \leq \int^+ (x, s). f x \partial \text{measure-spmf } (\text{exec-gpv callee gpv } s)$
 ⟨proof⟩

definition *weight-gpv* :: $(\text{'out}, \text{'ret}) \mathcal{I} \Rightarrow (\text{'a}, \text{'out}, \text{'ret}) \text{gpv} \Rightarrow \text{real}$

where $\text{weight-gpv } \mathcal{I} \text{ gpv} = \text{enn2real } (\text{expectation-gpv } 0 \mathcal{I} (\lambda-. 1) \text{ gpv})$

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

lemma *weight-gpv-Pause* [*simp*]:

$(\bigwedge r. r \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out} \implies \mathcal{I} \vdash_g c r \checkmark)$
 $\implies \text{weight-gpv } \mathcal{I} (\text{Pause out } c) = (\text{if } \text{out} \in \text{outs-}\mathcal{I} \mathcal{I} \text{ then } \text{INF } r \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out. } \text{weight-gpv } \mathcal{I} (c r) \text{ else } 0)$

⟨proof⟩

lemma *weight-gpv-nonneg*: $0 \leq \text{weight-gpv } \mathcal{I} \text{ gpv}$

⟨proof⟩

lemma *weight-gpv-le-1*: $\mathcal{I} \vdash_g \text{gpv } \checkmark \implies \text{weight-gpv } \mathcal{I} \text{ gpv} \leq 1$
 ⟨proof⟩

theorem *weight-exec-gpv*:
 assumes *callee*: $\bigwedge s x. x \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{lossless-spmf } (\text{callee } s \ x)$
 and *WT-gpv*: $\mathcal{I} \vdash_g \text{gpv } \checkmark$
 and *WT-callee*: $\bigwedge s. \mathcal{I} \vdash_c \text{callee } s \ \checkmark$
 shows $\text{weight-gpv } \mathcal{I} \ \text{gpv} \leq \text{weight-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s)$
 ⟨proof⟩

lemma (in *callee-invariant-on*) *weight-exec-gpv*:
 assumes *callee*: $\bigwedge s x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I \ s \rrbracket \implies \text{lossless-spmf } (\text{callee } s \ x)$
 and *WT-gpv*: $\mathcal{I} \vdash_g \text{gpv } \checkmark$
 and *I*: $I \ s$
 shows $\text{weight-gpv } \mathcal{I} \ \text{gpv} \leq \text{weight-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s)$
 including *lifting-syntax*
 ⟨proof⟩

6.5 Probabilistic termination

definition *pgen-lossless-gpv* :: $(c, r) \ \mathcal{I} \Rightarrow (a, c, r) \ \text{gpv} \Rightarrow \text{bool}$
 where $\text{pgen-lossless-gpv } \text{fail } \mathcal{I} \ \text{gpv} = (\text{expectation-gpv } \text{fail } \mathcal{I} \ (\lambda-. 1) \ \text{gpv} = 1)$ **for** *fail*

abbreviation *plossless-gpv* :: $(c, r) \ \mathcal{I} \Rightarrow (a, c, r) \ \text{gpv} \Rightarrow \text{bool}$
 where $\text{plossless-gpv} \equiv \text{pgen-lossless-gpv } 0$

abbreviation *pfinite-gpv* :: $(c, r) \ \mathcal{I} \Rightarrow (a, c, r) \ \text{gpv} \Rightarrow \text{bool}$
 where $\text{pfinite-gpv} \equiv \text{pgen-lossless-gpv } 1$

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

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

lemma *lossless-imp-plossless-gpv*:
 assumes *lossless-gpv*: $\mathcal{I} \ \text{gpv} \ \mathcal{I} \vdash_g \text{gpv } \checkmark$
 shows $\text{plossless-gpv } \mathcal{I} \ \text{gpv}$
 ⟨proof⟩

lemma *finite-imp-pfinite-gpv*:
 assumes *finite-gpv*: $\mathcal{I} \ \text{gpv} \ \mathcal{I} \vdash_g \text{gpv } \checkmark$
 shows $\text{pfinite-gpv } \mathcal{I} \ \text{gpv}$
 ⟨proof⟩

lemma *plossless-gpv-lossless-spmfD*:

assumes *lossless*: $p_{\text{lossless-gpv}} \mathcal{I} \text{ gpv}$
and *WT*: $\mathcal{I} \vdash_g \text{ gpv} \checkmark$
shows *lossless-spmf* (the-gpv gpv)
 $\langle \text{proof} \rangle$

lemma

shows *plossless-gpv-ContD*:
 $\llbracket p_{\text{lossless-gpv}} \mathcal{I} \text{ gpv}; \text{IO out } c \in \text{set-spmf} (\text{the-gpv gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out}; \mathcal{I} \vdash_g \text{ gpv} \checkmark \rrbracket$
 $\implies p_{\text{lossless-gpv}} \mathcal{I} (c \text{ input})$
and *pfinite-gpv-ContD*:
 $\llbracket p_{\text{finite-gpv}} \mathcal{I} \text{ gpv}; \text{IO out } c \in \text{set-spmf} (\text{the-gpv gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out}; \mathcal{I} \vdash_g \text{ gpv} \checkmark \rrbracket$
 $\implies p_{\text{finite-gpv}} \mathcal{I} (c \text{ input})$
 $\langle \text{proof} \rangle$

lemma *plossless-iff-colossless-pfinite*:

assumes *WT*: $\mathcal{I} \vdash_g \text{ gpv} \checkmark$
shows $p_{\text{lossless-gpv}} \mathcal{I} \text{ gpv} \longleftrightarrow \text{colossless-gpv } \mathcal{I} \text{ gpv} \wedge p_{\text{finite-gpv}} \mathcal{I} \text{ gpv}$
 $\langle \text{proof} \rangle$

lemma *pgen-lossless-gpv-Done* [*simp*]: $p_{\text{gen-lossless-gpv fail } \mathcal{I} (\text{Done } x)} \text{ for fail}$
 $\langle \text{proof} \rangle$

lemma *pgen-lossless-gpv-Fail* [*simp*]: $p_{\text{gen-lossless-gpv fail } \mathcal{I} \text{ Fail}} \longleftrightarrow \text{fail} = 1 \text{ for fail}$
 $\langle \text{proof} \rangle$

lemma *pgen-lossless-gpv-PauseI* [*simp, intro!*]:

$\llbracket \text{out} \in \text{outs-}\mathcal{I} \mathcal{I}; \bigwedge r. r \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out} \implies p_{\text{gen-lossless-gpv fail } \mathcal{I} (c \text{ r})} \rrbracket$
 $\implies p_{\text{gen-lossless-gpv fail } \mathcal{I} (\text{Pause out } c)} \text{ for fail}$
 $\langle \text{proof} \rangle$

lemma *pgen-lossless-gpv-bindI* [*simp, intro!*]:

$\llbracket p_{\text{gen-lossless-gpv fail } \mathcal{I} \text{ gpv}}; \bigwedge x. x \in \text{results-gpv } \mathcal{I} \text{ gpv} \implies p_{\text{gen-lossless-gpv fail } \mathcal{I} (f \text{ x})} \rrbracket$
 $\implies p_{\text{gen-lossless-gpv fail } \mathcal{I} (\text{bind-gpv gpv } f)} \text{ for fail}$
 $\langle \text{proof} \rangle$

lemma *pgen-lossless-gpv-lift-spmf* [*simp*]:

$p_{\text{gen-lossless-gpv fail } \mathcal{I} (\text{lift-spmf } p)} \longleftrightarrow \text{lossless-spmf } p \vee \text{fail} = 1 \text{ for fail}$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-top-pfinite*:

assumes $p_{\text{finite-gpv}} \mathcal{I} \text{ gpv}$
shows $\text{expectation-gpv } \top \mathcal{I} (\lambda \cdot \top) \text{ gpv} = \top$
 $\langle \text{proof} \rangle$

lemma *pfinite-INF-le-expectation-gpv*:

fixes $fail \mathcal{I} gpv f$
defines $c \equiv \min (INF\ x \in results-gpv \mathcal{I} gpv. f\ x) fail$
assumes $fin: pfinite-gpv \mathcal{I} gpv$
shows $c \leq expectation-gpv\ fail\ \mathcal{I}\ f\ gpv$ (**is** $?lhs \leq ?rhs$)
 $\langle proof \rangle$

lemma *plossless-INF-le-expectation-gpv*:
fixes $fail$
assumes $plossless-gpv \mathcal{I} gpv$ **and** $\mathcal{I} \vdash_g gpv \checkmark$
shows $(INF\ x \in results-gpv \mathcal{I} gpv. f\ x) \leq expectation-gpv\ fail\ \mathcal{I}\ f\ gpv$ (**is** $?lhs \leq ?rhs$)
 $\langle proof \rangle$

lemma *expectation-gpv-le-inline*:
fixes \mathcal{I}'
defines $expectation-gpv2 \equiv expectation-gpv\ 0\ \mathcal{I}'$
assumes $callee: \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies plossless-gpv\ \mathcal{I}' (callee\ s\ x)$
and $callee': \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies results-gpv\ \mathcal{I}' (callee\ s\ x) \subseteq responses-\mathcal{I}\ \mathcal{I}$
 $x \times UNIV$
and $WT-gpv: \mathcal{I} \vdash_g gpv \checkmark$
and $WT-callee: \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies \mathcal{I}' \vdash_g callee\ s\ x \checkmark$
shows $expectation-gpv\ 0\ \mathcal{I}\ f\ gpv \leq expectation-gpv2\ (\lambda(x, s). f\ x)$ (*inline callee gpv s*)
 $\langle proof \rangle$

lemma *plossless-inline*:
assumes $lossless: plossless-gpv \mathcal{I} gpv$
and $WT: \mathcal{I} \vdash_g gpv \checkmark$
and $callee: \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies plossless-gpv\ \mathcal{I}' (callee\ s\ x)$
and $callee': \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies results-gpv\ \mathcal{I}' (callee\ s\ x) \subseteq responses-\mathcal{I}\ \mathcal{I}$
 $x \times UNIV$
and $WT-callee: \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies \mathcal{I}' \vdash_g callee\ s\ x \checkmark$
shows $plossless-gpv\ \mathcal{I}'$ (*inline callee gpv s*)
 $\langle proof \rangle$

lemma *plossless-exec-gpv*:
assumes $lossless: plossless-gpv \mathcal{I} gpv$
and $WT: \mathcal{I} \vdash_g gpv \checkmark$
and $callee: \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies lossless-spmf (callee\ s\ x)$
and $callee': \bigwedge s\ x. x \in outs-\mathcal{I}\ \mathcal{I} \implies set-spmf (callee\ s\ x) \subseteq responses-\mathcal{I}\ \mathcal{I} \times UNIV$
shows $lossless-spmf (exec-gpv\ callee\ gpv\ s)$
 $\langle proof \rangle$

lemma *expectation-gpv-I-mono*:
defines $expectation-gpv' \equiv expectation-gpv$
assumes $le: \mathcal{I} \leq \mathcal{I}'$
and $WT: \mathcal{I} \vdash_g gpv \checkmark$

shows $\text{expectation-gpv fail } \mathcal{I} f \text{ gpv} \leq \text{expectation-gpv}' \text{ fail } \mathcal{I}' f \text{ gpv}$
 ⟨proof⟩

lemma *pgen-lossless-gpv-mono*:

assumes *: $\text{pgen-lossless-gpv fail } \mathcal{I} \text{ gpv}$
and $le: \mathcal{I} \leq \mathcal{I}'$
and $WT: \mathcal{I} \vdash_g \text{ gpv } \checkmark$
and $\text{fail}: \text{fail} \leq 1$
shows $\text{pgen-lossless-gpv fail } \mathcal{I}' \text{ gpv}$
 ⟨proof⟩

lemma *plossless-gpv-mono*:

$\llbracket \text{plossless-gpv } \mathcal{I} \text{ gpv}; \mathcal{I} \leq \mathcal{I}'; \mathcal{I} \vdash_g \text{ gpv } \checkmark \rrbracket \implies \text{plossless-gpv } \mathcal{I}' \text{ gpv}$
 ⟨proof⟩

lemma *pfinite-gpv-mono*:

$\llbracket \text{pfinite-gpv } \mathcal{I} \text{ gpv}; \mathcal{I} \leq \mathcal{I}'; \mathcal{I} \vdash_g \text{ gpv } \checkmark \rrbracket \implies \text{pfinite-gpv } \mathcal{I}' \text{ gpv}$
 ⟨proof⟩

lemma *pgen-lossless-gpv-parametric'*: **includes** *lifting-syntax* **shows**

$((=) \implies \text{rel-}\mathcal{I} \ C \ R \implies \text{rel-gpv}'' \ A \ C \ R \implies (=)) \text{ pgen-lossless-gpv}$
 pgen-lossless-gpv
 ⟨proof⟩

lemma *pgen-lossless-gpv-parametric*: **includes** *lifting-syntax* **shows**

$((=) \implies \text{rel-}\mathcal{I} \ C \ (=) \implies \text{rel-gpv} \ A \ C \implies (=)) \text{ pgen-lossless-gpv}$
 pgen-lossless-gpv
 ⟨proof⟩

lemma *pgen-lossless-gpv-map-gpv-id* [*simp*]:

$\text{pgen-lossless-gpv fail } \mathcal{I} (\text{map-gpv } f \ \text{id } \text{gpv}) = \text{pgen-lossless-gpv fail } \mathcal{I} \text{ gpv}$
 ⟨proof⟩

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

lemma *expectation-gpv-le-inline*:

defines $\text{expectation-gpv2} \equiv \text{expectation-gpv } 0 \ \mathcal{I}'$
assumes $\text{callee}: \bigwedge s \ x. \llbracket x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I \ s \rrbracket \implies \text{plossless-gpv } \mathcal{I}' (\text{callee } s \ x)$
and $WT\text{-gpv}: \mathcal{I} \vdash_g \text{ gpv } \checkmark$
and $I: I \ s$
shows $\text{expectation-gpv } 0 \ \mathcal{I} f \text{ gpv} \leq \text{expectation-gpv2 } (\lambda(x, s). f \ x)$ (*inline callee*
 $\text{gpv } s$)
 ⟨proof⟩

lemma *plossless-inline*:

assumes $\text{lossless}: \text{plossless-gpv } \mathcal{I} \text{ gpv}$
and $WT: \mathcal{I} \vdash_g \text{ gpv } \checkmark$
and $\text{callee}: \bigwedge s \ x. \llbracket I \ s; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{plossless-gpv } \mathcal{I}' (\text{callee } s \ x)$
and $I: I \ s$

shows *plossless-gpv* \mathcal{I}' (*inline callee gpv s*)
 ⟨*proof*⟩

end

lemma *expectation-left-gpv* [*simp*]:
 $expectation-gpv\ fail\ (\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}')\ f\ (left-gpv\ gpv) = expectation-gpv\ fail\ \mathcal{I}\ f\ gpv$
 ⟨*proof*⟩

lemma *expectation-right-gpv* [*simp*]:
 $expectation-gpv\ fail\ (\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}')\ f\ (right-gpv\ gpv) = expectation-gpv\ fail\ \mathcal{I}'\ f\ gpv$
 ⟨*proof*⟩

lemma *pgen-lossless-left-gpv* [*simp*]: *pgen-lossless-gpv* $fail\ (\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}')\ (left-gpv\ gpv)$
 $= pgen-lossless-gpv\ fail\ \mathcal{I}\ gpv$
 ⟨*proof*⟩

lemma *pgen-lossless-right-gpv* [*simp*]: *pgen-lossless-gpv* $fail\ (\mathcal{I} \oplus_{\mathcal{I}} \mathcal{I}')\ (right-gpv\ gpv)$
 $= pgen-lossless-gpv\ fail\ \mathcal{I}'\ gpv$
 ⟨*proof*⟩

lemma (**in** *raw-converter-invariant*) *expectation-gpv-le-inline-invariant*:
defines $expectation-gpv2 \equiv expectation-gpv\ 0\ \mathcal{I}'$
assumes *callee*: $\bigwedge s\ x.\ \llbracket x \in outs\ \mathcal{I}\ \mathcal{I};\ I\ s \rrbracket \implies plossless-gpv\ \mathcal{I}'\ (callee\ s\ x)$
and *WT-gpv*: $\mathcal{I} \vdash_g\ gpv\ \checkmark$
and *I*: $I\ s$
shows $expectation-gpv\ 0\ \mathcal{I}\ f\ gpv \leq expectation-gpv2\ (\lambda(x, s). f\ x)$ (*inline callee gpv s*)
 ⟨*proof*⟩

lemma (**in** *raw-converter-invariant*) *plossless-inline-invariant*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}\ gpv$
and *WT*: $\mathcal{I} \vdash_g\ gpv\ \checkmark$
and *callee*: $\bigwedge s\ x.\ \llbracket x \in outs\ \mathcal{I}\ \mathcal{I};\ I\ s \rrbracket \implies plossless-gpv\ \mathcal{I}'\ (callee\ s\ x)$
and *I*: $I\ s$
shows *plossless-gpv* \mathcal{I}' (*inline callee gpv s*)
 ⟨*proof*⟩

context *callee-invariant-on* **begin**

lemma *raw-converter-invariant*: *raw-converter-invariant* $\mathcal{I}\ \mathcal{I}'\ (\lambda s\ x.\ lift\ spmf\ (callee\ s\ x))\ I$
 ⟨*proof*⟩

lemma (**in** *callee-invariant-on*) *plossless-exec-gpv*:
assumes *lossless*: *plossless-gpv* $\mathcal{I}\ gpv$
and *WT*: $\mathcal{I} \vdash_g\ gpv\ \checkmark$
and *callee*: $\bigwedge s\ x.\ \llbracket x \in outs\ \mathcal{I}\ \mathcal{I};\ I\ s \rrbracket \implies lossless-spmf\ (callee\ s\ x)$
and *I*: $I\ s$

shows *lossless-spmf* (*exec-gpv callee gpv s*)
 ⟨*proof*⟩

end

lemma *expectation-gpv-mk-lossless-gpv*:

fixes $\mathcal{I} \ y$
defines $rhs \equiv \text{expectation-gpv } 0 \ \mathcal{I} \ (\lambda-. \ y)$
assumes $WT: \mathcal{I}' \vdash_g \text{ gpv } \checkmark$
and $\text{outs-}\mathcal{I} \ \mathcal{I} = \text{outs-}\mathcal{I} \ \mathcal{I}'$
shows $\text{expectation-gpv } 0 \ \mathcal{I}' \ (\lambda-. \ y) \text{ gpv} \leq rhs \ (\text{mk-lossless-gpv} \ (\text{responses-}\mathcal{I} \ \mathcal{I}') \ x \text{ gpv})$
 ⟨*proof*⟩

lemma *plossless-gpv-mk-lossless-gpv*:

assumes *plossless-gpv* $\mathcal{I} \ \text{gpv}$
and $\mathcal{I} \vdash_g \text{ gpv } \checkmark$
and $\text{outs-}\mathcal{I} \ \mathcal{I} = \text{outs-}\mathcal{I} \ \mathcal{I}'$
shows *plossless-gpv* $\mathcal{I}' \ (\text{mk-lossless-gpv} \ (\text{responses-}\mathcal{I} \ \mathcal{I}) \ x \text{ gpv})$
 ⟨*proof*⟩

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

assumes $\mathcal{I} \vdash_g \text{ gpv } \checkmark$
and $I \ s$
shows $\text{exec-gpv callee} \ (\text{mk-lossless-gpv} \ (\text{responses-}\mathcal{I} \ \mathcal{I}) \ x \text{ gpv}) \ s = \text{exec-gpv callee} \ \text{gpv} \ s$
 ⟨*proof*⟩

lemma *expectation-gpv-map-gpv'* [*simp*]:

$\text{expectation-gpv fail } \mathcal{I} \ f \ (\text{map-gpv}' \ g \ h \ k \ \text{gpv}) =$
 $\text{expectation-gpv fail} \ (\text{map-}\mathcal{I} \ h \ k \ \mathcal{I}) \ (f \circ g) \ \text{gpv}$
 ⟨*proof*⟩

lemma *plossless-gpv-map-gpv'* [*simp*]:

$\text{pgen-lossless-gpv } b \ \mathcal{I} \ (\text{map-gpv}' \ f \ g \ h \ \text{gpv}) \longleftrightarrow \text{pgen-lossless-gpv } b \ (\text{map-}\mathcal{I} \ g \ h \ \mathcal{I}) \ \text{gpv}$
 ⟨*proof*⟩

end

theory *GPV-Bisim* **imports**

GPV-Expectation

begin

6.6 Bisimulation for oracles

Bisimulation is a consequence of parametricity

lemma *exec-gpv-oracle-bisim'*:

assumes *: $X\ s1\ s2$
and *bisim*: $\bigwedge s1\ s2\ x.\ X\ s1\ s2 \implies \text{rel-spmf } (\lambda(a, s1') (b, s2')).\ a = b \wedge X\ s1'\ s2'$
 $(\text{oracle1 } s1\ x)\ (\text{oracle2 } s2\ x)$
shows $\text{rel-spmf } (\lambda(a, s1') (b, s2')).\ a = b \wedge X\ s1'\ s2'$ (*exec-gpv oracle1 gpv s1*)
(*exec-gpv oracle2 gpv s2*)
 $\langle \text{proof} \rangle$

lemma *exec-gpv-oracle-bisim*:

assumes *: $X\ s1\ s2$
and *bisim*: $\bigwedge s1\ s2\ x.\ X\ s1\ s2 \implies \text{rel-spmf } (\lambda(a, s1') (b, s2')).\ a = b \wedge X\ s1'\ s2'$
 $(\text{oracle1 } s1\ x)\ (\text{oracle2 } s2\ x)$
and *R*: $\bigwedge x\ s1'\ s2'.\ \llbracket X\ s1'\ s2'; (x, s1') \in \text{set-spmf } (\text{exec-gpv oracle1 gpv s1});$
 $(x, s2') \in \text{set-spmf } (\text{exec-gpv oracle2 gpv s2}) \rrbracket \implies R\ (x, s1')\ (x, s2')$
shows $\text{rel-spmf } R\ (\text{exec-gpv oracle1 gpv s1})\ (\text{exec-gpv oracle2 gpv s2})$
 $\langle \text{proof} \rangle$

lemma *run-gpv-oracle-bisim*:

assumes $X\ s1\ s2$
and $\bigwedge s1\ s2\ x.\ X\ s1\ s2 \implies \text{rel-spmf } (\lambda(a, s1') (b, s2')).\ a = b \wedge X\ s1'\ s2'$
 $(\text{oracle1 } s1\ x)\ (\text{oracle2 } s2\ x)$
shows $\text{run-gpv oracle1 gpv s1} = \text{run-gpv oracle2 gpv s2}$
 $\langle \text{proof} \rangle$

context

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

begin

partial-function (*spmf*) *exec-until-bad* :: $('x, 'a, 'b)\ \text{gpv} \Rightarrow 's1 \Rightarrow 's2 \Rightarrow (('x \times 's1) \times ('x \times 's2))\ \text{spmf}$

where

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

lemma *exec-until-bad-fixp-induct* [*case-names adm bottom step*]:

assumes *ccpo.admissible* (*fun-lub lub-spmf*) (*fun-ord* (*ord-spmf* (=))) ($\lambda f.\ P$
 $(\lambda \text{gpv } s1\ s2.\ f\ ((\text{gpv}, s1), s2)))$)
and $P\ (\lambda - -.\ \text{return-pmf None})$

and $\bigwedge \text{exec-until-bad}'. P \text{ exec-until-bad}' \implies$
 $P (\lambda \text{gpv } s1 \ s2. \text{ if bad1 } s1 \vee \text{ bad2 } s2 \text{ then pair-spmf } (\text{exec-gpv oracle1 gpv } s1)$
 $(\text{exec-gpv oracle2 gpv } s2)$
 $\text{ else bind-spmf } (\text{the-gpv gpv}) (\lambda \text{generat.}$
 $\text{ case generat of Pure } x \Rightarrow \text{ return-spmf } ((x, s1), (x, s2))$
 $| \text{ IO out } f \Rightarrow \text{ bind-spmf } (\text{joint-oracle } (s1, s2) \text{ out}) (\lambda((x, s1'), (y, s2')).$
 $\text{ if bad1 } s1' \vee \text{ bad2 } s2' \text{ then pair-spmf } (\text{exec-gpv oracle1 } (f \ x) \ s1') (\text{exec-gpv}$
 $\text{ oracle2 } (f \ y) \ s2')$
 $\text{ else exec-until-bad}' (f \ x) \ s1' \ s2'))$
shows $P \text{ exec-until-bad}$
 $\langle \text{proof} \rangle$

end

lemma *exec-gpv-oracle-bisim-bad-plossless*:

fixes $s1 :: 's1$ **and** $s2 :: 's2$ **and** $X :: 's1 \Rightarrow 's2 \Rightarrow \text{bool}$
and $\text{oracle1} :: 's1 \Rightarrow 'a \Rightarrow ('b \times 's1) \text{ spmf}$
and $\text{oracle2} :: 's2 \Rightarrow 'a \Rightarrow ('b \times 's2) \text{ spmf}$
assumes $*$: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and bad : $\text{bad1 } s1 = \text{bad2 } s2$
and bisim : $\bigwedge s1 \ s2 \ x. \llbracket X \ s1 \ s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{rel-spmf } (\lambda(a, s1') (b, s2').$
 $\text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then } X\text{-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2'))$
 $(\text{oracle1 } s1 \ x) (\text{oracle2 } s2 \ x)$
and bad-sticky1 : $\bigwedge s2. \text{bad2 } s2 \implies \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge$
 $X\text{-bad } s1 \ s2) \ \mathcal{I}$
and bad-sticky2 : $\bigwedge s1. \text{bad1 } s1 \implies \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge$
 $X\text{-bad } s1 \ s2) \ \mathcal{I}$
and lossless1 : $\bigwedge s1 \ x. \llbracket \text{bad1 } s1; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{lossless-spmf } (\text{oracle1 } s1 \ x)$
and lossless2 : $\bigwedge s2 \ x. \llbracket \text{bad2 } s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{lossless-spmf } (\text{oracle2 } s2 \ x)$
and lossless : *plossless-gpv* $\mathcal{I} \ \text{gpv}$
and WT-oracle1 : $\bigwedge s1. \mathcal{I} \vdash c \ \text{oracle1 } s1 \ \checkmark$
and WT-oracle2 : $\bigwedge s2. \mathcal{I} \vdash c \ \text{oracle2 } s2 \ \checkmark$
and WT-gpv : $\mathcal{I} \vdash g \ \text{gpv} \ \checkmark$
shows $\text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then } X\text{-bad}$
 $s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2')) (\text{exec-gpv oracle1 gpv } s1) (\text{exec-gpv oracle2 gpv}$
 $s2)$
 $(\text{is rel-spmf } ?R \ ?p \ ?q)$
 $\langle \text{proof} \rangle$

lemma *exec-gpv-oracle-bisim-bad'*:

fixes $s1 :: 's1$ **and** $s2 :: 's2$ **and** $X :: 's1 \Rightarrow 's2 \Rightarrow \text{bool}$
and $\text{oracle1} :: 's1 \Rightarrow 'a \Rightarrow ('b \times 's1) \text{ spmf}$
and $\text{oracle2} :: 's2 \Rightarrow 'a \Rightarrow ('b \times 's2) \text{ spmf}$
assumes $*$: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and bad : $\text{bad1 } s1 = \text{bad2 } s2$
and bisim : $\bigwedge s1 \ s2 \ x. \llbracket X \ s1 \ s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{rel-spmf } (\lambda(a, s1') (b, s2').$
 $\text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then } X\text{-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2'))$
 $(\text{oracle1 } s1 \ x) (\text{oracle2 } s2 \ x)$
and bad-sticky1 : $\bigwedge s2. \text{bad2 } s2 \implies \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge$

$X\text{-bad } s1 \ s2) \mathcal{I}$
and $\text{bad-sticky2}: \bigwedge s1. \text{bad1 } s1 \implies \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge X\text{-bad } s1 \ s2) \mathcal{I}$
and $\text{lossless1}: \bigwedge s1 \ x. \llbracket \text{bad1 } s1; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{lossless-spmf } (\text{oracle1 } s1 \ x)$
and $\text{lossless2}: \bigwedge s2 \ x. \llbracket \text{bad2 } s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{lossless-spmf } (\text{oracle2 } s2 \ x)$
and $\text{lossless}: \text{lossless-gpv } \mathcal{I} \ \text{gpv}$
and $\text{WT-oracle1}: \bigwedge s1. \mathcal{I} \vdash c \ \text{oracle1 } s1 \ \checkmark$
and $\text{WT-oracle2}: \bigwedge s2. \mathcal{I} \vdash c \ \text{oracle2 } s2 \ \checkmark$
and $\text{WT-gpv}: \mathcal{I} \vdash g \ \text{gpv} \ \checkmark$
shows $\text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if } \text{bad2 } s2' \text{ then } X\text{-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2')) (\text{exec-gpv } \text{oracle1 } \ \text{gpv } \ s1) (\text{exec-gpv } \text{oracle2 } \ \text{gpv } \ s2)$
 $\langle \text{proof} \rangle$

lemma $\text{exec-gpv-oracle-bisim-bad-invariant}$:

fixes $s1 :: 's1$ **and** $s2 :: 's2$ **and** $X :: 's1 \Rightarrow 's2 \Rightarrow \text{bool}$ **and** $I1 :: 's1 \Rightarrow \text{bool}$
and $I2 :: 's2 \Rightarrow \text{bool}$
and $\text{oracle1} :: 's1 \Rightarrow 'a \Rightarrow ('b \times 's1) \ \text{spmf}$
and $\text{oracle2} :: 's2 \Rightarrow 'a \Rightarrow ('b \times 's2) \ \text{spmf}$
assumes $*$: $\text{if } \text{bad2 } s2 \text{ then } X\text{-bad } s1 \ s2 \text{ else } X \ s1 \ s2$
and $\text{bad}: \text{bad1 } s1 = \text{bad2 } s2$
and $\text{bisim}: \bigwedge s1 \ s2 \ x. \llbracket X \ s1 \ s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I}; I1 \ s1; I2 \ s2 \rrbracket \implies \text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if } \text{bad2 } s2' \text{ then } X\text{-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2')) (\text{oracle1 } s1 \ x) (\text{oracle2 } s2 \ x)$
and $\text{bad-sticky1}: \bigwedge s2. \llbracket \text{bad2 } s2; I2 \ s2 \rrbracket \implies \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge X\text{-bad } s1 \ s2) \ \mathcal{I}$
and $\text{bad-sticky2}: \bigwedge s1. \llbracket \text{bad1 } s1; I1 \ s1 \rrbracket \implies \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge X\text{-bad } s1 \ s2) \ \mathcal{I}$
and $\text{lossless1}: \bigwedge s1 \ x. \llbracket \text{bad1 } s1; I1 \ s1; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{lossless-spmf } (\text{oracle1 } s1 \ x)$
and $\text{lossless2}: \bigwedge s2 \ x. \llbracket \text{bad2 } s2; I2 \ s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{lossless-spmf } (\text{oracle2 } s2 \ x)$
and $\text{lossless}: \text{lossless-gpv } \mathcal{I} \ \text{gpv}$
and $\text{WT-gpv}: \mathcal{I} \vdash g \ \text{gpv} \ \checkmark$
and $I1: \text{callee-invariant-on oracle1 } I1 \ \mathcal{I}$
and $I2: \text{callee-invariant-on oracle2 } I2 \ \mathcal{I}$
and $s1: I1 \ s1$
and $s2: I2 \ s2$
shows $\text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if } \text{bad2 } s2' \text{ then } X\text{-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2')) (\text{exec-gpv } \text{oracle1 } \ \text{gpv } \ s1) (\text{exec-gpv } \text{oracle2 } \ \text{gpv } \ s2)$
including lifting-syntax
 $\langle \text{proof} \rangle$

lemma $\text{exec-gpv-oracle-bisim-bad}$:

assumes $*$: $\text{if } \text{bad2 } s2 \text{ then } X\text{-bad } s1 \ s2 \text{ else } X \ s1 \ s2$
and $\text{bad}: \text{bad1 } s1 = \text{bad2 } s2$
and $\text{bisim}: \bigwedge s1 \ s2 \ x. X \ s1 \ s2 \implies \text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if } \text{bad2 } s2' \text{ then } X\text{-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2')) (\text{oracle1 } s1 \ x) (\text{oracle2 } s2 \ x)$

$s2\ x)$
and *bad-sticky1*: $\bigwedge s2. \text{bad2 } s2 \implies \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge X\text{-bad } s1\ s2) \mathcal{I}$
and *bad-sticky2*: $\bigwedge s1. \text{bad1 } s1 \implies \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge X\text{-bad } s1\ s2) \mathcal{I}$
and *lossless1*: $\bigwedge s1\ x. \text{bad1 } s1 \implies \text{lossless-spmf } (\text{oracle1 } s1\ x)$
and *lossless2*: $\bigwedge s2\ x. \text{bad2 } s2 \implies \text{lossless-spmf } (\text{oracle2 } s2\ x)$
and *lossless*: $\text{lossless-gpv } \mathcal{I}\ \text{gpv}$
and *WT-oracle1*: $\bigwedge s1. \mathcal{I} \vdash c\ \text{oracle1 } s1\ \checkmark$
and *WT-oracle2*: $\bigwedge s2. \mathcal{I} \vdash c\ \text{oracle2 } s2\ \checkmark$
and *WT-gpv*: $\mathcal{I} \vdash g\ \text{gpv}\ \checkmark$
and *R*: $\bigwedge a\ s1\ b\ s2. \llbracket \text{bad1 } s1 = \text{bad2 } s2; \neg \text{bad2 } s2 \implies a = b \wedge X\ s1\ s2; \text{bad2 } s2 \implies X\text{-bad } s1\ s2 \rrbracket \implies R\ (a, s1)\ (b, s2)$
shows *rel-spmf* $R\ (\text{exec-gpv } \text{oracle1}\ \text{gpv } s1)\ (\text{exec-gpv } \text{oracle2}\ \text{gpv } s2)$
<proof>

lemma *exec-gpv-oracle-bisim-bad-full*:

assumes $X\ s1\ s2$
and $\text{bad1 } s1 = \text{bad2 } s2$
and $\bigwedge s1\ s2\ x. X\ s1\ s2 \implies \text{rel-spmf } (\lambda(a, s1'). (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\neg \text{bad2 } s2' \longrightarrow a = b \wedge X\ s1'\ s2'))\ (\text{oracle1 } s1\ x)\ (\text{oracle2 } s2\ x)$
and *callee-invariant* $\text{oracle1 } \text{bad1}$
and *callee-invariant* $\text{oracle2 } \text{bad2}$
and $\bigwedge s1\ x. \text{bad1 } s1 \implies \text{lossless-spmf } (\text{oracle1 } s1\ x)$
and $\bigwedge s2\ x. \text{bad2 } s2 \implies \text{lossless-spmf } (\text{oracle2 } s2\ x)$
and *lossless-gpv* $\mathcal{I}\text{-full}\ \text{gpv}$
and *R*: $\bigwedge a\ s1\ b\ s2. \llbracket \text{bad1 } s1 = \text{bad2 } s2; \neg \text{bad2 } s2 \implies a = b \wedge X\ s1\ s2 \rrbracket \implies R\ (a, s1)\ (b, s2)$
shows *rel-spmf* $R\ (\text{exec-gpv } \text{oracle1}\ \text{gpv } s1)\ (\text{exec-gpv } \text{oracle2}\ \text{gpv } s2)$
<proof>

lemma *max-enn2ereal*: $\text{max } (\text{enn2ereal } x)\ (\text{enn2ereal } y) = \text{enn2ereal } (\text{max } x\ y)$
including *ennreal.lifting* *<proof>*

lemma *identical-until-bad*:

assumes *bad-eq*: $\text{map-spmf } \text{bad } p = \text{map-spmf } \text{bad } q$
and *not-bad*: $\text{measure } (\text{measure-spmf } (\text{map-spmf } (\lambda x. (f\ x, \text{bad } x))\ p))\ (A \times \{\text{False}\}) = \text{measure } (\text{measure-spmf } (\text{map-spmf } (\lambda x. (f\ x, \text{bad } x))\ q))\ (A \times \{\text{False}\})$
shows $|\text{measure } (\text{measure-spmf } (\text{map-spmf } f\ p))\ A - \text{measure } (\text{measure-spmf } (\text{map-spmf } f\ q))\ A| \leq \text{spmfs } (\text{map-spmf } \text{bad } p)\ \text{True}$
<proof>

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

fixes $f :: 's \Rightarrow 'r\ \text{spmfs}$
and $g :: 'x \times 's \Rightarrow 'r \Rightarrow 'y\ \text{spmfs}$
and $s :: 's$
defines $\text{exec-gpv2} \equiv \text{exec-gpv}$
assumes *cond*: $\bigwedge s\ x\ y\ s'. \llbracket (y, s') \in \text{set-spmfs } (\text{callee } s\ x); I\ s \rrbracket \implies f\ s = f\ s'$
and $\mathcal{I} = \mathcal{I}\text{-full}$

shows $\text{bind-spmf} (\text{exec-gpv} \text{ callee } \text{gpv } s) (\lambda \text{as}. \text{bind-spmf} (f (\text{snd } \text{as})) (g \text{ as})) =$
 $\text{exec-gpv2} (\lambda (r, s) x. \text{bind-spmf} (\text{callee } s x) (\lambda (y, s'). \text{if } I \text{ s' } \wedge r = \text{None} \text{ then}$
 $\text{map-spmf} (\lambda r. (y, (\text{Some } r, s'))) (f s') \text{ else return-spmf } (y, (r, s')))) \text{gpv} (\text{None},$
 $s)$
 $\gg= (\lambda (a, r, s). \text{case } r \text{ of } \text{None} \Rightarrow \text{bind-spmf} (f s) (g (a, s)) \mid \text{Some } r' \Rightarrow g (a,$
 $s) r')$
(is ?lhs = ?rhs is - = bind-spmf (exec-gpv2 ?callee2 - -) -)
 $\langle \text{proof} \rangle$

primcorec $\text{gpv-stop} :: ('a, 'c, 'r) \text{gpv} \Rightarrow ('a \text{ option}, 'c, 'r \text{ option}) \text{gpv}$
where
 $\text{the-gpv} (\text{gpv-stop } \text{gpv}) =$
 $\text{map-spmf} (\text{map-generat } \text{Some } \text{id} (\lambda \text{rpv } \text{input}. \text{case } \text{input} \text{ of } \text{None} \Rightarrow \text{Done } \text{None}$
 $\mid \text{Some } \text{input}' \Rightarrow \text{gpv-stop } (\text{rpv } \text{input}'))$
 $(\text{the-gpv } \text{gpv})$

lemma $\text{gpv-stop-Done} [\text{simp}]: \text{gpv-stop} (\text{Done } x) = \text{Done} (\text{Some } x)$
 $\langle \text{proof} \rangle$

lemma $\text{gpv-stop-Fail} [\text{simp}]: \text{gpv-stop } \text{Fail} = \text{Fail}$
 $\langle \text{proof} \rangle$

lemma $\text{gpv-stop-Pause} [\text{simp}]: \text{gpv-stop} (\text{Pause } \text{out } \text{rpv}) = \text{Pause } \text{out} (\lambda \text{input}. \text{case}$
 $\text{input} \text{ of } \text{None} \Rightarrow \text{Done } \text{None} \mid \text{Some } \text{input}' \Rightarrow \text{gpv-stop } (\text{rpv } \text{input}'))$
 $\langle \text{proof} \rangle$

lemma $\text{gpv-stop-lift-spmf} [\text{simp}]: \text{gpv-stop} (\text{lift-spmf } p) = \text{lift-spmf} (\text{map-spmf}$
 $\text{Some } p)$
 $\langle \text{proof} \rangle$

lemma $\text{gpv-stop-bind} [\text{simp}]:$
 $\text{gpv-stop} (\text{bind-gpv } \text{gpv } f) = \text{bind-gpv} (\text{gpv-stop } \text{gpv}) (\lambda x. \text{case } x \text{ of } \text{None} \Rightarrow \text{Done}$
 $\text{None} \mid \text{Some } x' \Rightarrow \text{gpv-stop} (f x'))$
 $\langle \text{proof} \rangle$

context includes lifting-syntax **begin**

lemma $\text{gpv-stop-parametric}'$:
notes $[\text{transfer-rule}] = \text{the-gpv-parametric}' \text{ the-gpv-parametric}' \text{ Done-parametric}'$
 $\text{corec-gpv-parametric}'$
shows $(\text{rel-gpv}'' A C R \text{ ==>} \text{rel-gpv}'' (\text{rel-option } A) C (\text{rel-option } R)) \text{gpv-stop}$
 gpv-stop
 $\langle \text{proof} \rangle$

lemma $\text{gpv-stop-parametric} [\text{transfer-rule}]:$
shows $(\text{rel-gpv } A C \text{ ==>} \text{rel-gpv} (\text{rel-option } A) C) \text{gpv-stop } \text{gpv-stop}$
 $\langle \text{proof} \rangle$

lemma gpv-stop-transfer :

$(rel-gpv'' A B C ==> rel-gpv'' (pcr-Some A) B (pcr-Some C)) (\lambda x. x) gpv-stop$
 $\langle proof \rangle$

end

lemma *gpv-stop-map'* [*simp*]:

$gpv-stop (map-gpv' f g h gpv) = map-gpv' (map-option f) g (map-option h)$
 $(gpv-stop gpv)$
 $\langle proof \rangle$

lemma *interaction-bound-gpv-stop* [*simp*]:

$interaction-bound consider (gpv-stop gpv) = interaction-bound consider gpv$
 $\langle proof \rangle$

abbreviation *exec-gpv-stop* :: $('s \Rightarrow 'c \Rightarrow ('r option \times 's) spmf) \Rightarrow ('a, 'c, 'r)$
 $gpv \Rightarrow 's \Rightarrow ('a option \times 's) spmf$

where *exec-gpv-stop callee gpv* $\equiv exec-gpv callee (gpv-stop gpv)$

abbreviation *inline-stop* :: $('s \Rightarrow 'c \Rightarrow ('r option \times 's, 'c', 'r') gpv) \Rightarrow ('a, 'c, 'r)$
 $gpv \Rightarrow 's \Rightarrow ('a option \times 's, 'c', 'r') gpv$

where *inline-stop callee gpv* $\equiv inline callee (gpv-stop gpv)$

context

fixes *joint-oracle* :: $'s1 \Rightarrow 's2 \Rightarrow 'c \Rightarrow (('r option \times 's1) option \times ('r option \times 's2) option) pmf$

and *callee1* :: $'s1 \Rightarrow 'c \Rightarrow ('r option \times 's1) spmf$

notes $[[function-internals]]$

begin

partial-function (*spmf*) *exec-until-stop* :: $('a option, 'c, 'r) gpv \Rightarrow 's1 \Rightarrow 's2 \Rightarrow$
 $bool \Rightarrow ('a option \times 's1 \times 's2) spmf$

where

$exec-until-stop gpv s1 s2 b =$

$(if b then$

$bind-spmf (the-gpv gpv) (\lambda generat. case generat of$

$Pure x \Rightarrow return-spmf (x, s1, s2)$

$| IO out rpv \Rightarrow bind-pmf (joint-oracle s1 s2 out) (\lambda(a, b).$

$case a of None \Rightarrow return-pmf None$

$| Some (r1, s1') \Rightarrow (case b of None \Rightarrow undefined | Some (r2, s2') \Rightarrow$

$(case (r1, r2) of (None, None) \Rightarrow exec-until-stop (Done None) s1' s2'$

$True$

$| (Some r1', Some r2') \Rightarrow exec-until-stop (rpv r1') s1' s2' True$

$| (None, Some r2') \Rightarrow exec-until-stop (Done None) s1' s2' True$

$| (Some r1', None) \Rightarrow exec-until-stop (rpv r1') s1' s2' False))))$

$else$

$bind-spmf (the-gpv gpv) (\lambda generat. case generat of$

$Pure x \Rightarrow return-spmf (None, s1, s2)$

$| IO out rpv \Rightarrow bind-spmf (callee1 s1 out) (\lambda(r1, s1').$

$case r1 of None \Rightarrow exec-until-stop (Done None) s1' s2 False$

| *Some* $r1' \Rightarrow \text{exec-until-stop } (rpv \ r1') \ s1' \ s2 \ \text{False}))$

end

lemma *ord-spmf-exec-gpv-stop*:

fixes *callee1* :: ('c, 'r option, 's) callee

and *callee2* :: ('c, 'r option, 's) callee

and *S* :: 's \Rightarrow 's \Rightarrow bool

and *gpv* :: ('a, 'c, 'r) gpv

assumes *bisim*:

$\bigwedge s1 \ s2 \ x. \llbracket S \ s1 \ s2; \neg \text{stop } s2 \rrbracket \Longrightarrow$

$\text{ord-spmf } (\lambda(r1, s1') (r2, s2'). \text{le-option } r2 \ r1 \wedge S \ s1' \ s2' \wedge (r2 = \text{None} \wedge r1 \neq \text{None} \longleftrightarrow \text{stop } s2'))$

$(\text{callee1 } s1 \ x) (\text{callee2 } s2 \ x)$

and *init*: $S \ s1 \ s2$

and *go*: $\neg \text{stop } s2$

and *sticking*: $\bigwedge s1 \ s2 \ x \ y \ s1'. \llbracket (y, s1') \in \text{set-spmf } (\text{callee1 } s1 \ x); S \ s1 \ s2; \text{stop } s2 \rrbracket \Longrightarrow S \ s1' \ s2$

shows $\text{ord-spmf } (\text{rel-prod } (\text{ord-option } \top)^{-1-1} \ S) (\text{exec-gpv-stop } \text{callee1 } \text{gpv } s1) (\text{exec-gpv-stop } \text{callee2 } \text{gpv } s2)$

<proof>

end

theory *GPV-Applicative imports*

Generative-Probabilistic-Value

SPMF-Applicative

begin

6.7 Applicative instance for $(-, 'out, 'in) \text{ gpv}$

definition *ap-gpv* :: ('a \Rightarrow 'b, 'out, 'in) gpv \Rightarrow ('a, 'out, 'in) gpv \Rightarrow ('b, 'out, 'in) gpv

where $\text{ap-gpv } f \ x = \text{bind-gpv } f \ (\lambda f'. \text{bind-gpv } x \ (\lambda x'. \text{Done } (f' \ x)))$

adhoc-overloading *Applicative.ap* \equiv *ap-gpv*

abbreviation (*input*) *pure-gpv* :: 'a \Rightarrow ('a, 'out, 'in) gpv

where $\text{pure-gpv} \equiv \text{Done}$

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

lemma *ap-gpv-id*: $\text{pure-gpv } (\lambda x. \ x) \diamond x = x$

<proof>

lemma *ap-gpv-comp*: $\text{pure-gpv } (\circ) \diamond u \diamond v \diamond w = u \diamond (v \diamond w)$

<proof>

lemma *ap-gpv-homo*: $\text{pure-gpv } f \diamond \text{pure-gpv } x = \text{pure-gpv } (f \ x)$

<proof>

lemma *ap-gpv-interchange*: $u \diamond \text{pure-gpv } x = \text{pure-gpv } (\lambda f. f x) \diamond u$
 ⟨proof⟩

applicative *gpv*
for

pure: *pure-gpv*
ap: *ap-gpv*
 ⟨proof⟩

lemma *map-conv-ap-gpv*: $\text{map-gpv } f (\lambda x. x) \text{ gpv} = \text{pure-gpv } f \diamond \text{gpv}$
 ⟨proof⟩

lemma *exec-gpv-ap*:

exec-gpv callee $(f \diamond x) \sigma =$
exec-gpv callee $f \sigma \gg= (\lambda(f', \sigma'). \text{pure-spmf } (\lambda(x', \sigma''). (f' x', \sigma'')) \diamond \text{exec-gpv}$
callee $x \sigma')$
 ⟨proof⟩

lemma *exec-gpv-ap-pure* [*simp*]:

exec-gpv callee $(\text{pure-gpv } f \diamond x) \sigma = \text{pure-spmf } (\text{apfst } f) \diamond \text{exec-gpv callee } x \sigma$
 ⟨proof⟩

end

end

7 Cyclic groups

theory *Cyclic-Group* **imports**

HOL-Algebra.Coset

begin

record *'a cyclic-group* = *'a monoid* +
generator :: *'a* (*'g1*)

locale *cyclic-group = group* *G*

for *G* :: (*'a*, *'b*) *cyclic-group-scheme* (**structure**)

+

assumes *generator-closed* [*intro*, *simp*]: *generator* $G \in \text{carrier } G$

and *generator*: $\text{carrier } G \subseteq \text{range } (\lambda n :: \text{nat}. \text{generator } G [_]_G n)$

begin

lemma *generatorE* [*elim?*]:

assumes $x \in \text{carrier } G$

obtains $n :: \text{nat}$ **where** $x = \text{generator } G [_] n$

⟨proof⟩

lemma *inj-on-generator*: *inj-on* $(([_]) \mathbf{g}) \{..<\text{order } G\}$

<proof>

lemma *finite-carrier*: *finite (carrier G)*

<proof>

lemma *carrier-conv-generator*: *carrier G = (λn. g [↑] n) ‘ {..*order G*}*

<proof>

lemma *bij-betw-generator-carrier*:

*bij-betw (λn :: nat. g [↑] n) {..*order G*} (carrier G)*

<proof>

lemma *order-gt-0*: *order G > 0*

<proof>

end

lemma (**in monoid**) *order-in-range-Suc*: *order G ∈ range Suc ↔ finite (carrier*

G)

<proof>

end

theory *Cyclic-Group-SPMF imports*

HOL-Probability.SPMF Cyclic-Group

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*]: $0 < n \implies \text{set-spmf } (\text{sample-uniform } n)$
 $= \{..<n\}$
 <proof>

lemma (in *cyclic-group*) *sample-uniform-one-time-pad*:
assumes [*simp*]: $c \in \text{carrier } G$
shows
 $\text{map-spmf } (\lambda x. \mathbf{g} [\uparrow] x \otimes c) (\text{sample-uniform } (\text{order } G)) =$
 $\text{map-spmf } (\lambda x. \mathbf{g} [\uparrow] x) (\text{sample-uniform } (\text{order } G))$
 (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.