

CryptHOL

Andreas Lochbihler

December 20, 2018

Abstract

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

Contents

| | | |
|----------|--|----------|
| 1 | Miscellaneous library additions | 4 |
| 1.1 | HOL | 4 |
| 1.2 | Relations | 4 |
| 1.3 | Pairs | 7 |
| 1.4 | Sums | 7 |
| 1.5 | Option | 8 |
| 1.5.1 | Predicator and relator | 8 |
| 1.5.2 | Orders on option | 9 |
| 1.5.3 | Filter for option | 9 |
| 1.5.4 | Assert for option | 10 |
| 1.5.5 | Join on options | 10 |
| 1.5.6 | Zip on options | 11 |
| 1.5.7 | Binary supremum on <i>'a option</i> | 12 |
| 1.5.8 | Maps | 13 |

| | | |
|----------|---|-----------|
| 1.6 | Countable | 13 |
| 1.7 | Extended naturals | 14 |
| 1.8 | Extended non-negative reals | 14 |
| 1.9 | BNF material | 15 |
| 1.10 | Transfer and lifting material | 17 |
| 1.11 | Arithmetic | 18 |
| 1.12 | Chain-complete partial orders and <i>partial-function</i> | 19 |
| 1.13 | Folding over finite sets | 23 |
| 1.14 | Parametrisation of transfer rules | 23 |
| 1.15 | Lists | 23 |
| | 1.15.1 List of a given length | 24 |
| | 1.15.2 The type of lists of a given length | 25 |
| 1.16 | Streams and infinite lists | 25 |
| 1.17 | Monomorphic monads | 26 |
| 1.18 | Measures | 26 |
| 1.19 | Sequence space | 27 |
| 1.20 | Probability mass functions | 27 |
| 1.21 | Subprobability mass functions | 29 |
| | 1.21.1 Embedding of <i>'a option</i> into <i>'a spmf</i> | 35 |
| 1.22 | Applicative instance for <i>'a set</i> | 37 |
| 1.23 | Applicative instance for <i>'a spmf</i> | 38 |
| 1.24 | Exclusive or on lists | 39 |
| 1.25 | The environment functor | 41 |
| 1.26 | Setup for <i>partial-function</i> for sets | 43 |
| 2 | Negligibility | 46 |
| 3 | The resumption-error monad | 49 |
| | 3.1 Setup for <i>partial-function</i> | 54 |
| | 3.2 Setup for lifting and transfer | 57 |
| 4 | Generative probabilistic values | 58 |
| | 4.1 Single-step generative | 58 |
| | 4.2 Type definition | 63 |
| | 4.3 Generalised mapper and relator | 67 |
| | 4.4 Simple, derived operations | 70 |
| | 4.5 Monad structure | 74 |
| | 4.6 Embedding <i>'a spmf</i> as a monad | 78 |
| | 4.7 Embedding <i>'a option</i> as a monad | 81 |
| | 4.8 Embedding resumptions | 82 |
| | 4.9 Assertions | 84 |
| | 4.10 Order for (<i>'a, 'out, 'in</i>) <i>gpv</i> | 85 |
| | 4.11 Bounds on interaction | 86 |
| | 4.12 Typing | 93 |

| | | |
|----------|---|------------|
| 4.12.1 | Interface between gpvs and rpvs / callees | 93 |
| 4.12.2 | Type judgements | 96 |
| 4.13 | Sub-gpvs | 98 |
| 4.14 | Losslessness | 99 |
| 4.15 | Sequencing with failure handling included | 106 |
| 4.16 | Inlining | 109 |
| 4.17 | Running GPVs | 118 |
| 4.18 | Expectation transformer semantics | 131 |
| 4.19 | Probabilistic termination | 138 |
| 4.20 | Bisimulation for oracles | 140 |
| 4.21 | Applicative instance for $(-, 'out, 'in) gpv$ | 147 |
| 5 | Oracle combinators | 148 |
| 5.1 | Shared state | 148 |
| 5.2 | Shared state with aborts | 151 |
| 5.3 | Disjoint state | 151 |
| 5.4 | Indexed oracles | 153 |
| 5.5 | State extension | 153 |
| 6 | Combining GPVs | 154 |
| 6.1 | Shared state without interrupts | 154 |
| 6.2 | Shared state with interrupts | 155 |
| 7 | Cyclic groups | 155 |

1 Miscellaneous library additions

theory *Misc-CryptHOL* **imports**

Probabilistic-While.While-SPMF

HOL-Library.Rewrite

HOL-Library.Simps-Case-Conv

HOL-Library.Type-Length

HOL-Eisbach.Eisbach

Coinductive.TLList

Monad-Normalisation.Monad-Normalisation

Monomorphic-Monad.Monomorphic-Monad

begin

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

1.1 HOL

lemma *asm-rl-conv*: $(PROP P \implies PROP P) \equiv Trueprop True$

<proof>

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

lemma *if-mono-cong*: $\llbracket b \implies x \leq x'; \neg b \implies y \leq y' \rrbracket \implies If\ b\ x\ y \leq If\ b\ x'\ y'$

<proof>

lemma *if-cong-then*: $\llbracket b = b'; b' \implies t = t'; e = e' \rrbracket \implies If\ b\ t\ e = If\ b'\ t'\ e'$

<proof>

lemma *if-False-eq*: $\llbracket b \implies False; e = e' \rrbracket \implies If\ b\ t\ e = e'$

<proof>

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

<proof>

lemma *inj-on-fun-updD*: $\llbracket inj\ on\ (f(x := y))\ A; x \notin A \rrbracket \implies inj\ on\ f\ A$

<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>

1.2 Relations

inductive *Imagep* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b \Rightarrow bool$

for $R\ P$

where $\text{Image}I: \llbracket P \ x; R \ x \ y \rrbracket \Longrightarrow \text{Image}P \ R \ P \ y$

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

lemma transp-tranclp-id :
 assumes $\text{transp} \ R$
 shows $\text{tranclp} \ R = R$
 $\langle \text{proof} \rangle$

lemma transp-inv-image : $\text{transp} \ r \Longrightarrow \text{transp} \ (\lambda x \ y. r \ (f \ x) \ (f \ y))$
 $\langle \text{proof} \rangle$

lemma $\text{Domain}P\text{-converse}P$: $\text{Domain}P \ R^{-1-1} = \text{Range}P \ R$
 $\langle \text{proof} \rangle$

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

definition $\text{restrict-relp} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$
 $(- \upharpoonright (- \otimes -) [53, 54, 54] 53)$
where $\text{restrict-relp} \ R \ P \ Q = (\lambda x \ y. R \ x \ y \wedge P \ x \wedge Q \ y)$

lemma $\text{restrict-relp-apply}$ [simp]: $(R \upharpoonright P \otimes Q) \ x \ y \longleftrightarrow R \ x \ y \wedge P \ x \wedge Q \ y$
 $\langle \text{proof} \rangle$

lemma restrict-relpI [intro?]: $\llbracket R \ x \ y; P \ x; Q \ y \rrbracket \Longrightarrow (R \upharpoonright P \otimes Q) \ x \ y$
 $\langle \text{proof} \rangle$

lemma restrict-relpE [elim? , cases pred]:
 assumes $(R \upharpoonright P \otimes Q) \ x \ y$
 obtains $(\text{restrict-relp}) \ R \ x \ y \ P \ x \ Q \ y$
 $\langle \text{proof} \rangle$

lemma $\text{converse}P\text{-restrict-relp}$ [simp]: $(R \upharpoonright P \otimes Q)^{-1-1} = R^{-1-1} \upharpoonright Q \otimes P$
 $\langle \text{proof} \rangle$

lemma $\text{restrict-relp-restrict-relp}$ [simp]: $R \upharpoonright P \otimes Q \upharpoonright P' \otimes Q' = R \upharpoonright \text{inf} \ P \ P' \otimes \text{inf} \ Q \ Q'$
 $\langle \text{proof} \rangle$

lemma $\text{restrict-relp-cong}$:
 $\llbracket P = P'; Q = Q'; \bigwedge x \ y. \llbracket P \ x; Q \ y \rrbracket \Longrightarrow R \ x \ y = R' \ x \ y \rrbracket \Longrightarrow R \upharpoonright P \otimes Q = R' \upharpoonright P' \otimes Q'$
 $\langle \text{proof} \rangle$

lemma *restrict-relp-cong-simp*:

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

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

lemma *restrict-relp-mono*: $\llbracket R \leq R'; P \leq P'; Q \leq Q' \rrbracket \implies R \upharpoonright P \otimes Q \leq R' \upharpoonright P' \otimes Q'$
 $\langle \text{proof} \rangle$

lemma *restrict-relp-mono'*:

$\llbracket (R \upharpoonright P \otimes Q) x y; \llbracket R x y; P x; Q y \rrbracket \implies R' x y \ \&\&\& \ P' x \ \&\&\& \ Q' y \rrbracket$
 $\implies (R' \upharpoonright P' \otimes Q') x y$
 $\langle \text{proof} \rangle$

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

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

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

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

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*]: $\text{rel-prod2 } R a (c, b) \longleftrightarrow R a b$
 $\langle \text{proof} \rangle$

lemma *restrict-rel-prod*:

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

lemma *restrict-rel-prod1*:

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

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>

1.3 Pairs

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

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

shows *fst-corec-prod*: $fst\ (corec\text{-}prod\ f\ g\ s) = f\ s$

and *snd-corec-prod*: $snd\ (corec\text{-}prod\ f\ g\ s) = g\ s$

<proof>

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

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

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

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

1.4 Sums

lemma *islE*:

assumes *isl* x

obtains l **where** $x = Inl\ l$

<proof>

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

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

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

<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>

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*:
 $\llbracket \text{pred-option } P \ x; \bigwedge a. \llbracket a \in \text{set-option } x; P \ a \rrbracket \implies P' \ a \rrbracket \implies \text{pred-option } P' \ x$
<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?*]:
 $\llbracket \text{rel-option } R \ x \ y; \text{pred-option } P \ x; \text{pred-option } Q \ y \rrbracket \implies \text{rel-option } (R \upharpoonright P \otimes Q) \ x \ y$
<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*:

rel-option ($R \upharpoonright P \otimes Q$) $x y \longleftrightarrow \text{rel-option } R x y \wedge \text{pred-option } P x \wedge \text{pred-option } Q y$

<proof>

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

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

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

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

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

<proof>

1.5.2 Orders on option

abbreviation *le-option* :: $'a \text{ option} \Rightarrow 'a \text{ option} \Rightarrow \text{bool}$

where *le-option* $\equiv \text{ord-option } (=)$

lemma *le-option-bind-mono*:

$\llbracket \text{le-option } x y; \bigwedge a. a \in \text{set-option } x \implies \text{le-option } (f a) (g a) \rrbracket$

$\implies \text{le-option } (\text{Option.bind } x f) (\text{Option.bind } y g)$

<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* :: $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'b \text{ option} \Rightarrow \text{bool}$

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** $y = \text{Some } z \wedge R x z$

<proof>

1.5.3 Filter for option

fun *filter-option* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ option} \Rightarrow 'a \text{ option}$

where

filter-option $P \text{ None} = \text{None}$

$\mid \text{filter-option } P (\text{Some } x) = (\text{if } P x \text{ then } \text{Some } x \text{ else } \text{None})$

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*: $filter-option P x = Option.bind\ x\ (\lambda y. if\ P\ y\ then\ Some\ y\ else\ None)$
 ⟨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*: $\text{map-option } f \text{ (join-option } x) = \text{join-option (map-option (map-option } f) x)$
<proof>

lemma *bind-conv-join-option*: $\text{Option.bind } x f = \text{join-option (map-option } f x)$
<proof>

lemma *join-conv-bind-option*: $\text{join-option } x = \text{Option.bind } x \text{ id}$
<proof>

lemma *join-option-parametric* [*transfer-rule*]:
 includes *lifting-syntax* **shows**
 $(\text{rel-option (rel-option } R) \implies \text{rel-option } R) \text{ join-option join-option}$
<proof>

lemma *join-option-eq-Some* [*simp*]: $\text{join-option } x = \text{Some } y \iff x = \text{Some (Some } y)$
<proof>

lemma *Some-eq-join-option* [*simp*]: $\text{Some } y = \text{join-option } x \iff x = \text{Some (Some } y)$
<proof>

lemma *join-option-eq-None*: $\text{join-option } x = \text{None} \iff x = \text{None} \vee x = \text{Some None}$
<proof>

lemma *None-eq-join-option*: $\text{None} = \text{join-option } x \iff x = \text{None} \vee x = \text{Some None}$
<proof>

1.5.6 Zip on options

function *zip-option* :: 'a option \Rightarrow 'b option \Rightarrow ('a \times 'b) option
where

$\text{zip-option (Some } x) \text{ (Some } y) = \text{Some (} x, y)$
 | $\text{zip-option - None} = \text{None}$
 | $\text{zip-option None -} = \text{None}$
<proof>

termination <proof>

lemma *zip-option-eq-Some-iff* [*iff*]:
 $\text{zip-option } x y = \text{Some (} a, b) \iff x = \text{Some } a \wedge y = \text{Some } b$
<proof>

lemma *set-zip-option* [*simp*]:
 $\text{set-option (zip-option } x y) = \text{set-option } x \times \text{set-option } y$
<proof>

lemma *zip-map-option1*: $\text{zip-option } (\text{map-option } f \ x) \ y = \text{map-option } (\text{apfst } f) \ (\text{zip-option } x \ y)$
(*zip-option* $x \ y$)
(*proof*)

lemma *zip-map-option2*: $\text{zip-option } x \ (\text{map-option } g \ y) = \text{map-option } (\text{apsnd } g) \ (\text{zip-option } x \ y)$
(*zip-option* $x \ y$)
(*proof*)

lemma *map-zip-option*:
 $\text{map-option } (\text{map-prod } f \ g) \ (\text{zip-option } x \ y) = \text{zip-option } (\text{map-option } f \ x) \ (\text{map-option } g \ y)$
(*proof*)

lemma *zip-conv-bind-option*:
 $\text{zip-option } x \ y = \text{Option.bind } x \ (\lambda x. \text{Option.bind } y \ (\lambda y. \text{Some } (x, y)))$
(*proof*)

lemma *zip-option-parametric* [*transfer-rule*]:
includes *lifting-syntax* **shows**
 $(\text{rel-option } R \ ==\ ==\ ==\ ==\ > \text{rel-option } Q \ ==\ ==\ ==\ ==\ > \text{rel-option } (\text{rel-prod } R \ Q)) \ \text{zip-option}$
zip-option
(*proof*)

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

1.5.7 Binary supremum on 'a option

primrec *sup-option* :: 'a option \Rightarrow 'a option \Rightarrow 'a option
where

$\text{sup-option } x \ \text{None} = x$
| $\text{sup-option } x \ (\text{Some } y) = (\text{Some } y)$

lemma *sup-option-idem* [*simp*]: $\text{sup-option } x \ x = x$
(*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 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*: $f x = \text{None} \longleftrightarrow x \notin \text{dom } f$
<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 $\text{dom } m \subseteq \text{ran } m \longleftrightarrow \text{dom } m = \text{ran } m$
<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*]: $\text{Map.empty} \equiv \text{Map.empty}$

lemma *map-le-Some1D*: $\llbracket m \subseteq_m m'; m x = \text{Some } y \rrbracket \implies m' x = \text{Some } y$
<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* (*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* (*lfp F x*)
<proof>

1.7 Extended naturals

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

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

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

lemma *ennreal-real-conv-ennreal-of-enat*: $\text{ennreal } (\text{real } n) = \text{ennreal-of-enat } n$
 ⟨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]*: $(\text{SUP } x:A. f x) = (0 :: \text{enat}) \longleftrightarrow (\forall x \in A. f x = 0)$
 ⟨proof⟩

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

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

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

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

1.8 Extended non-negative reals

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

lemma (in *finite-measure*) *nn-integral-indicator-neq-top*:
 $f \text{ - ' } A \in \text{sets } M \implies (\int^+ x. \text{indicator } A (f x) \partial M) \neq \top$
 <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** $\llbracket \text{is-equality } A; \text{reflp } B \rrbracket \implies \text{reflp } (A \implies B)$
 <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 $\text{rel-fun } (\text{BNF-Def.Grp } A \text{ Abs}) (\text{BNF-Def.Grp } B g) = \text{BNF-Def.Grp } \{f. f \text{ ' } A \subseteq B\} (\text{Rep } \text{---} > g)$
 <proof>

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

lemma *rel-set-comp-Grp*:
 $\text{rel-set } R = (\text{BNF-Def.Grp } \{x. x \subseteq \{(x, y). R x y\}\} ((\text{' } \text{fst}))^{-1-1} \text{ OO } \text{BNF-Def.Grp } \{x. x \subseteq \{(x, y). R x y\}\} ((\text{' } \text{snd}))$
 <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**
 $(A \hat{=} \text{--} 1 \text{ ==>} B \hat{=} \text{--} 1) = (A \text{ ==>} B) \hat{=} \text{--} 1$
 ⟨*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*]:
left-total (*BNF-Def.Grp A f*) \longleftrightarrow $A = \text{UNIV}$
 ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

1.10 Transfer and lifting material

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

lemma *monotone-parametric* [transfer-rule]:
assumes [transfer-rule]: *bi-total* A
shows $((A \text{====>} A \text{====>} (=)) \text{====>} (B \text{====>} B \text{====>} (=)) \text{====>} (A \text{====>} B) \text{====>} (=))$ *monotone monotone*
 ⟨proof⟩

lemma *fun-ord-parametric* [transfer-rule]:
assumes [transfer-rule]: *bi-total* C
shows $((A \text{====>} B \text{====>} (=)) \text{====>} (C \text{====>} A) \text{====>} (C \text{====>} B))$

$====> (=)$ *fun-ord fun-ord*
 $\langle proof \rangle$

lemma *Plus-parametric* [*transfer-rule*]:
 $(rel\text{-set } A \text{ }====> \text{ } rel\text{-set } B \text{ }====> \text{ } rel\text{-set } (rel\text{-sum } A \text{ } B)) (<+>) (<+>)$
 $\langle proof \rangle$

lemma *pred-fun-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total* A
shows $((A \text{ }====> (=)) \text{ }====> (B \text{ }====> (=)) \text{ }====> (A \text{ }====> B) \text{ }====> (=))$ *pred-fun pred-fun*
 $\langle proof \rangle$

lemma *rel-fun-eq-OO*: $((=) \text{ }====> A) \text{ } OO ((=) \text{ }====> B) = ((=) \text{ }====> A \text{ } OO B)$
 $\langle proof \rangle$

end

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

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

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

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

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

1.11 Arithmetic

lemma *abs-diff-triangle-ineq2*: $|a - b| \text{ } :: \text{ } - \text{ } :: \text{ } ordered\text{-ab-group-add-abs} | \leq |a - c| + |c - b|$
 $\langle proof \rangle$

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 $n :: \text{nat}$ **shows** $\text{real } n \leq 1 \longleftrightarrow n \leq 1$
 ⟨*proof*⟩

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

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

lemma *fun-ordD*: $\text{fun-ord } ord \ f \ g \implies ord \ (f \ x) \ (g \ x)$
 ⟨*proof*⟩

lemma *parallel-fixp-induct-strong*:
assumes *ccpo1*: *class.ccpo luba orda* (*mk-less orda*)
and *ccpo2*: *class.ccpo lubb ordb* (*mk-less ordb*)
and *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) ($\lambda x. P \ (fst \ x)$)
 (*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 $P \ (ccpo.fixp \ luba \ orda \ f) \ (ccpo.fixp \ lubb \ ordb \ g)$
 ⟨*proof*⟩

lemma *parallel-fixp-induct-strong-uc*:
assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: $\bigwedge x. \text{monotone } (fun\text{-ord } orda) \ orda \ (\lambda f. U1 \ (F \ (C1 \ f)) \ x)$
and *G*: $\bigwedge y. \text{monotone } (fun\text{-ord } ordb) \ ordb \ (\lambda g. U2 \ (G \ (C2 \ g)) \ y)$
and *eq1*: $f \equiv C1 \ (ccpo.fixp \ (fun\text{-lub } luba) \ (fun\text{-ord } orda) \ (\lambda f. U1 \ (F \ (C1 \ f))))$
and *eq2*: $g \equiv C2 \ (ccpo.fixp \ (fun\text{-lub } lubb) \ (fun\text{-ord } ordb) \ (\lambda g. U2 \ (G \ (C2 \ g))))$
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*: $\bigwedge f' \ g'. \llbracket \bigwedge x. \text{orda } (U1 \ f' \ x) \ (U1 \ f \ x); \bigwedge y. \text{ordb } (U2 \ g' \ y) \ (U2 \ g \ y);$
 $P \ (U1 \ f') \ (U2 \ g') \rrbracket \implies P \ (U1 \ (F \ f')) \ (U2 \ (G \ g'))$
shows $P \ (U1 \ f) \ (U2 \ g)$
 ⟨*proof*⟩

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

lemmas *parallel-fixp-induct-strong-2-2* = *parallel-fixp-induct-strong-uc* [
of - - - - *case-prod - curry case-prod - curry,*
where $P = \lambda f \ g. P \ (curry \ f) \ (curry \ g),$

unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - refl refl,
split-format (complete), unfolded prod.case]
for *P*

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

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

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

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

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

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

lemma *mcont-finite-chains*:
assumes *finite*: $\text{finite-chains } \text{ord}$
and *mono*: $\text{monotone } \text{ord } \text{ord}' f$
and *ccpo*: $\text{class.ccpo } \text{lub } \text{ord } (\text{mk-less } \text{ord})$
and *ccpo'*: $\text{class.ccpo } \text{lub}' \text{ord}' (\text{mk-less } \text{ord}')$
shows $\text{mcont } \text{lub } \text{ord } \text{lub}' \text{ord}' f$
 $\langle \text{proof} \rangle$

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 $P (U f)$
 $\langle \text{proof} \rangle$

end

lemmas *parallel-fixp-induct-2-4* = *parallel-fixp-induct-uc*
of - - - case-prod - curry $\lambda f. \text{case-prod } (\text{case-prod } (\text{case-prod } f)) - \lambda f. \text{curry } (\text{curry } (\text{curry } f))$,
where $P = \lambda f g. P (\text{curry } f) (\text{curry } (\text{curry } (\text{curry } g)))$,
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*) option-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*]:
[[monotone (*fun-ord* (\geq)) (\geq) *f*; monotone (*fun-ord* (\geq)) (\geq) *g*]
 \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 $\text{Finite-Set.fold } f z (A \cup B) = \text{Finite-Set.fold } f (\text{Finite-Set.fold } f z A) B$

<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'*:

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

<proof>

definition *disjointp* :: $(a \Rightarrow \text{bool}) \text{ list} \Rightarrow \text{bool}$

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

lemma *disjointpD*:

$\llbracket \text{disjointp } xs; (xs ! n) x; (xs ! m) x; n < \text{length } xs; m < \text{length } xs \rrbracket \implies n = m$

<proof>

lemma *disjointpD'*:

$\llbracket \text{disjointp } xs; P x; Q x; xs ! n = P; xs ! m = Q; n < \text{length } xs; m < \text{length } xs \rrbracket \implies n = m$

<proof>

1.15.1 List of a given length

inductive-set $nlists :: 'a \text{ set} \Rightarrow \text{nat} \Rightarrow 'a \text{ list set for } A \ n$

where $nlists: \llbracket \text{set } xs \subseteq A; \text{length } xs = n \rrbracket \Longrightarrow xs \in nlists \ A \ n$

hide-fact (open) $nlists$

lemma $nlists\text{-alt-def}$: $nlists \ A \ n = \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\}$
 $\langle \text{proof} \rangle$

lemma $nlists\text{-empty}$: $nlists \ \{\} \ n = (\text{if } n = 0 \text{ then } \{\{\}\} \text{ else } \{\})$
 $\langle \text{proof} \rangle$

lemma $nlists\text{-empty-gt0}$ $[simp]$: $n > 0 \Longrightarrow nlists \ \{\} \ n = \{\}$
 $\langle \text{proof} \rangle$

lemma $nlists\text{-0}$ $[simp]$: $nlists \ A \ 0 = \{\{\}\}$
 $\langle \text{proof} \rangle$

lemma $Cons\text{-in-nlists-Suc}$ $[simp]$: $x \# xs \in nlists \ A \ (Suc \ n) \longleftrightarrow x \in A \wedge xs \in nlists \ A \ n$
 $\langle \text{proof} \rangle$

lemma $Nil\text{-in-nlists}$ $[simp]$: $\{\} \in nlists \ A \ n \longleftrightarrow n = 0$
 $\langle \text{proof} \rangle$

lemma $Cons\text{-in-nlists-iff}$: $x \# xs \in nlists \ A \ n \longleftrightarrow (\exists n'. n = Suc \ n' \wedge x \in A \wedge xs \in nlists \ A \ n')$
 $\langle \text{proof} \rangle$

lemma $in\text{-nlists-Suc-iff}$: $xs \in nlists \ A \ (Suc \ n) \longleftrightarrow (\exists x \ xs'. xs = x \# xs' \wedge x \in A \wedge xs' \in nlists \ A \ n)$
 $\langle \text{proof} \rangle$

lemma $nlists\text{-Suc}$: $nlists \ A \ (Suc \ n) = (\bigcup x \in A. (\#) \ x \ ' \ nlists \ A \ n)$
 $\langle \text{proof} \rangle$

lemma $replicate\text{-in-nlists}$ $[simp, \text{intro}]$: $x \in A \Longrightarrow replicate \ n \ x \in nlists \ A \ n$
 $\langle \text{proof} \rangle$

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

lemma $finite\text{-nlists}$ $[simp]$: $finite \ A \Longrightarrow finite \ (nlists \ A \ n)$
 $\langle \text{proof} \rangle$

lemma $finite\text{-nlistsD}$:
assumes $finite \ (nlists \ A \ n)$
shows $finite \ A \ \vee \ n = 0$
 $\langle \text{proof} \rangle$

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

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

lemma *in-nlists-UNIV*: $xs \in nlists\ UNIV\ n \longleftrightarrow \text{length } xs = n$
 ⟨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* $\longleftrightarrow x = \text{shd } ys \wedge \text{sprefix } xs\ (\text{stl } ys)$

lemma *sprefix-append*: $\text{sprefix } (xs\ @\ ys)\ zs \longleftrightarrow \text{sprefix } xs\ zs \wedge \text{sprefix } ys\ (\text{sdrop } (\text{length } xs)\ zs)$
 ⟨proof⟩

lemma *sprefix-stake-same* [*simp*]: $\text{sprefix } (\text{stake } n\ xs)\ xs$
 ⟨proof⟩

lemma *sprefix-same-imp-eq*:
assumes $\text{sprefix } xs\ ys\ \text{sprefix } xs'\ ys$
and $\text{length } xs = \text{length } xs'$
shows $xs = xs'$
 ⟨proof⟩

lemma *sprefix-shift-same* [*simp*]:
 $\text{sprefix } xs\ (xs\ @-\ ys)$
 ⟨proof⟩

lemma *sprefix-shift* [*simp*]:
 $\text{length } xs \leq \text{length } ys \implies \text{sprefix } xs\ (ys\ @-\ zs) \longleftrightarrow \text{prefix } xs\ ys$
 ⟨proof⟩

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

lemma *tlength-eq-infinity-iff*: $\text{tlength } xs = \infty \longleftrightarrow \neg \text{tfinite } xs$
including *tlist.lifting* ⟨proof⟩

1.17 Monomorphic monads

context includes *lifting-syntax* **begin**
<ML>

definition *bind-option* :: 'm fail \Rightarrow 'a option \Rightarrow ('a \Rightarrow 'm) \Rightarrow 'm
where *bind-option fail* x f = (case x of None \Rightarrow fail | Some x' \Rightarrow 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 monad. (x = None \implies m = fail) \implies *bind-option fail* x (λ -. m) = m
<proof>

end

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

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

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

end

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

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

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

1.18 Measures

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

lemma (in *sigma-algebra*) *sets-Collect-countable-Ex1*:

$(\bigwedge i :: 'i :: \text{countable}. \{x \in \Omega. P i x\} \in M) \implies \{x \in \Omega. \exists !i. P i x\} \in M$
 <proof>

lemma *pred-countable-Ex1* [*measurable*]:

$(\bigwedge i :: - :: \text{countable}. \text{Measurable.pred } M (\lambda x. P i x))$
 $\implies \text{Measurable.pred } M (\lambda x. \exists !i. P i x)$
 <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>

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*:

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

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

lemma *pmf-pred-mono-strong*:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *pred-pmf-cond* [*simp*]:

$\text{set-pmf } p \cap A \neq \{\} \implies \text{pred-pmf } P \text{ (cond-pmf } p \text{ } A) = \text{pred-pmf } (\lambda x. x \in A \longrightarrow P \text{ } x) \text{ } p$
 $\langle \text{proof} \rangle$

lemma *pred-pmf-pair* [*simp*]:

$\text{pred-pmf } P \text{ (pair-pmf } p \text{ } q) = \text{pred-pmf } (\lambda x. \text{pred-pmf } (P \circ \text{Pair } x) \text{ } q) \text{ } p$

<proof>

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

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

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

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

1.21 Subprobability mass functions

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

<proof>

lemma *ord-spmf-conv*:

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

<proof>

lemma *ord-spmf-expand*:

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

lemma *ord-spmf-eqD-measure*: $\text{ord-spmf } (=) p q \implies \text{measure } (\text{measure-spmf } p)$
 $A \leq \text{measure } (\text{measure-spmf } q) A$
<proof>

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)
<proof>

lemma *ord-spmf-bind-pmfI1*:

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

lemma *ord-spmf-bind-spmfI1*:

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

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

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 $\text{rel-spmf } R (\text{spmf-of-set } A) (\text{spmf-of-set } B)$
<proof>

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 $\text{nn-integral } (\text{measure-spmf } p) f = \text{nn-integral } (\text{restrict-space } (\text{measure-pmf } p) (\text{range } \text{Some})) (f \circ \text{the})$
<proof>

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

lemma *return-pmf-bind-option*:

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

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

lemma *rel-spmf-OO-trans* [*trans*]:
 $\llbracket rel\text{-}spmf\ R\ p\ q; rel\text{-}spmf\ S\ q\ r \rrbracket \implies rel\text{-}spmf\ (R\ OO\ S)\ p\ r$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

declare

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

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

lemma *return-pmf-map-option-conv-bind*:
 $return\text{-}pmf\ (map\text{-}option\ f\ x) = bind\text{-}spmf\ (return\text{-}pmf\ x)\ (return\text{-}spmf\ \circ\ f)$
 ⟨proof⟩

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

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

lemma *bind-pmf-spmf-assoc*:
 $g\ None = return\text{-}pmf\ None$
 $\implies bind\text{-}pmf\ (bind\text{-}spmf\ p\ f)\ g = bind\text{-}spmf\ p\ (\lambda x.\ bind\text{-}pmf\ (f\ x)\ g)$
 ⟨proof⟩

abbreviation *pred-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*: $pred\text{-}spmf\ P\ p \longleftrightarrow (\forall x \in set\text{-}spmf\ p.\ P\ x)$
 ⟨proof⟩

lemma *spmf-pred-mono-strong*:
 $\llbracket pred\text{-}spmf\ P\ p; \bigwedge a.\ \llbracket a \in set\text{-}spmf\ p; P\ a \rrbracket \implies P'\ a \rrbracket \implies pred\text{-}spmf\ P'\ p$
 ⟨proof⟩

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

lemma *rel-spmf-restrict-relI* [*intro?*]:
 $\llbracket rel\text{-}spmf\ R\ p\ q; pred\text{-}spmf\ P\ p; pred\text{-}spmf\ Q\ q \rrbracket \implies rel\text{-}spmf\ (R \upharpoonright P \otimes Q)\ p\ q$
 ⟨proof⟩

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

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

obtains $rel\text{-}spmf\ R\ x\ y\ pred\text{-}spmf\ P\ x\ pred\text{-}spmf\ Q\ y$

<proof>

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]:

$\text{pred-spmf } P (\text{try-spmf } p \ q) = (\text{pred-spmf } P \ p \wedge (\neg \text{lossless-spmf } p \longrightarrow \text{pred-spmf } P \ q))$
 $\langle \text{proof} \rangle$

lemma *pred-spmf-cond* [*simp*]:

$\text{pred-spmf } P (\text{cond-spmf } p \ A) = \text{pred-spmf } (\lambda x. x \in A \longrightarrow P \ x) \ p$
 $\langle \text{proof} \rangle$

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

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

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

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

includes *lifting-syntax* **shows**
 $(\text{rel-pmf } A \ ==\ ==> \text{rel-spmf } A) \ \text{spmf-of-pmf } \text{spmf-of-pmf}$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *pred-spmf-top*:

$\text{pred-spmf } (\lambda-. \ \text{True}) = (\lambda-. \ \text{True})$
 $\langle \text{proof} \rangle$

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

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

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

assumes *NO-MATCH* (*map-option* *g*) *f*
shows $\text{set-spmf } (\text{map-pmf } f \ p) = (\bigcup x \in \text{set-pmf } p. \ \text{set-option } (f \ x))$
 $\langle \text{proof} \rangle$

lemma *rel-spmf-bindI'*:

```

[[ rel-spmf A p q;  $\wedge x y. [ A x y; x \in \text{set-spmf } p; y \in \text{set-spmf } q ] \implies \text{rel-spmf } B$ 
(f x) (g y) ]
 $\implies \text{rel-spmf } B (p \ggg f) (q \ggg g)$ 
<proof>

```

1.21.1 Embedding of 'a option into 'a spmf

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

```

lemma bind-option-spmf-monad [simp]: monad.bind-option (return-pmf None) x
= bind-spmf (return-pmf x)
<proof>

```

locale option-to-spmf begin

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

```

definition the-pmf :: 'a pmf  $\Rightarrow$  '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*]:

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

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

$((=(=) \implies (=)) \implies \text{cr-spmf-option} \implies \text{cr-spmf-option}) \text{ map-spmf map-option}$
<proof>

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

$\text{cr-spmf-option (return-pmf None) None}$
<proof>

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

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

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

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

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

$((=(=) \implies (=) \implies (=)) \implies \text{cr-spmf-option} \implies \text{cr-spmf-option} \implies (=)) \text{ rel-spmf rel-option}$
<proof>

end

end

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

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

definition *cr-option-le-spmf* :: $'a \text{ option} \Rightarrow 'a \text{ spmf} \Rightarrow \text{bool}$

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*]:

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

lemma *map-option-le-spmf-transfer* [*transfer-rule*]:
 $((=) \implies (=)) \implies \text{cr-option-le-spmf} \implies \text{cr-option-le-spmf}$ *map-option*
map-spmf
 ⟨*proof*⟩

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

end

end

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

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

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

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

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

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

end

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

1.22 Applicative instance for 'a set

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

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

lemma *in-ap-setI*: $\llbracket f' \in f; x' \in x \rrbracket \Longrightarrow f' x' \in f \diamond x$
<proof>

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

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

end

end

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

1.23 Applicative instance for *'a* *spmf*

abbreviation (*input*) *pure-spmf* :: *'a* \Rightarrow *'a* *spmf*
where *pure-spmf* \equiv *return-spmf*

definition *ap-spmf* :: (*'a* \Rightarrow *'b*) *spmf* \Rightarrow *'a* *spmf* \Rightarrow *'b* *spmf*
where *ap-spmf* $f x = \text{map-spmf } (\lambda(f, x). f x)$ (*pair-spmf* $f x$)

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

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

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

lemma *ap-spmf-id*: $\text{pure-spmf } (\lambda x. x) \diamond x = x$
<proof>

lemma *ap-spmf-comp*: $\text{pure-spmf } (\circ) \diamond u \diamond v \diamond w = u \diamond (v \diamond w)$
<proof>

lemma *ap-spmf-homo*: $\text{pure-spmf } f \diamond \text{pure-spmf } x = \text{pure-spmf } (f x)$
<proof>

lemma *ap-spmf-interchange*: $u \diamond \text{pure-spmf } x = \text{pure-spmf } (\lambda f. f x) \diamond u$
<proof>

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

applicative *spmof* (*C*)
for
pure: *pure-spmf*
ap: *ap-spmf*
 <proof>

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

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

lemma *bind-pmf-ap-return-spmf* [*simp*]: *bind-pmf* (*ap-spmf* (*return-spmf* *f*) *p*) *g*
 = *bind-pmf* *p* ($g \circ \text{map-option } f$)
 <proof>

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

end

end

1.24 Exclusive or on lists

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

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

lemma *xor-bool-def* [*iff*]: **fixes** $x y :: \text{bool}$ **shows** $x \oplus y \longleftrightarrow x \neq y$
 <proof>

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

lemma *xor-assoc*:
fixes $x y :: 'a :: \text{boolean-algebra}$
shows $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
 <proof>

lemma *xor-left-commute*:

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

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

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

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

lemmas $\text{xor-ac} = \text{xor-assoc}\ \text{xor-commute}\ \text{xor-left-commute}$

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

lemma xor-list-unfold :
 $xs\ [\oplus]\ ys = (\text{case}\ xs\ \text{of}\ [] \Rightarrow []\ |\ x\ \# xs' \Rightarrow (\text{case}\ ys\ \text{of}\ [] \Rightarrow []\ |\ y\ \# ys' \Rightarrow x \oplus y\ \# xs' [\oplus]\ ys'))$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *xor-list-inverse [simp]:*

fixes *xs :: 'a :: boolean-algebra list*

shows $xs [\oplus] xs = \text{replicate } (\text{length } xs) \text{ bot}$

<proof>

lemma *xor-replicate-bot-right [simp]:*

fixes *xs :: 'a :: boolean-algebra list*

shows $\llbracket \text{length } xs \leq n; x = \text{bot} \rrbracket \implies xs [\oplus] \text{replicate } n x = xs$

<proof>

lemma *xor-replicate-bot-left [simp]:*

fixes *xs :: 'a :: boolean-algebra list*

shows $\llbracket \text{length } xs \leq n; x = \text{bot} \rrbracket \implies \text{replicate } n x [\oplus] xs = xs$

<proof>

lemma *xor-list-left-inverse [simp]:*

fixes *xs :: 'a :: boolean-algebra list*

shows $\text{length } ys \leq \text{length } xs \implies xs [\oplus] (xs [\oplus] ys) = ys$

<proof>

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

<proof>

lemma *inj-on-xor-list-nlists [simp]:*

fixes *xs :: 'a :: boolean-algebra list*

shows $n \leq \text{length } xs \implies \text{inj-on } (xor\text{-list } xs) (nlists \text{ UNIV } n)$

<proof>

lemma *one-time-pad:*

fixes *xs :: - :: boolean-algebra list*

shows $\text{length } xs \geq n \implies \text{map-spmf } (xor\text{-list } xs) (\text{spmof-of-set } (nlists \text{ UNIV } n))$
 $= \text{spmof-of-set } (nlists \text{ UNIV } n)$

<proof>

end

theory *Environment-Function imports*

Applicative-Lifting.Applicative-Environment

begin

1.25 The environment functor

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

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

<proof>

context includes *applicative-syntax begin*

lemma *ap-envir-apply* [*simp*]: $(f \diamond x) i = f i (x i)$
<proof>

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

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

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

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

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

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

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

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

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

lemma *pred-envir-mono*: $\llbracket \text{pred-envir } p f; \bigwedge x. p (f x) \Longrightarrow q (g x) \rrbracket \Longrightarrow \text{pred-envir } q g$
<proof>

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

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

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

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

lemma *rel-envirD*: $rel\text{-}envir\ p\ f\ g \implies p\ (f\ x)\ (g\ x)$
(proof)

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

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

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

lemma *pred-envir-mono2*: $\llbracket rel\text{-}envir\ p\ f\ g; \bigwedge x. p\ (f\ x)\ (g\ x) \implies q\ (f'\ x) \rrbracket \implies$
 $pred\text{-}envir\ q\ f'$
(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*: $fun\text{-}lub\ Sup = (Sup :: - \Rightarrow - :: complete\text{-}lattice)$
(proof)

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

abbreviation *mono-set* $\equiv monotone\ (fun\text{-}ord\ (\subseteq))\ (\subseteq)$

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*: $f \equiv C (\text{ccpo.fixp } (\text{fun-lub } \text{Sup}) (\text{fun-ord } (\leq))) (\lambda f. U (F (C f)))$
and *inverse2*: $\bigwedge f. U (C f) = f$
and *step*: $\bigwedge f x y. \llbracket \bigwedge x y. y \in U f x \implies P x y; y \in U (F f) x \rrbracket \implies P x y$
and *enforce-variable-ordering*: $x = x$
and *elem*: $y \in U f x$
shows $P x y$
 $\langle \text{proof} \rangle$

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

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

lemma *monotone-le-le*: $\text{monotone } (\leq) (\leq) = \text{mono}$
 $\langle \text{proof} \rangle$

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

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

$\langle \text{ML} \rangle$

lemma [*partial-function-mono*]:
shows *insert-mono*: $\text{mono-set } A \implies \text{mono-set } (\lambda f. \text{insert } x (A f))$
and *UNION-mono*: $\llbracket \text{mono-set } B; \bigwedge y. \text{mono-set } (\lambda f. C y f) \rrbracket \implies \text{mono-set } (\lambda f. \bigcup_{y \in B} f. C y f)$
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*: $\llbracket \text{mono-set } A; \text{mono-set } B \rrbracket \implies \text{mono-set } (\lambda f. A f \cap B f)$
and *Diff-mono1*: $\text{mono-set } A \implies \text{mono-set } (\lambda f. A f - X)$

and *image-mono*: $\text{mono-set } A \implies \text{mono-set } (\lambda f. g \text{ ' } A f)$
and *vimage-mono*: $\text{mono-set } A \implies \text{mono-set } (\lambda f. g \text{ -' } A f)$
 <proof>

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

test *xs i j* = insert 4 (test [] 0 j \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* (λf :: '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*: $\bigwedge x. \text{mono-coset } (\lambda f. U (F (C f)) x)$
and *eq*: $f \equiv C (\text{ccpo.fixp } (\text{fun-lub } \text{Inter}) (\text{fun-ord } (\geq))) (\lambda f. U (F (C f)))$
and *inverse2*: $\bigwedge f. U (C f) = f$

and *step*: $\bigwedge f' x y. [\bigwedge x. U f' x = U f' x; \neg P x y] \implies y \in U (F (C (\text{sup}$
 ($\lambda x. \{y. \neg P x y\} (U f)))) x$

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

lemma *[partial-function-mono]*:

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

context begin

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

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

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 proof \rangle

end

end

2 Negligibility

theory *Negligible imports*

Complex-Main

Landau-Symbols.Landau-More

begin

named-theorems *negligible-intros*

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

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

lemma negligibleI *[intro?]*:

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

lemma negligibleD :

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

lemma negligibleD-real :

assumes $\text{negligible } f$

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

<proof>

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

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*: *negligible f*
and *g*: *negligible g*
shows *negligible* ($\lambda\eta. f \eta * g \eta :: \text{real}$)
 $\langle \text{proof} \rangle$

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

lemma *negligible-powr* [*negligible-intros*]:
assumes *f*: *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 *negligible f*
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *negligible-times-poly*:
assumes *f*: *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*:

$\llbracket f \in O(\lambda x. x \text{ powr } n); \text{negligible } g \rrbracket \implies \text{negligible } (\lambda x. f x * g x)$
<proof>

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*) = ($\lambda \text{inp. Done None}$)

code-datatype *Done Pause*

primcorec *bind-resumption* ::

(*'a, 'out, 'in*) *resumption*

\Rightarrow (*'a* \Rightarrow (*'b, 'out, 'in*) *resumption*) \Rightarrow (*'b, 'out, 'in*) *resumption*

where

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

| *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*) = ($\lambda \text{inp. if is-Done } x \text{ then resume } (f \text{ (the (result$

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

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

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

lemma *is-Done-bind-resumption* [*simp*]:

$is-Done (x \ggg f) \longleftrightarrow is-Done x \wedge (result\ x \neq None \longrightarrow is-Done (f (the (result\ x))))$

$\langle proof \rangle$

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

$is-Done (x \ggg f) \implies result (x \ggg f) = result\ x \ggg result \circ f$

$\langle proof \rangle$

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

$\neg is-Done (x \ggg f) \implies output (x \ggg f) = (if\ is-Done\ x\ then\ output (f (the (result\ x)))\ else\ output\ x)$

$\langle proof \rangle$

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

$\neg is-Done (x \ggg f) \implies$

$resume (x \ggg f) =$

$(if\ is-Done\ x\ then\ resume (f (the (result\ x)))$

$else (\lambda inp. resume\ x\ inp \ggg f))$

$\langle proof \rangle$

definition *DONE* :: $'a \Rightarrow ('a, 'out, 'in)$ *resumption*

where *DONE* = *Done* \circ *Some*

definition *ABORT* :: $('a, 'out, 'in)$ *resumption*

where *ABORT* = *Done* *None*

lemma [*simp*]:

shows *is-Done-DONE*: $is-Done (DONE\ a)$

and *is-Done-ABORT*: $is-Done\ ABORT$

and *result-DONE*: $result (DONE\ a) = Some\ a$

and *result-ABORT*: $result\ ABORT = None$

and *DONE-inject*: $DONE\ a = DONE\ b \longleftrightarrow a = b$

and *DONE-neq-ABORT*: $DONE\ a \neq ABORT$

and *ABORT-neq-DONE*: $ABORT \neq DONE\ a$

and *ABORT-eq-Done*: $\bigwedge a. ABORT = Done\ a \longleftrightarrow a = None$

and *Done-eq-ABORT*: $\bigwedge a. Done\ a = ABORT \longleftrightarrow a = None$

and *DONE-eq-Done*: $\bigwedge b. DONE\ a = Done\ b \longleftrightarrow b = Some\ a$

and *Done-eq-DONE*: $\bigwedge b. Done\ b = DONE\ a \longleftrightarrow b = Some\ a$

and *DONE-neq-Pause*: $DONE\ a \neq Pause\ out\ c$

and *Pause-neq-DONE*: $Pause\ out\ c \neq DONE\ a$

and *ABORT-neq-Pause*: $ABORT \neq Pause\ out\ c$

and *Pause-neq-ABORT*: $Pause\ out\ c \neq ABORT$

<proof>

lemma *resume-ABORT* [*simp*]:

resume (Done r) = (λinp. ABORT)

<proof>

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

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

<proof>

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

<proof>

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

<proof>

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

<proof>

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

assumes *is-Done r*

obtains *(DONE) x where r = DONE x | (ABORT) r = ABORT*

<proof>

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

<proof>

lemma *Done-bind* [*code*]:

Done a \ggg f = (case a of None \Rightarrow Done None | Some a \Rightarrow f a)

<proof>

lemma *DONE-bind* [*simp*]:

DONE a \ggg f = f a

<proof>

lemma *bind-resumption-Pause* [*simp, code*]: **fixes cont shows**

Pause out cont \ggg f

= Pause out (λinp. cont inp \ggg f)

<proof>

lemma *bind-DONE* [*simp*]:

x \ggg DONE = x

<proof>

lemma *bind-bind-resumption*:

fixes *r :: ('a, 'in, 'out) resumption*

shows *(r \ggg f) \ggg g = do { x \leftarrow r; f x \ggg g }*

<proof>

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

lemma *ABORT-bind [simp]:* $ABORT \ggg f = ABORT$
(proof)

lemma *bind-resumption-is-Done:* $is-Done\ f \implies f \ggg g = (if\ result\ f = None\ then\ ABORT\ else\ g\ (the\ (result\ f)))$
(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:*
 $results\ (bind-resumption\ x\ f) = (\bigcup a \in results\ x. results\ (f\ a))$
(is ?lhs = ?rhs)
(proof)

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

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

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

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

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

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

$\neg \text{is-Done } r$
 $\implies \text{resume } (\text{map-resumption } f1 \ f2 \ r) = \text{map-resumption } f1 \ f2 \circ \text{resume } r$
 <proof>

lemma *rel-resumption-is-DoneD*: $\text{rel-resumption } A \ B \ r1 \ r2 \implies \text{is-Done } r1 \longleftrightarrow \text{is-Done } r2$
 <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: flat-ord None a a' \Longrightarrow *resumption-ord* (Done a) (Done a')

| *Done-Pause*: *resumption-ord* ABORT (Pause out c)

| *Pause-Pause*: ((=) \Longrightarrow *resumption-ord*) c c' \Longrightarrow *resumption-ord* (Pause out c) (Pause out c')

inductive-simps *resumption-ord-simps* [simp]:

resumption-ord (Pause out c) r

resumption-ord r (Done a)

lemma *resumption-ord-is-DoneD*:

\llbracket *resumption-ord* r r'; *is-Done* r' $\rrbracket \Longrightarrow$ *is-Done* r

<proof>

lemma *resumption-ord-resultD*:

\llbracket *resumption-ord* r r'; *is-Done* r' $\rrbracket \Longrightarrow$ flat-ord None (result r) (result r')

<proof>

lemma *resumption-ord-outputD*:

\llbracket *resumption-ord* r r'; \neg *is-Done* r $\rrbracket \Longrightarrow$ output r = output r'

<proof>

lemma *resumption-ord-resumeD*:

\llbracket *resumption-ord* r r'; \neg *is-Done* r $\rrbracket \Longrightarrow$ ((=) \Longrightarrow *resumption-ord*) (resume r) (resume r')

<proof>

lemma *resumption-ord-abort*:

\llbracket *resumption-ord* r r'; *is-Done* r; \neg *is-Done* r' $\rrbracket \Longrightarrow$ result r = None

<proof>

lemma *resumption-ord-coinduct* [consumes 1, case-names Done Abort Pause, case-conclusion Pause output resume, coinduct pred: *resumption-ord*]:

assumes X r r'

and *Done*: \bigwedge r r'. \llbracket X r r'; *is-Done* r' $\rrbracket \Longrightarrow$ *is-Done* r \wedge flat-ord None (result r) (result r')

and *Abort*: \bigwedge r r'. \llbracket X r r'; \neg *is-Done* r'; *is-Done* r $\rrbracket \Longrightarrow$ result r = None

and *Pause*: \bigwedge r r'. \llbracket X r r'; \neg *is-Done* r; \neg *is-Done* r' \rrbracket

\Longrightarrow output r = output r' \wedge ((=) \Longrightarrow (λ r r'. X r r' \vee *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 = 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) *resumption*
where

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

lemma *is-Done-resumption-lub* [*simp*]:
 $\text{is-Done } (\text{resumption-lub } R) \longleftrightarrow (\forall r \in R. \text{is-Done } r)$
 ⟨*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*]:
 $\exists r \in R. \neg \text{is-Done } r \implies \text{output } (\text{resumption-lub } R) = (\text{THE out. out} \in \text{output } ' (R \cap \{r. \neg \text{is-Done } r\}))$
 ⟨*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) resumption-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 $F \equiv \lambda \text{results } r. \text{case } r \text{ of } \text{resumption.Done } x \Rightarrow \text{set-option } x \mid \text{resumption.Pause out } c \Rightarrow \bigcup \text{input. results } (c \text{ input})$

shows *results-conv-fixp*: *results* \equiv *ccpo.fixp (fun-lub Union) (fun-ord (\subseteq)) F* (**is** - \equiv ?*fixp*)

and *results-mono*: $\bigwedge x. \text{monotone } (fun-ord (\subseteq)) (\subseteq) (\lambda f. F f x)$ (**is** *PROP ?mono*)

<proof>

lemma *mcont-case-resumption*:

fixes $f\ g$
defines $h \equiv \lambda r. \text{if is-Done } r \text{ then } f \text{ (result } r) \text{ else } g \text{ (output } r) \text{ (resume } r) \text{ } r$
assumes $mcont1: mcont \text{ (flat-lub } None) \text{ option-ord lub ord } f$
and $mcont2: \bigwedge out. mcont \text{ (fun-lub resumption-lub) (fun-ord resumption-ord) lub ord } (\lambda c. g \text{ out } c \text{ (Pause out } c))$
and $ccpo: class.ccpo \text{ lub ord (mk-less ord)}$
and $bot: \bigwedge x. ord \text{ (f } None) \text{ } x$
shows $mcont \text{ resumption-lub resumption-ord lub ord } (\lambda r. \text{case } r \text{ of Done } x \Rightarrow f \text{ } x \mid \text{Pause out } c \Rightarrow g \text{ out } c \text{ } r)$
(is mcont ?lub ?ord - - ?f)
 $\langle proof \rangle$

lemma *mcont2mcont-results*[*THEN mcont2mcont, cont-intro, simp*]:

shows *mcont-results*: $mcont \text{ resumption-lub resumption-ord Union } (\subseteq) \text{ results}$
 $\langle proof \rangle$

lemma *mono2mono-results*[*THEN lfp.mono2mono, cont-intro, simp*]:

shows *monotone-results*: $monotone \text{ resumption-ord } (\subseteq) \text{ results}$
 $\langle proof \rangle$

lemma *fixes f F*

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

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

shows *mcont-outputs*: $mcont \text{ resumption-lub resumption-ord Union } (\subseteq) \text{ outputs}$
 $\langle proof \rangle$

lemma *mono2mono-outputs*[*THEN lfp.mono2mono, cont-intro, simp*]:

shows *monotone-outputs*: $monotone \text{ resumption-ord } (\subseteq) \text{ outputs}$
 $\langle proof \rangle$

lemma *pred-resumption-antimono*:

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

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*]:
 $rel-resumption\ A\ B\ OO\ rel-resumption\ C\ D = rel-resumption\ (A\ OO\ C)\ (B\ OO\ D)$
 ⟨*proof*⟩

lemma *left-total-rel-resumption* [*transfer-rule*]:
 $\llbracket left-total\ R1;\ left-total\ R2 \rrbracket \implies left-total\ (rel-resumption\ R1\ R2)$
 ⟨*proof*⟩

lemma *left-unique-rel-resumption* [*transfer-rule*]:
 $\llbracket left-unique\ R1;\ left-unique\ R2 \rrbracket \implies left-unique\ (rel-resumption\ R1\ R2)$
 ⟨*proof*⟩

lemma *right-total-rel-resumption* [*transfer-rule*]:
 $\llbracket right-total\ R1;\ right-total\ R2 \rrbracket \implies right-total\ (rel-resumption\ R1\ R2)$
 ⟨*proof*⟩

lemma *right-unique-rel-resumption* [*transfer-rule*]:
 $\llbracket right-unique\ R1;\ right-unique\ R2 \rrbracket \implies right-unique\ (rel-resumption\ R1\ R2)$
 ⟨*proof*⟩

lemma *bi-total-rel-resumption* [*transfer-rule*]:
 $\llbracket bi-total\ A;\ bi-total\ B \rrbracket \implies bi-total\ (rel-resumption\ A\ B)$
 ⟨*proof*⟩

lemma *bi-unique-rel-resumption* [*transfer-rule*]:
 $\llbracket bi-unique\ A;\ bi-unique\ B \rrbracket \implies bi-unique\ (rel-resumption\ A\ B)$
 ⟨*proof*⟩

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

end

4 Generative probabilistic values

theory *Generat* imports
Misc-CryptHOL
 begin

4.1 Single-step generative

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

datatype-compat *generat*

lemma *IO-code-cong*: $out = out' \implies IO\ out\ c = IO\ out'\ c$ *<proof>*
<ML>

lemma *is-Pure-map-generat* [*simp*]: $is-Pure\ (map-generat\ f\ g\ h\ x) = is-Pure\ x$
<proof>

lemma *result-map-generat* [*simp*]: $is-Pure\ x \implies result\ (map-generat\ f\ g\ h\ x) = f$
 $(result\ x)$
<proof>

lemma *output-map-generat* [*simp*]: $\neg is-Pure\ x \implies output\ (map-generat\ f\ g\ h\ x)$
 $= g\ (output\ x)$
<proof>

lemma *continuation-map-generat* [*simp*]: $\neg is-Pure\ x \implies continuation\ (map-generat$
 $f\ g\ h\ x) = h\ (continuation\ x)$
<proof>

lemma [*simp*]:

shows *map-generat-eq-Pure*:

$map-generat\ f\ g\ h\ generat = Pure\ x \longleftrightarrow (\exists x'. generat = Pure\ x' \wedge x = f\ x')$

and *Pure-eq-map-generat*:

$Pure\ x = map-generat\ f\ g\ h\ generat \longleftrightarrow (\exists x'. generat = Pure\ x' \wedge x = f\ x')$

<proof>

lemma [*simp*]:

shows *map-generat-eq-IO*:

$map-generat\ f\ g\ h\ generat = IO\ out\ c \longleftrightarrow (\exists out'\ c'. generat = IO\ out'\ c' \wedge out$
 $= g\ out' \wedge c = h\ c')$

and *IO-eq-map-generat*:

$IO\ out\ c = map-generat\ f\ g\ h\ generat \longleftrightarrow (\exists out'\ c'. generat = IO\ out'\ c' \wedge out$
 $= g\ out' \wedge c = h\ c')$

<proof>

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

assumes *is-Pure generat*

obtains $(Pure)\ x$ **where** $generat = Pure\ x$

<proof>

lemma *not-is-PureE*:

assumes $\neg is-Pure\ generat$

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

<proof>

lemma *rel-generatI*:

$\llbracket is-Pure\ x \longleftrightarrow is-Pure\ y;$

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

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

$$\Longrightarrow \text{rel-generat } A \text{ Out } R \ x \ y$$
 <proof>

lemma *rel-generatD'*:

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

$$\Longrightarrow (\text{is-Pure } x \longleftrightarrow \text{is-Pure } y) \wedge$$

$$(\text{is-Pure } x \longrightarrow \text{is-Pure } y \longrightarrow A (\text{result } x) (\text{result } y)) \wedge$$

$$(\neg \text{is-Pure } x \longrightarrow \neg \text{is-Pure } y \longrightarrow \text{Out } (\text{output } x) (\text{output } y) \wedge R (\text{continuation } x) (\text{continuation } y))$$
 <proof>

lemma *rel-generatD*:

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

lemma *rel-generat-mono*:

$$\llbracket \text{rel-generat } A \ B \ C \ x \ y; \bigwedge x \ y. A \ x \ y \Longrightarrow A' \ x \ y; \bigwedge x \ y. B \ x \ y \Longrightarrow B' \ x \ y; \bigwedge x \ y. C \ x \ y \Longrightarrow C' \ x \ y \rrbracket$$

$$\Longrightarrow \text{rel-generat } A' \ B' \ C' \ x \ y$$
 <proof>

lemma *rel-generat-mono'* [*mono*]:

$$\llbracket \bigwedge x \ y. A \ x \ y \longrightarrow A' \ x \ y; \bigwedge x \ y. B \ x \ y \longrightarrow B' \ x \ y; \bigwedge x \ y. C \ x \ y \longrightarrow C' \ x \ y \rrbracket$$

$$\Longrightarrow \text{rel-generat } A \ B \ C \ x \ y \longrightarrow \text{rel-generat } A' \ B' \ C' \ x \ y$$
 <proof>

lemma *rel-generat-same*:

$$\text{rel-generat } A \ B \ C \ r \ r \longleftrightarrow$$

$$(\forall x \in \text{generat-pures } r. A \ x \ x) \wedge$$

$$(\forall \text{out} \in \text{generat-outs } r. B \ \text{out} \ \text{out}) \wedge$$

$$(\forall c \in \text{generat-contrs } r. C \ c \ c)$$
 <proof>

lemma *rel-generat-reflI*:

$$\llbracket \bigwedge y. y \in \text{generat-pures } x \Longrightarrow A \ y \ y;$$

$$\bigwedge \text{out}. \text{out} \in \text{generat-outs } x \Longrightarrow B \ \text{out} \ \text{out};$$

$$\bigwedge \text{cont}. \text{cont} \in \text{generat-contrs } x \Longrightarrow C \ \text{cont} \ \text{cont} \rrbracket$$

$$\Longrightarrow \text{rel-generat } A \ B \ C \ x \ x$$
 <proof>

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

$\wedge \text{reflp } C$
 $\langle \text{proof} \rangle$

lemma *transp-rel-generatI*:
 assumes *transp A transp B transp C*
 shows *transp (rel-generat A B C)*
 $\langle \text{proof} \rangle$

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

lemma *rel-generat-Pure1*: $\text{rel-generat } A \ B \ C \ (\text{Pure } x) = (\lambda r. \exists y. r = \text{Pure } y \wedge A \ x \ y)$
 $\langle \text{proof} \rangle$

lemma *rel-generat-IO1*: $\text{rel-generat } A \ B \ C \ (\text{IO } \text{out } c) = (\lambda r. \exists \text{out}' \ c'. r = \text{IO } \text{out}' \ c' \wedge B \ \text{out } \text{out}' \wedge C \ c \ c')$
 $\langle \text{proof} \rangle$

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

lemma *finite-generat-outs [simp]*: *finite (generat-outs generat)*
 $\langle \text{proof} \rangle$

lemma *countable-generat-outs [simp]*: *countable (generat-outs generat)*
 $\langle \text{proof} \rangle$

lemma *case-map-generat*:
 $\text{case-generat } \text{pure } \text{io } (\text{map-generat } a \ b \ d \ r) =$
 $\text{case-generat } (\text{pure } \circ a) \ (\lambda \text{out}. \text{io } (b \ \text{out}) \circ d) \ r$
 $\langle \text{proof} \rangle$

lemma *continuation-in-generat-contr*:
 $\neg \text{is-Pure } r \implies \text{continuation } r \in \text{generat-contrs } r$
 $\langle \text{proof} \rangle$

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]*: $\text{dest-IO } \text{generat} = \text{Some } (\text{out}, c) \iff \text{generat} = \text{IO } \text{out } c$
 $\langle \text{proof} \rangle$

lemma *dest-IO-eq-None-iff* [simp]: $\text{dest-IO generat} = \text{None} \longleftrightarrow \text{is-Pure generat}$
 ⟨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

$\text{generat-lub lub1 lub2 lub3 } A =$
 (if $\exists x \in A. \text{is-Pure } x$ then $\text{Pure} (\text{lub1 } (\text{result } ' (A \cap \{f. \text{is-Pure } f\})))$
 else $\text{IO} (\text{lub2 } (\text{output } ' (A \cap \{f. \neg \text{is-Pure } f\}))) (\text{lub3 } (\text{continuation } ' (A \cap \{f. \neg \text{is-Pure } f\})))$)

lemma *is-Pure-generat-lub* [simp]:
 $\text{is-Pure } (\text{generat-lub lub1 lub2 lub3 } A) \longleftrightarrow (\exists x \in A. \text{is-Pure } x)$
 ⟨proof⟩

lemma *result-generat-lub* [simp]:
 $\exists x \in A. \text{is-Pure } x \Longrightarrow \text{result } (\text{generat-lub lub1 lub2 lub3 } A) = \text{lub1 } (\text{result } ' (A \cap \{f. \text{is-Pure } f\})))$
 ⟨proof⟩

lemma *output-generat-lub*:
 $\forall x \in A. \neg \text{is-Pure } x \Longrightarrow \text{output } (\text{generat-lub lub1 lub2 lub3 } A) = \text{lub2 } (\text{output } ' (A \cap \{f. \neg \text{is-Pure } f\})))$
 ⟨proof⟩

lemma *continuation-generat-lub*:
 $\forall x \in A. \neg \text{is-Pure } x \Longrightarrow \text{continuation } (\text{generat-lub lub1 lub2 lub3 } A) = \text{lub3 } (\text{continuation } ' (A \cap \{f. \neg \text{is-Pure } f\})))$
 ⟨proof⟩

lemma *generat-lub-map* [simp]:
 $\text{generat-lub lub1 lub2 lub3 } (\text{map-generat } f \ g \ h \ ' A) = \text{generat-lub } (\text{lub1} \circ (' f) (\text{lub2} \circ (' g) (\text{lub3} \circ (' h) A)$
 ⟨proof⟩

lemma *map-generat-lub* [simp]:
 $\text{map-generat } f \ g \ h (\text{generat-lub lub1 lub2 lub3 } A) = \text{generat-lub } (f \circ \text{lub1}) (g \circ \text{lub2}) (h \circ \text{lub3}) A$
 ⟨proof⟩

abbreviation $\text{generat-lub}' :: ('cont\ set \Rightarrow 'cont') \Rightarrow ('a, 'out, 'cont)\ generat\ set$
 $\Rightarrow ('a, 'out, 'cont')\ generat$
where $\text{generat-lub}' \equiv \text{generat-lub}\ (\lambda A.\ THE\ x.\ x \in A)\ (\lambda A.\ THE\ x.\ x \in A)$
end

theory *Generative-Probabilistic-Value* **imports**

Resumption

Generat

HOL-Types-To-Sets.Types-To-Sets

begin

hide-const (**open**) *Done*

4.2 Type definition

context notes $[[bnf-internals]]$ **begin**

codatatype $(\text{results}'\text{-gpv}: 'a, \text{outs}'\text{-gpv}: 'out, 'in)\ \text{gpv}$
 $=\ GPV\ (\text{the-gpv}: ('a, 'out, 'in) \Rightarrow ('a, 'out, 'in)\ \text{gpv})\ \text{generat}\ \text{spmf}$

end

declare $\text{gpv.rel-eq}\ [\text{relator-eq}]$

Reactive values are like generative, except that they take an input first.

type-synonym $('a, 'out, 'in)\ \text{rpv} = 'in \Rightarrow ('a, 'out, 'in)\ \text{gpv}$
 $\langle ML \rangle$
typ $('a, 'out, 'in)\ \text{rpv}$

Effectively, $('a, 'out, 'in)\ \text{gpv}$ and $('a, 'out, 'in)\ \text{rpv}$ are mutually recursive.

lemma $\text{eq-GPV-iff}: f = GPV\ g \longleftrightarrow \text{the-gpv}\ f = g$
 $\langle proof \rangle$

declare $\text{gpv.set}[simp\ del]$

declare $\text{gpv.set-map}[simp]$

lemma $\text{rel-gpv-def}'$:

$\text{rel-gpv}\ A\ B\ \text{gpv}\ \text{gpv}' \longleftrightarrow$
 $(\exists\ \text{gpv}''.\ (\forall\ (x, y) \in \text{results}'\text{-gpv}\ \text{gpv}''.\ A\ x\ y) \wedge (\forall\ (x, y) \in \text{outs}'\text{-gpv}\ \text{gpv}''.\ B\ x$
 $y) \wedge$
 $\text{map-gpv}\ \text{fst}\ \text{fst}\ \text{gpv}'' = \text{gpv} \wedge \text{map-gpv}\ \text{snd}\ \text{snd}\ \text{gpv}'' = \text{gpv}')$
 $\langle proof \rangle$

definition $\text{results}'\text{-rpv} :: ('a, 'out, 'in)\ \text{rpv} \Rightarrow 'a\ \text{set}$
where $\text{results}'\text{-rpv}\ \text{rpv} = \text{range}\ \text{rpv} \gg \text{results}'\text{-gpv}$

definition $outs'-rpv :: ('a, 'out, 'in) rpv \Rightarrow 'out\ set$
where $outs'-rpv\ rpv = range\ rpv \ggg\ outs'-gpv$

abbreviation $rel-rpv$

$:: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('out \Rightarrow 'out' \Rightarrow bool)$
 $\Rightarrow ('in \Rightarrow ('a, 'out, 'in)\ gpv) \Rightarrow ('in \Rightarrow ('b, 'out', 'in)\ gpv) \Rightarrow bool$

where $rel-rpv\ A\ B \equiv rel-fun\ (=)\ (rel-gpv\ A\ B)$

lemma $in-results'-rpv\ [iff]: x \in results'-rpv\ rpv \longleftrightarrow (\exists\ input.\ x \in results'-gpv\ (rpv\ input))$
 $\langle proof \rangle$

lemma $in-outs-rpv\ [iff]: out \in outs'-rpv\ rpv \longleftrightarrow (\exists\ input.\ out \in outs'-gpv\ (rpv\ input))$
 $\langle proof \rangle$

lemma $results'-GPV\ [simp]:$
 $results'-gpv\ (GPV\ r) =$
 $(set-spmf\ r \ggg\ generat-pures) \cup$
 $((set-spmf\ r \ggg\ generat-contrs) \ggg\ results'-rpv)$
 $\langle proof \rangle$

lemma $outs'-GPV\ [simp]:$
 $outs'-gpv\ (GPV\ r) =$
 $(set-spmf\ r \ggg\ generat-outs) \cup$
 $((set-spmf\ r \ggg\ generat-contrs) \ggg\ outs'-rpv)$
 $\langle proof \rangle$

lemma $outs'-gpv-unfold:$
 $outs'-gpv\ r =$
 $(set-spmf\ (the-gpv\ r) \ggg\ generat-outs) \cup$
 $((set-spmf\ (the-gpv\ r) \ggg\ generat-contrs) \ggg\ outs'-rpv)$
 $\langle proof \rangle$

lemma $outs'-gpv-induct\ [consumes\ 1,\ case-names\ Out\ Cont,\ induct\ set:\ outs'-gpv]:$
assumes $x \in outs'-gpv\ gpv$
and $Out: \bigwedge generat\ gpv.\ \llbracket generat \in set-spmf\ (the-gpv\ gpv); x \in generat-outs\ generat \rrbracket \Longrightarrow P\ gpv$
and $Cont: \bigwedge generat\ gpv\ c\ input.\$
 $\llbracket generat \in set-spmf\ (the-gpv\ gpv); c \in generat-contrs\ generat; x \in outs'-gpv\ (c\ input); P\ (c\ input) \rrbracket \Longrightarrow P\ gpv$
shows $P\ gpv$
 $\langle proof \rangle$

lemma $outs'-gpv-cases\ [consumes\ 1,\ case-names\ Out\ Cont,\ cases\ set:\ outs'-gpv]:$
assumes $x \in outs'-gpv\ gpv$
obtains $(Out)\ generat$ **where** $generat \in set-spmf\ (the-gpv\ gpv)\ x \in generat-outs\ generat$

| (Cont) generat c input **where** generat \in set-spmf (the-gpv gpv) c \in generat-contrs generat $x \in$ outs'-gpv (c input)
 <proof>

lemma outs'-gpvI [intro?]:

shows outs'-gpv-Out: \llbracket generat \in set-spmf (the-gpv gpv); $x \in$ generat-outs generat $\rrbracket \implies x \in$ outs'-gpv gpv

and outs'-gpv-Cont: \llbracket generat \in set-spmf (the-gpv gpv); c \in generat-contrs generat; $x \in$ outs'-gpv (c input) $\rrbracket \implies x \in$ outs'-gpv gpv

<proof>

lemma results'-gpv-induct [consumes 1, case-names Pure Cont, induct set: results'-gpv]:

assumes x: $x \in$ results'-gpv gpv

and Pure: \bigwedge generat gpv. \llbracket generat \in set-spmf (the-gpv gpv); $x \in$ generat-pures generat $\rrbracket \implies P$ gpv

and Cont: \bigwedge generat gpv c input.

\llbracket generat \in set-spmf (the-gpv gpv); c \in generat-contrs generat; $x \in$ results'-gpv (c input); P (c input) $\rrbracket \implies P$ gpv

shows P gpv

<proof>

lemma results'-gpv-cases [consumes 1, case-names Pure Cont, cases set: results'-gpv]:

assumes x \in results'-gpv gpv

obtains (Pure) generat **where** generat \in set-spmf (the-gpv gpv) $x \in$ generat-pures generat

| (Cont) generat c input **where** generat \in set-spmf (the-gpv gpv) c \in generat-contrs generat $x \in$ results'-gpv (c input)

<proof>

lemma results'-gpvI [intro?]:

shows results'-gpv-Pure: \llbracket generat \in set-spmf (the-gpv gpv); $x \in$ generat-pures generat $\rrbracket \implies x \in$ results'-gpv gpv

and results'-gpv-Cont: \llbracket generat \in set-spmf (the-gpv gpv); c \in generat-contrs generat; $x \in$ results'-gpv (c input) $\rrbracket \implies x \in$ results'-gpv gpv

<proof>

lemma left-unique-rel-gpv [transfer-rule]:

\llbracket left-unique A; left-unique B $\rrbracket \implies$ left-unique (rel-gpv A B)

<proof>

lemma right-unique-rel-gpv [transfer-rule]:

\llbracket right-unique A; right-unique B $\rrbracket \implies$ right-unique (rel-gpv A B)

<proof>

lemma bi-unique-rel-gpv [transfer-rule]:

\llbracket bi-unique A; bi-unique B $\rrbracket \implies$ bi-unique (rel-gpv A B)

<proof>

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

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

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

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*:
 $\llbracket \text{pred-gpv } P \ Q \ x; \bigwedge a. \llbracket a \in \text{results}'\text{-gpv } x; P \ a \rrbracket \Longrightarrow P' \ a; \bigwedge b. \llbracket b \in \text{outs}'\text{-gpv } x; Q \ b \rrbracket \Longrightarrow Q' \ b \rrbracket \Longrightarrow \text{pred-gpv } P' \ Q' \ x$
 $\langle \text{proof} \rangle$

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?*]:
 $\llbracket \text{rel-gpv } R \ R' \ x \ y; \text{pred-gpv } P \ P' \ x; \text{pred-gpv } Q \ Q' \ y \rrbracket \Longrightarrow \text{rel-gpv } (R \upharpoonright P \otimes Q) \ (R' \upharpoonright P' \otimes Q') \ x \ y$
 $\langle \text{proof} \rangle$

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 $map\text{-}gpv' :: ('a \Rightarrow 'b) \Rightarrow ('out \Rightarrow 'out') \Rightarrow ('ret' \Rightarrow 'ret) \Rightarrow ('a, 'out, 'ret) gpv \Rightarrow ('b, 'out', 'ret') gpv$

where

$map\text{-}gpv' f g h gpv =$
 $GPV (map\text{-}spmf (map\text{-}generat f g ((\circ) (map\text{-}gpv' f g h))) (map\text{-}spmf (map\text{-}generat id id (map\text{-}fun h id)) (the\text{-}gpv gpv)))$

declare $map\text{-}gpv'.sel$ [*simp del*]

lemma $map\text{-}gpv'.sel$ [*simp*]:

$the\text{-}gpv (map\text{-}gpv' f g h gpv) = map\text{-}spmf (map\text{-}generat f g (h \text{----}> map\text{-}gpv' f g h)) (the\text{-}gpv gpv)$
 $\langle proof \rangle$

lemma $map\text{-}gpv'\text{-}GPV$ [*simp*]:

$map\text{-}gpv' f g h (GPV p) = GPV (map\text{-}spmf (map\text{-}generat f g (h \text{----}> map\text{-}gpv' f g h)) p)$
 $\langle proof \rangle$

lemma $map\text{-}gpv'\text{-}id$: $map\text{-}gpv' id id id = id$

$\langle proof \rangle$

lemma $map\text{-}gpv'\text{-}comp$: $map\text{-}gpv' f g h (map\text{-}gpv' f' g' h' gpv) = map\text{-}gpv' (f \circ f') (g \circ g') (h' \circ h) gpv$

$\langle proof \rangle$

functor gpv : $map\text{-}gpv'$ $\langle proof \rangle$

lemma $map\text{-}gpv\text{-}conv\text{-}map\text{-}gpv'$: $map\text{-}gpv f g = map\text{-}gpv' f g id$

$\langle proof \rangle$

coinductive $rel\text{-}gpv'' :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('out \Rightarrow 'out' \Rightarrow bool) \Rightarrow ('ret \Rightarrow 'ret' \Rightarrow bool) \Rightarrow ('a, 'out, 'ret) gpv \Rightarrow ('b, 'out', 'ret') gpv \Rightarrow bool$

for $A C R$

where

$rel\text{-}spmf (rel\text{-}generat A C (R \text{====}> rel\text{-}gpv'' A C R)) (the\text{-}gpv gpv) (the\text{-}gpv gpv')$
 $\implies rel\text{-}gpv'' A C R gpv gpv'$

lemma $rel\text{-}gpv''\text{-}coinduct$ [*consumes 1, case-names rel-gpv'', coinduct pred: rel-gpv''*]:

$\llbracket X gpv gpv';$
 $\bigwedge gpv gpv'. X gpv gpv'$
 $\implies rel\text{-}spmf (rel\text{-}generat A C (R \text{====}> (\lambda gpv gpv'. X gpv gpv' \vee rel\text{-}gpv'' A C R gpv gpv'))$
 $(the\text{-}gpv gpv) (the\text{-}gpv gpv') \rrbracket$
 $\implies rel\text{-}gpv'' A C R gpv gpv'$

<proof>

lemma *rel-gpv''D*:

$rel-gpv'' A C R gpv gpv'$
 $\implies rel-spmf (rel-generat A C (R ==> rel-gpv'' A C R)) (the-gpv gpv) (the-gpv gpv')$
<proof>

lemma *rel-gpv''-GPV [simp]*:

$rel-gpv'' A C R (GPV p) (GPV q) \longleftrightarrow$
 $rel-spmf (rel-generat A C (R ==> rel-gpv'' A C R)) p q$
<proof>

lemma *rel-gpv-conv-rel-gpv''*: $rel-gpv A C = rel-gpv'' A C (=)$

<proof>

lemma *rel-gpv''-eq* :

$rel-gpv'' (=) (=) (=) = (=)$
<proof>

lemma *rel-gpv''-mono*:

assumes $A \leq A' C \leq C' R' \leq R$
shows $rel-gpv'' A C R \leq rel-gpv'' A' C' R'$
<proof>

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

<proof>

lemma *rel-gpv''-pos-distr*:

$rel-gpv'' A C R OO rel-gpv'' A' C' R' \leq rel-gpv'' (A OO A') (C OO C') (R OO R')$
<proof>

lemma *left-unique-rel-gpv''*:

$\llbracket left-unique A; left-unique C; left-total R \rrbracket \implies left-unique (rel-gpv'' A C R)$
<proof>

lemma *right-unique-rel-gpv''*:

$\llbracket right-unique A; right-unique C; right-total R \rrbracket \implies right-unique (rel-gpv'' A C R)$
<proof>

lemma *bi-unique-rel-gpv'' [transfer-rule]*:

$\llbracket bi-unique A; bi-unique C; bi-total R \rrbracket \implies bi-unique (rel-gpv'' A C R)$
<proof>

lemma *rel-gpv''-neg-distr*:

assumes R : *left-unique R right-total R*

and R' : *right-unique* R' *left-total* R'
shows $\text{rel-gpv}''(A \text{ OO } A') (C \text{ OO } C') (R \text{ OO } R') \leq \text{rel-gpv}'' A C R \text{ OO } \text{rel-gpv}'' A' C' R'$
 $\langle \text{proof} \rangle$

lemma *left-total-rel-gpv'*:
 $\llbracket \text{left-total } A; \text{left-total } C; \text{left-unique } R; \text{right-total } R \rrbracket \implies \text{left-total } (\text{rel-gpv}'' A C R)$
 $\langle \text{proof} \rangle$

lemma *right-total-rel-gpv'*:
 $\llbracket \text{right-total } A; \text{right-total } C; \text{right-unique } R; \text{left-total } R \rrbracket \implies \text{right-total } (\text{rel-gpv}'' A C R)$
 $\langle \text{proof} \rangle$

lemma *bi-total-rel-gpv'* [*transfer-rule*]:
 $\llbracket \text{bi-total } A; \text{bi-total } C; \text{bi-unique } R; \text{bi-total } R \rrbracket \implies \text{bi-total } (\text{rel-gpv}'' A C R)$
 $\langle \text{proof} \rangle$

lemma *rel-fun-conversep-grp-grp*:
 $\text{rel-fun } (\text{conversep } (\text{BNF-Def.Grp UNIV } f)) (\text{BNF-Def.Grp } B g) = \text{BNF-Def.Grp } \{x. (x \circ f) ' \text{UNIV} \subseteq B\} (\text{map-fun } f g)$
 $\langle \text{proof} \rangle$

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* $(\text{rel-gpv}'' R1 R2 R3)$ $(\text{map-gpv}' Abs1 Abs2 Rep3)$ $(\text{map-gpv}' Rep1 Rep2 Abs3)$ $(\text{rel-gpv}'' T1 T2 T3)$
(is *Quotient* $?R$ $?abs$ $?rep$ $?T$)
 $\langle \text{proof} \rangle$

lemma *rel-gpv''-Grp*:
 $\text{rel-gpv}'' (\text{BNF-Def.Grp } A f) (\text{BNF-Def.Grp } B g) (\text{BNF-Def.Grp UNIV } h)^{-1-1}$
 $=$
 $\text{BNF-Def.Grp } \{x. \text{results}'\text{-gpv } x \subseteq A \wedge \text{outs}'\text{-gpv } x \subseteq B\} (\text{map-gpv}' f g h)$
(is $?lhs = ?rhs$)
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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'*:

$((S \text{====>} \text{rel-spmf } (\text{rel-generat } A \ C \ (R \text{====>} \text{rel-sum } (\text{rel-gpv}'' \ A \ C \ R) \ S)))$
 $\text{====>} \ S \text{====>} \ \text{rel-gpv}'' \ A \ C \ R)$
corec-gpv corec-gpv
<proof>

lemma *map-gpv'-parametric [transfer-rule]*:

$((A \text{====>} \ A') \text{====>} \ (C \text{====>} \ C') \text{====>} \ (R' \text{====>} \ R) \text{====>} \ \text{rel-gpv}''$
 $A \ C \ R \text{====>} \ \text{rel-gpv}'' \ A' \ C' \ R') \ \text{map-gpv}' \ \text{map-gpv}'$
<proof>

lemma *map-gpv-parametric'*: $((A \text{====>} \ A') \text{====>} \ (C \text{====>} \ C') \text{====>} \ \text{rel-gpv}''$
 $A \ C \ R \text{====>} \ \text{rel-gpv}'' \ A' \ C' \ R) \ \text{map-gpv} \ \text{map-gpv}$
<proof>

end

4.4 Simple, derived operations

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

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

primcorec *lift-spmf* :: $'a \text{ spmf} \Rightarrow ('a, 'out, 'in) \text{ gpv}$
where *the-gpv* (*lift-spmf* *p*) = *map-spmf* *Pure* *p*

definition *Fail* :: $('a, 'out, 'in) \text{ gpv}$
where *Fail* = *GPV* (*return-pmf* *None*)

definition *React* :: $('in \Rightarrow 'out \times ('a, 'out, 'in) \text{ rpv}) \Rightarrow ('a, 'out, 'in) \text{ rpv}$
where *React* *f* *input* = *case-prod* *Pause* (*f* *input*)

definition *rFail* :: $('a, 'out, 'in) \text{ rpv}$
where *rFail* = $(\lambda-. \text{Fail})$

lemma *Done-inject [simp]*: $\text{Done } x = \text{Done } y \iff x = y$
<proof>

lemma *Pause-inject [simp]*: $\text{Pause } out \ c = \text{Pause } out' \ c' \iff out = out' \wedge c = c'$
<proof>

lemma *[simp]*:

shows *Done-neq-Pause*: $\text{Done } x \neq \text{Pause } out \ c$
and *Pause-neq-Done*: $\text{Pause } out \ c \neq \text{Done } x$

$\langle proof \rangle$

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]: $lift-spmf (return-spmf\ x) = Done\ x$
 $\langle proof \rangle$

lemma *lift-spmf-None* [simp]: $lift-spmf (return-pmf\ None) = Fail$
 $\langle proof \rangle$

lemma *the-gpv-lift-spmf* [simp]: $the-gpv (lift-spmf\ r) = map-spmf\ Pure\ r$
 $\langle proof \rangle$

lemma *outs'-gpv-lift-spmf* [simp]: $outs'-gpv (lift-spmf\ p) = \{\}$
 $\langle proof \rangle$

lemma *results'-gpv-lift-spmf* [simp]: $results'-gpv (lift-spmf\ p) = set-spmf\ p$
 $\langle proof \rangle$

lemma *pred-gpv-lift-spmf* [simp]: $pred-gpv\ P\ Q (lift-spmf\ p) = pred-spmf\ P\ p$
 $\langle proof \rangle$

lemma *lift-spmf-inject* [simp]: $lift-spmf\ p = lift-spmf\ q \iff p = q$
 $\langle proof \rangle$

lemma *map-lift-spmf*: $map-gpv\ f\ g (lift-spmf\ p) = lift-spmf (map-spmf\ f\ p)$
 $\langle proof \rangle$

lemma *lift-map-spmf*: $lift-spmf (map-spmf\ f\ p) = map-gpv\ f\ id (lift-spmf\ p)$
 $\langle proof \rangle$

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]*: $Fail = GPV\ f \longleftrightarrow f = return-pmf\ None$
<proof>

lemma *outs'-gpv-Fail* *[simp]*: $outs'-gpv\ Fail = \{\}$
<proof>

lemma *results'-gpv-Fail* *[simp]*: $results'-gpv\ Fail = \{\}$
<proof>

lemma *pred-gpv-Fail* *[simp]*: $pred-gpv\ P\ Q\ Fail$
<proof>

lemma *React-inject* *[iff]*: $React\ f = React\ f' \longleftrightarrow f = f'$
<proof>

lemma *React-apply* *[simp]*: $f\ input = (out, c) \implies React\ f\ input = Pause\ out\ c$
<proof>

lemma *rFail-apply* *[simp]*: $rFail\ input = Fail$
<proof>

lemma *[simp]*:
shows *rFail-neq-React*: $rFail \neq React\ f$
and *React-neq-rFail*: $React\ f \neq rFail$
<proof>

lemma *rel-gpv-FailI* *[simp]*: $rel-gpv\ A\ C\ Fail\ Fail$

$\langle \text{proof} \rangle$

lemma *rel-gpv-Done* [iff]: $\text{rel-gpv } A \ C \ (\text{Done } x) \ (\text{Done } y) \longleftrightarrow A \ x \ y$
 $\langle \text{proof} \rangle$

lemma *rel-gpv''-Done* [iff]: $\text{rel-gpv}'' \ A \ C \ R \ (\text{Done } x) \ (\text{Done } y) \longleftrightarrow A \ x \ y$
 $\langle \text{proof} \rangle$

lemma *rel-gpv-Pause* [iff]:
 $\text{rel-gpv } A \ C \ (\text{Pause } \text{out}' \ c) \ (\text{Pause } \text{out}' \ c') \longleftrightarrow C \ \text{out} \ \text{out}' \ \wedge \ (\forall x. \text{rel-gpv } A \ C \ (c \ x) \ (c' \ x))$
 $\langle \text{proof} \rangle$

lemma *rel-gpv''-Pause* [iff]:
 $\text{rel-gpv}'' \ A \ C \ R \ (\text{Pause } \text{out}' \ c) \ (\text{Pause } \text{out}' \ c') \longleftrightarrow C \ \text{out} \ \text{out}' \ \wedge \ (\forall x \ x'. R \ x \ x' \longrightarrow \text{rel-gpv}'' \ A \ C \ R \ (c \ x) \ (c' \ x'))$
 $\langle \text{proof} \rangle$

lemma *rel-gpv-lift-spmf* [iff]: $\text{rel-gpv } A \ C \ (\text{lift-spmf } p) \ (\text{lift-spmf } q) \longleftrightarrow \text{rel-spmf } A \ p \ q$
 $\langle \text{proof} \rangle$

lemma *rel-gpv''-lift-spmf* [iff]:
 $\text{rel-gpv}'' \ A \ C \ R \ (\text{lift-spmf } p) \ (\text{lift-spmf } q) \longleftrightarrow \text{rel-spmf } A \ p \ q$
 $\langle \text{proof} \rangle$

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

lemmas *Fail-parametric* [transfer-rule] = *rel-gpv-FailI*

lemma *Fail-parametric'* [simp]: $\text{rel-gpv}'' \ A \ C \ R \ \text{Fail} \ \text{Fail}$
 $\langle \text{proof} \rangle$

lemma *Done-parametric* [transfer-rule]: $(A \ ==\ ==\ ==\ ==\ > \text{rel-gpv } A \ C) \ \text{Done} \ \text{Done}$
 $\langle \text{proof} \rangle$

lemma *Done-parametric'*: $(A \ ==\ ==\ ==\ ==\ > \text{rel-gpv}'' \ A \ C \ R) \ \text{Done} \ \text{Done}$
 $\langle \text{proof} \rangle$

lemma *Pause-parametric* [transfer-rule]:
 $(C \ ==\ ==\ ==\ ==\ > ((=) \ ==\ ==\ ==\ ==\ > \text{rel-gpv } A \ C) \ ==\ ==\ ==\ ==\ > \text{rel-gpv } A \ C) \ \text{Pause} \ \text{Pause}$
 $\langle \text{proof} \rangle$

lemma *Pause-parametric'*:
 $(C \ ==\ ==\ ==\ ==\ > (R \ ==\ ==\ ==\ ==\ > \text{rel-gpv}'' \ A \ C \ R) \ ==\ ==\ ==\ ==\ > \text{rel-gpv}'' \ A \ C \ R) \ \text{Pause} \ \text{Pause}$
 $\langle \text{proof} \rangle$

lemma *lift-spmf-parametric* [transfer-rule]:
 $(\text{rel-spmf } A \ ==\ ==\ ==\ ==\ > \text{rel-gpv } A \ C) \ \text{lift-spmf} \ \text{lift-spmf}$
 $\langle \text{proof} \rangle$

lemma *lift-spmf-parametric'*:
 (*rel-spmf* A ==> *rel-gpv''* A C R) *lift-spmf lift-spmf*
 <proof>
end

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

lemma *map-gpv'-Done* [*simp*]: *map-gpv'* f g h (Done x) = Done (f x)
 <proof>

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

lemma *map-gpv'-Pause* [*simp*]: *map-gpv'* f g h (Pause x c) = Pause (g x) (*map-gpv'*
 f g h ∘ c ∘ h)
 <proof>

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

lemma *map-gpv'-Fail* [*simp*]: *map-gpv'* f g h Fail = Fail
 <proof>

4.5 Monad structure

primcorec *bind-gpv* :: ('a, 'out, 'in) *gpv* ⇒ ('a ⇒ ('b, 'out, 'in) *gpv*) ⇒ ('b, 'out,
 'in) *gpv*

where

the-gpv (*bind-gpv* r f) =
map-spmf (*map-generat id id* ((∘) (*case-sum id* (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))
 })

$\langle proof \rangle$

lemma *bind-gpv-code-cong*: $f = f' \implies \text{bind-gpv } f \ g = \text{bind-gpv } f' \ g$ $\langle proof \rangle$
 $\langle ML \rangle$

lemma *bind-gpv-sel*:

$\text{the-gpv } (r \ggg f) =$
do {
 $\text{generat} \leftarrow \text{the-gpv } r;$
 case generat of $\text{Pure } x \Rightarrow \text{the-gpv } (f \ x)$
 | $\text{IO out } c \Rightarrow \text{return-spmf } (\text{IO out } (\lambda \text{input}. \text{bind-gpv } (c \ \text{input}) \ f))$
}

$\langle proof \rangle$

lemma *bind-gpv-sel' [simp]*:

$\text{the-gpv } (r \ggg f) =$
do {
 $\text{generat} \leftarrow \text{the-gpv } r;$
 if $\text{is-Pure } \text{generat}$ then $\text{the-gpv } (f \ (\text{result } \text{generat}))$
 else $\text{return-spmf } (\text{IO } (\text{output } \text{generat}) \ (\lambda \text{input}. \text{bind-gpv } (\text{continuation } \text{generat} \ \text{input}) \ f))$
}

$\langle proof \rangle$

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

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

lemma *if-distrib-bind-gpv2 [if-distrib]*:

$\text{bind-gpv } \text{gpv } (\lambda y. \text{if } b \text{ then } f \ y \ \text{else } g \ y) = (\text{if } b \text{ then } \text{bind-gpv } \text{gpv } f \ \text{else } \text{bind-gpv } \text{gpv } g)$
 $\langle proof \rangle$

lemma *lift-spmf-bind*: $\text{lift-spmf } r \ggg f = \text{GPV } (r \ggg \text{the-gpv } \circ f)$
 $\langle proof \rangle$

lemma *the-gpv-bind-gpv-lift-spmf [simp]*:

$\text{the-gpv } (\text{bind-gpv } (\text{lift-spmf } p) \ f) = \text{bind-spmf } p \ (\text{the-gpv } \circ f)$
 $\langle proof \rangle$

lemma *lift-spmf-bind-spmf*: $\text{lift-spmf } (p \ggg f) = \text{lift-spmf } p \ggg (\lambda x. \text{lift-spmf } (f \ x))$

$\langle proof \rangle$

lemma *lift-bind-spmf*: $\text{lift-spmf } (\text{bind-spmf } p \ f) = \text{bind-gpv } (\text{lift-spmf } p) \ (\text{lift-spmf } \circ f)$

$\langle proof \rangle$

lemma *GPV-bind*:

$GPV\ f \ggg\ g =$
 $GPV\ (f \ggg\ (\lambda\ generat.\ case\ generat\ of\ Pure\ x \Rightarrow\ the\ gpv\ (g\ x) \mid IO\ out\ c \Rightarrow$
 $return\ spmf\ (IO\ out\ (\lambda\ input.\ c\ input \ggg\ g))))$
<proof>

lemma *GPV-bind'*:

$GPV\ f \ggg\ g = GPV\ (f \ggg\ (\lambda\ generat.\ if\ is\ Pure\ generat\ then\ the\ gpv\ (g\ (result$
 $generat))\ else\ return\ spmf\ (IO\ (output\ generat)\ (\lambda\ input.\ continuation\ generat\ in-$
 $put \ggg\ g))))$
<proof>

lemma *bind-gpv-assoc*:

fixes $f :: ('a, 'out, 'in)\ gpv$
shows $(f \ggg\ g) \ggg\ h = f \ggg\ (\lambda\ x.\ g\ x \ggg\ h)$
<proof>

lemma *map-gpv-bind-gpv*: $map\ gpv\ f\ g\ (bind\ gpv\ gpv\ h) = bind\ gpv\ (map\ gpv\ id\ g$
 $gpv)\ (\lambda\ x.\ map\ gpv\ f\ g\ (h\ x))$
<proof>

lemma *map-gpv-id-bind-gpv*: $map\ gpv\ f\ id\ (bind\ gpv\ gpv\ g) = bind\ gpv\ gpv\ (map\ gpv$
 $f\ id \circ g)$
<proof>

lemma *map-gpv-conv-bind*:

$map\ gpv\ f\ (\lambda\ x.\ x)\ x = bind\ gpv\ x\ (\lambda\ x.\ Done\ (f\ x))$
<proof>

lemma *bind-map-gpv*: $bind\ gpv\ (map\ gpv\ f\ id\ gpv)\ g = bind\ gpv\ gpv\ (g \circ f)$
<proof>

lemma *outs-bind-gpv*:

$outs'\ gpv\ (bind\ gpv\ x\ f) = outs'\ gpv\ x \cup (\bigcup x \in results'\ gpv\ x.\ outs'\ gpv\ (f\ x))$
(is ?lhs = ?rhs)
<proof>

lemma *bind-gpv-Fail [simp]*: $Fail \ggg\ f = Fail$
<proof>

lemma *bind-gpv-eq-Fail*:

$bind\ gpv\ gpv\ f = Fail \iff (\forall x \in set\ spmf\ (the\ gpv\ gpv).\ is\ Pure\ x) \wedge (\forall x \in results'\ gpv$
 $gpv.\ f\ x = Fail)$
(is ?lhs = ?rhs)
<proof>

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

lemma *bind-gpv-parametric* [*transfer-rule*]:

$(rel-gpv\ A\ C\ ==> (A\ ==> rel-gpv\ B\ C)\ ==> rel-gpv\ B\ C)\ bind-gpv$
bind-gpv
<proof>

lemma *bind-gpv-parametric'*:

$(rel-gpv''\ A\ C\ R\ ==> (A\ ==> rel-gpv''\ B\ C\ R)\ ==> rel-gpv''\ B\ C\ R)$
bind-gpv bind-gpv
<proof>

end

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

<proof>

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

<proof>

lemma *rel-gpv-bindI*:

$\llbracket rel-gpv\ A\ C\ gpv\ gpv'; \bigwedge x\ y.\ A\ x\ y \implies rel-gpv\ B\ C\ (f\ x)\ (g\ y) \rrbracket$
 $\implies rel-gpv\ B\ C\ (bind-gpv\ gpv\ f)\ (bind-gpv\ gpv'\ g)$
<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* = (*λinput. bind-gpv (rpv input) f*)

lemma *bind-rpv-apply* [*simp*]: *bind-rpv rpv f input* = *bind-gpv (rpv input) f*

<proof>

ad hoc overloading *Monad-Syntax.bind 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 (λrpv. bind-rpv*

$rpv\ g) \circ f)$
 $\langle proof \rangle$

lemma *bind-rpv-assoc*: $bind-rpv\ (bind-rpv\ rpv\ f)\ g = bind-rpv\ rpv\ ((\lambda gpv.\ bind-gpv\ gpv\ g) \circ f)$
 $\langle proof \rangle$

lemma *bind-rpv-Done* [*simp*]: $bind-rpv\ Done\ f = f$
 $\langle proof \rangle$

lemma *results'-rpv-Done* [*simp*]: $results'-rpv\ Done = UNIV$
 $\langle proof \rangle$

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 $(R\ OO\ R' \implies S\ OO\ S') \leq ((R \implies S)\ OO\ (R' \implies S'))$

$\langle proof \rangle$

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 $rel-gpv' :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a, 'out, 'in)\ gpv \Rightarrow ('b, 'out, 'in)\ gpv \Rightarrow bool$

where $rel-gpv'\ A = rel-gpv\ A (=)$

lemma *rel-gpv'-eq* [*relator-eq*]: $rel-gpv'\ (=) = (=)$
 $\langle proof \rangle$

lemma *rel-gpv'-mono* [*relator-mono*]: $A \leq B \implies rel-gpv'\ A \leq rel-gpv'\ B$
 $\langle proof \rangle$

lemma *rel-gpv'-distr* [*relator-distr*]: $rel-gpv'\ A\ OO\ rel-gpv'\ B = rel-gpv'\ (A\ OO\ B)$
 $\langle proof \rangle$

lemma *left-unique-rel-gpv'* [*transfer-rule*]: $left-unique\ A \implies left-unique\ (rel-gpv'\ A)$
 $\langle proof \rangle$

lemma *right-unique-rel-gpv'* [*transfer-rule*]: $right-unique\ A \implies right-unique\ (rel-gpv'\ A)$
 $\langle proof \rangle$

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

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

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

lemma *bi-total-rel-gpv'* [*transfer-rule*]: *bi-total A* \implies *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* \Rightarrow (*'a, 'out, 'in*) *gpv* \Rightarrow *bool*
where *cr-spmf-gpv p gpv* \longleftrightarrow *gpv = lift-spmf p*

definition *spmf-of-gpv* :: (*'a, 'out, 'in*) *gpv* \Rightarrow *'a spmf*
where *spmf-of-gpv gpv* = (*THE p. gpv = lift-spmf p*)

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

lemma *rel-spmf-setD2*:
 [*rel-spmf A p q; y* \in *set-spmf q*] \implies $\exists x \in$ *set-spmf p. A x y*
 ⟨*proof*⟩

lemma *rel-gpv-lift-spmf1*: *rel-gpv A B (lift-spmf p) gpv* \longleftrightarrow ($\exists q. gpv = lift-spmf q \wedge rel-spmf A p q$)
 ⟨*proof*⟩

lemma *rel-gpv-lift-spmf2*: *rel-gpv A B gpv (lift-spmf q)* \longleftrightarrow ($\exists p. gpv = lift-spmf p \wedge rel-spmf A p q$)
 ⟨*proof*⟩

definition *pcr-spmf-gpv* :: (*'a* \Rightarrow *'b* \Rightarrow *bool*) \Rightarrow *'a spmf* \Rightarrow (*'b, 'out, 'in*) *gpv* \Rightarrow *bool*

where *pcr-spmf-gpv A* = *cr-spmf-gpv OO rel-gpv A (=)*

lemma *pcr-cr-eq-spmf-gpv*: *pcr-spmf-gpv (=)* = *cr-spmf-gpv*
 ⟨*proof*⟩

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

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

lemma *right-unique-cr-spmf-gpv*: *right-unique cr-spmf-gpv*
<proof>

lemma *right-unique-pcr-spmf-gpv* [*transfer-rule*]:
right-unique A ==> right-unique (pcr-spmf-gpv A)
<proof>

lemma *bi-unique-cr-spmf-gpv*: *bi-unique cr-spmf-gpv*
<proof>

lemma *bi-unique-pcr-spmf-gpv* [*transfer-rule*]: *bi-unique A ==> bi-unique (pcr-spmf-gpv A)*
<proof>

lemma *left-total-cr-spmf-gpv*: *left-total cr-spmf-gpv*
<proof>

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

context includes *lifting-syntax begin*

lemma *return-spmf-gpv-transfer'*:
((=) ==> cr-spmf-gpv) return-spmf Done
<proof>

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

lemma *bind-spmf-gpv-transfer'*:
(cr-spmf-gpv ==> ((=) ==> cr-spmf-gpv) ==> cr-spmf-gpv) bind-spmf
bind-gpv
<proof>

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

lemma *lift-spmf-gpv-transfer'*:
((=) ==> cr-spmf-gpv) (λx. x) lift-spmf
<proof>

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

lemma *fail-spmf-gpv-transfer'*: *cr-spmf-gpv (return-pmf None) Fail*
<proof>

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

lemma *map-spmf-gpv-transfer'*:
 $((=) \implies R \implies cr\text{-}spmf\text{-}gpv \implies cr\text{-}spmf\text{-}gpv) (\lambda f g. map\text{-}spmf\ f)$
map-gpv
<proof>

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
<proof>

end

end

4.7 Embedding 'a option as a monad

locale *option-to-gpv begin*

interpretation *option-to-spmf <proof>*

interpretation *spmf-to-gpv <proof>*

definition *cr-option-gpv* :: 'a option \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*:
 $cr\text{-}option\text{-}gpv = cr\text{-}spmf\text{-}option^{-1-1} OO cr\text{-}spmf\text{-}gpv$
<proof>

context includes *lifting-syntax begin*

These transfer rules should follow from merging the transfer rules, but this has not yet been implemented.

lemma *return-option-gpv-transfer [transfer-rule]*:
 $((=) \implies cr\text{-}option\text{-}gpv) Some Done$
<proof>

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

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

<proof>

lemma *map-option-gpv-transfer* [*transfer-rule*]:

$((=) \implies R \implies \text{cr-option-gpv} \implies \text{cr-option-gpv}) (\lambda f g. \text{map-option } f)$
map-gpv
<proof>

end

end

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

interpretation *option-le-spmf* *<proof>*

interpretation *spmf-to-gpv* *<proof>*

definition *cr-option-le-gpv* :: *'a option* \Rightarrow (*'a, 'out, 'in*) *gpv* \Rightarrow *bool*

where *cr-option-le-gpv* *x gpv* \longleftrightarrow *gpv* = (*lift-spmf* \circ *return-pmf*) *x* \vee *x* = *None*

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

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

$((=) \implies \text{cr-option-le-gpv}) \text{ Some Done}$
<proof>

lemma *bind-option-gpv-transfer* [*transfer-rule*]:

$(\text{cr-option-le-gpv} \implies ((=) \implies \text{cr-option-le-gpv}) \implies \text{cr-option-le-gpv})$
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*]:

$((((=) \implies (=)) \implies \text{cr-option-le-gpv} \implies \text{cr-option-le-gpv}) \text{ map-option})$
 $(\lambda f. \text{map-gpv } f \text{ id})$
<proof>

end

end

4.8 Embedding resumptions

primcorec *lift-resumption* :: (*'a, 'out, 'in*) *resumption* \Rightarrow (*'a, 'out, 'in*) *gpv*

where

the-gpv (*lift-resumption* *r*) =

(*case* *r* of *resumption.Done None* \Rightarrow *return-pmf None*)

| *resumption.Done* (Some x') => *return-spmf* (Pure x')
 | *resumption.Pause* out c => *map-spmf* (*map-generat id id* ((\circ) *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* \circ *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* \circ 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*]:

$\bigwedge A$. *pred-gpv* A C (*lift-resumption* r) = *pred-resumption* A C r
 <proof>

lemma *lift-resumption-bind*: *lift-resumption* ($r \gg= f$) = *lift-resumption* $r \gg=$
lift-resumption $\circ f$

<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)*
 ⟨*proof*⟩

lemma *the-gpv-bind-assert* [*simp*]:
the-gpv (bind-gpv (assert-gpv b) f) =
bind-spmf (assert-spmf b) (the-gpv \circ f)
 ⟨*proof*⟩

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

primcorec *try-gpv* :: (*'a*, *'call*, *'ret*) *gpv* \Rightarrow (*'a*, *'call*, *'ret*) *gpv* \Rightarrow (*'a*, *'call*, *'ret*)
gpv (TRY - ELSE - [0,60] 59)

where

the-gpv (TRY gpv ELSE gpv') =
*map-spmf (map-generat id id (λ c input. case c input of Inl gpv \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 (λ c input. TRY c input ELSE gpv')) (the-gpv
 gpv) ELSE the-gpv gpv'*
 ⟨*proof*⟩

lemma *try-gpv-Done* [*simp*]: *TRY Done x ELSE gpv'* = *Done x*

$\langle proof \rangle$

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]:
 $(rel\ gpv\ A\ C\ ==>\ rel\ gpv\ A\ C\ ==>\ rel\ gpv\ A\ C)\ try\ gpv\ try\ gpv$
 $\langle proof \rangle$

lemma *try-gpv-parametric'*:
 $(rel\ gpv''\ A\ C\ R\ ==>\ rel\ gpv''\ A\ C\ R\ ==>\ rel\ gpv''\ A\ C\ R)\ try\ gpv\ try\ gpv$
 $\langle proof \rangle$
end

lemma *map-try-gpv*: $map\ gpv\ f\ g\ (TRY\ gpv\ ELSE\ gpv') = TRY\ map\ gpv\ f\ g\ gpv\ ELSE\ map\ gpv\ f\ g\ gpv'$
 $\langle proof \rangle$

lemma *map'-try-gpv*: $map\ gpv'\ f\ g\ h\ (TRY\ gpv\ ELSE\ gpv') = TRY\ map\ gpv'\ f\ g\ h\ gpv\ ELSE\ map\ gpv'\ f\ g\ h\ gpv'$
 $\langle proof \rangle$

lemma *try-bind-assert-gpv*:
 $TRY\ (assert\ gpv\ b\ \gg\ f)\ ELSE\ gpv = (if\ b\ then\ TRY\ (f\ ())\ ELSE\ gpv\ else\ gpv)$
 $\langle proof \rangle$

4.10 Order for ('a, 'out, 'in) gpv

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

where

$ord\ spmf\ (rel\ generat\ (=)\ (=)\ (rel\ fun\ (=)\ ord\ gpv))\ f\ g \Longrightarrow ord\ gpv\ (GPV\ f)\ (GPV\ g)$

inductive-simps *ord-gpv-simps* [*simp*]:
ord-gpv (*GPV* *f*) (*GPV* *g*)

lemma *ord-gpv-coinduct* [*consumes 1, case-names ord-gpv, coinduct pred: ord-gpv*]:
assumes $X f g$
and step: $\bigwedge f g. X f g \implies \text{ord-spmf } (\text{rel-generat } (=) (=) (\text{rel-fun } (=) X)) (\text{the-gpv } f) (\text{the-gpv } g)$
shows *ord-gpv* *f g*
 $\langle \text{proof} \rangle$

lemma *ord-gpv-the-gpvD*:
 $\text{ord-gpv } f g \implies \text{ord-spmf } (\text{rel-generat } (=) (=) (\text{rel-fun } (=) \text{ord-gpv})) (\text{the-gpv } f) (\text{the-gpv } g)$
 $\langle \text{proof} \rangle$

lemma *reflp-equality*: *reflp* (=)
 $\langle \text{proof} \rangle$

lemma *ord-gpv-reflI* [*simp*]: *ord-gpv* *f f*
 $\langle \text{proof} \rangle$

lemma *reflp-ord-gpv*: *reflp* *ord-gpv*
 $\langle \text{proof} \rangle$

lemma *ord-gpv-trans*:
assumes *ord-gpv* *f g* *ord-gpv* *g h*
shows *ord-gpv* *f h*
 $\langle \text{proof} \rangle$

lemma *ord-gpv-compp*: (*ord-gpv* *OO* *ord-gpv*) = *ord-gpv*
 $\langle \text{proof} \rangle$

lemma *transp-ord-gpv* [*simp*]: *transp* *ord-gpv*
 $\langle \text{proof} \rangle$

lemma *ord-gpv-antisym*:
 $\llbracket \text{ord-gpv } f g; \text{ord-gpv } g f \rrbracket \implies f = g$
 $\langle \text{proof} \rangle$

lemma *RFail-least* [*simp*]: *ord-gpv* *Fail* *f*
 $\langle \text{proof} \rangle$

4.11 Bounds on interaction

context

fixes *consider* :: 'out \Rightarrow bool

notes *monotone-SUP*[*partial-function-mono*] [[*function-internals*]]

begin

$\langle \text{ML} \rangle$

partial-function (*lfp-strong*) *interaction-bound* :: ('a, 'out, 'in) gpv \Rightarrow enat

where

interaction-bound gpv =
 (SUP generat: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: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)))
]]
 \Rightarrow *P* (λ gpv. [] generat \in set-spmf (the-gpv gpv). case generat of Pure x \Rightarrow 0
 | IO out c \Rightarrow if consider out then eSuc ([] input. *interaction-bound'* (c
 input)) else ([] input. *interaction-bound'* (c input)))]]
 \Rightarrow *P interaction-bound*
 <proof>

lemma *interaction-bound-IO*:

IO out c \in set-spmf (the-gpv gpv)
 \Rightarrow (if consider out then eSuc (*interaction-bound* (c input)) else *interaction-bound*
 (c input)) \leq *interaction-bound gpv*
 <proof>

lemma *interaction-bound-IO-consider*:

[[*IO out c* \in set-spmf (the-gpv gpv); consider out]]
 \Rightarrow eSuc (*interaction-bound* (c input)) \leq *interaction-bound gpv*
 <proof>

lemma *interaction-bound-IO-ignore*:

[[*IO out c* \in set-spmf (the-gpv gpv); \neg consider out]]
 \Rightarrow *interaction-bound* (c input) \leq *interaction-bound gpv*
 <proof>

lemma *interaction-bound-Done* [*simp*]: *interaction-bound* (Done x) = 0

<proof>

lemma *interaction-bound-Fail* [*simp*]: *interaction-bound* Fail = 0

<proof>

lemma *interaction-bound-Pause* [*simp*]:

interaction-bound (Pause out c) =

(if consider out then eSuc (SUP input. interaction-bound (c input)) else (SUP input. interaction-bound (c input)))
 ⟨proof⟩

lemma *interaction-bound-lift-spmf* [simp]: interaction-bound (lift-spmf p) = 0
 ⟨proof⟩

lemma *interaction-bound-assert-gpv* [simp]: interaction-bound (assert-gpv b) = 0
 ⟨proof⟩

lemma *interaction-bound-bind-step*:

assumes IH: $\bigwedge p. \text{interaction-bound}' (p \ggg f) \leq \text{interaction-bound } p + (\bigsqcup x \in \text{results}'\text{-gpv } p. \text{interaction-bound}' (f x))$

and unfold: $\bigwedge \text{gpv}. \text{interaction-bound}' \text{ gpv} \leq (\bigsqcup \text{generat} \in \text{set-spmf } (\text{the-gpv } \text{gpv}). \text{case generat of Pure } x \Rightarrow 0$

| IO out c \Rightarrow if consider out then eSuc ($\bigsqcup \text{input}. \text{interaction-bound}' (c \text{ input})$) else $\bigsqcup \text{input}. \text{interaction-bound}' (c \text{ input})$)

shows ($\bigsqcup \text{generat} \in \text{set-spmf } (\text{the-gpv } (p \ggg f)). \text{case generat of Pure } x \Rightarrow 0$

| IO out c \Rightarrow

if consider out then eSuc ($\bigsqcup \text{input}. \text{interaction-bound}' (c \text{ input})$)
 else $\bigsqcup \text{input}. \text{interaction-bound}' (c \text{ input})$)

$\leq \text{interaction-bound } p +$

($\bigsqcup x \in \text{results}'\text{-gpv } p. \text{interaction-bound}' (f x)$).

$\bigsqcup \text{generat} \in \text{set-spmf } (\text{the-gpv } (f x)). \text{case generat of Pure } x \Rightarrow 0$

| IO out c \Rightarrow

if consider out then eSuc ($\bigsqcup \text{input}. \text{interaction-bound}' (c \text{ input})$)
 else $\bigsqcup \text{input}. \text{interaction-bound}' (c \text{ input})$)

(is (SUP generat':?bind. ?g generat') \leq ?p + ?f)

⟨proof⟩

lemma *interaction-bound-bind*:

defines $ib1 \equiv \text{interaction-bound}$

shows $\text{interaction-bound } (p \ggg f) \leq ib1 p + (\text{SUP } x:\text{results}'\text{-gpv } p. \text{interaction-bound } (f x))$

⟨proof⟩

lemma *interaction-bound-bind-lift-spmf* [simp]:

$\text{interaction-bound } (\text{lift-spmf } p \ggg f) = (\text{SUP } x:\text{set-spmf } p. \text{interaction-bound } (f x))$

⟨proof⟩

end

abbreviation *interaction-any-bound* :: ('a, 'out, 'in) gpv \Rightarrow enat

where $\text{interaction-any-bound} \equiv \text{interaction-bound } (\lambda_. \text{True})$

lemma *interaction-any-bound-coinduct* [consumes 1, case-names *interaction-bound*]:

assumes $X: X \text{ gpv } n$
and $*$: $\bigwedge \text{ gpv } n \text{ out } c \text{ input. } \llbracket X \text{ gpv } n; IO \text{ out } c \in \text{set-spmf } (\text{the-gpv } \text{ gpv}) \rrbracket$
 $\implies \exists n'. (X (c \text{ input}) n' \vee \text{interaction-any-bound } (c \text{ input}) \leq n') \wedge \text{eSuc } n' \leq n$
shows $\text{interaction-any-bound } \text{ gpv} \leq n$
 $\langle \text{proof} \rangle$

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

lemma *interaction-bound-parametric'*:

assumes $[\text{transfer-rule}]$: *bi-total* R

shows $((C \implies (=)) \implies \text{rel-gpv}'' A C \implies (=)) \text{interaction-bound}$
 interaction-bound
 $\langle \text{proof} \rangle$

lemma *interaction-bound-parametric* $[\text{transfer-rule}]$:

$((C \implies (=)) \implies \text{rel-gpv } A C \implies (=)) \text{interaction-bound interaction-bound}$
 $\langle \text{proof} \rangle$

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* $:: ('out \implies \text{bool}) \implies ('a, 'out, 'in) \text{ gpv} \implies \text{enat} \implies \text{bool}$

for *consider gpv n where*

interaction-bounded-by: $\llbracket \text{interaction-bound consider gpv} \leq n \rrbracket \implies \text{interaction-bounded-by consider gpv } n$

lemmas *interaction-bounded-byI* = *interaction-bounded-by*

hide-fact (**open**) *interaction-bounded-by*

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

lemma *interaction-bounded-by-parametric* $[\text{transfer-rule}]$:

$((C \implies (=)) \implies \text{rel-gpv } A C \implies (=) \implies (=)) \text{interaction-bounded-by}$
 $\text{interaction-bounded-by}$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-parametric'*:

notes *interaction-bound-parametric'* $[\text{transfer-rule}]$

assumes $[\text{transfer-rule}]$: *bi-total* R

shows $((C \implies (=)) \implies \text{rel-gpv}'' A C \implies (=) \implies (=))$
 $\text{interaction-bounded-by interaction-bounded-by}$

$\langle \text{proof} \rangle$

end

lemma *interaction-bounded-by-mono*:

$\llbracket \text{interaction-bounded-by consider gpv } n; n \leq m \rrbracket \implies \text{interaction-bounded-by consider gpv } m$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-contD*:

[[*interaction-bounded-by consider gpv n; IO out c ∈ set-spmf (the-gpv gpv); consider out*]]
⇒ $n > 0 \wedge \text{interaction-bounded-by consider } (c \text{ input}) (n - 1)$
<proof>

lemma *interaction-bounded-by-contD-ignore*:

[[*interaction-bounded-by consider gpv n; IO out c ∈ set-spmf (the-gpv gpv)*]]
⇒ *interaction-bounded-by consider (c input) n*
<proof>

lemma *interaction-bounded-byI-epred*:

assumes $\bigwedge \text{out } c. \llbracket \text{IO out } c \in \text{set-spmf (the-gpv gpv); consider out} \rrbracket \Rightarrow n \neq 0 \wedge (\forall \text{input. interaction-bounded-by consider } (c \text{ input}) (n - 1))$
and $\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf (the-gpv gpv); } \neg \text{consider out} \rrbracket \Rightarrow \text{interaction-bounded-by consider } (c \text{ input}) n$
shows *interaction-bounded-by consider gpv n*
<proof>

lemma *interaction-bounded-by-IO*:

[[*IO out c ∈ set-spmf (the-gpv gpv); interaction-bounded-by consider gpv n; consider out*]]
⇒ $n \neq 0 \wedge \text{interaction-bounded-by consider } (c \text{ input}) (n - 1)$
<proof>

lemma *interaction-bounded-by-0*: *interaction-bounded-by consider gpv 0* \longleftrightarrow *interaction-bound consider gpv = 0*
<proof>

abbreviation *interaction-bounded-by'* :: $(\text{'out} \Rightarrow \text{bool}) \Rightarrow (\text{'a, 'out, 'in}) \text{ gpv} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where *interaction-bounded-by'* *consider gpv n* \equiv *interaction-bounded-by consider gpv (enat n)*

named-theorems *interaction-bound*

lemmas *interaction-bounded-by-start = interaction-bounded-by-mono*

method *interaction-bound-start = (rule interaction-bounded-by-start)*

method *interaction-bound-step uses add simp =*

((match conclusion in interaction-bounded-by - - - => fail | - => (solves <clarsimp simp add: simp>>)) | rule add interaction-bound)

method *interaction-bound-rec uses add simp =*

(interaction-bound-step add: add simp: simp; (interaction-bound-rec add: add simp: simp)?)

method *interaction-bound uses add simp =*

(interaction-bound-start, interaction-bound-rec add: add simp: simp)

lemma *interaction-bounded-by-Done* [simp]: *interaction-bounded-by consider (Done x) n*
 ⟨proof⟩

lemma *interaction-bounded-by-DoneI* [interaction-bound]:
interaction-bounded-by consider (Done x) 0
 ⟨proof⟩

lemma *interaction-bounded-by-Fail* [simp]: *interaction-bounded-by consider Fail n*
 ⟨proof⟩

lemma *interaction-bounded-by-FailI* [interaction-bound]: *interaction-bounded-by consider Fail 0*
 ⟨proof⟩

lemma *interaction-bounded-by-lift-spmf* [simp]: *interaction-bounded-by consider (lift-spmf p) n*
 ⟨proof⟩

lemma *interaction-bounded-by-lift-spmfI* [interaction-bound]:
interaction-bounded-by consider (lift-spmf p) 0
 ⟨proof⟩

lemma *interaction-bounded-by-assert-gpv* [simp]: *interaction-bounded-by consider (assert-gpv b) n*
 ⟨proof⟩

lemma *interaction-bounded-by-assert-gpvI* [interaction-bound]:
interaction-bounded-by consider (assert-gpv b) 0
 ⟨proof⟩

lemma *interaction-bounded-by-Pause* [simp]:
interaction-bounded-by consider (Pause out c) n \longleftrightarrow
(if consider out then $0 < n \wedge (\forall \text{input. interaction-bounded-by consider (c input) (n - 1))$ else $(\forall \text{input. interaction-bounded-by consider (c input) n)$)
 ⟨proof⟩

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 $1 + (SUP \text{input. } n \text{ input})$ else $(SUP \text{input. } n \text{ input})$)*
 ⟨proof⟩

lemma *interaction-bounded-by-bindI* [interaction-bound]:
 $\llbracket \text{interaction-bounded-by consider } gpv \ n; \bigwedge x. x \in \text{results}'\text{-gpv } gpv \implies \text{interaction-bounded-by consider (f x) (m x)} \rrbracket$
 \implies *interaction-bounded-by consider (gpv \ggg f) (n + (SUP x:results'-gpv gpv. m x))*
 ⟨proof⟩

lemma *interaction-bounded-by-bind-PauseI* [*interaction-bound*]:

$(\bigwedge \text{input}. \text{interaction-bounded-by consider } (c \text{ input} \ggg f) (n \text{ input}))$
 $\implies \text{interaction-bounded-by consider } (\text{Pause out } c \ggg f)$ (if consider out then
SUP *input. n input + 1* else *SUP input. n input*)
 ⟨*proof*⟩

lemma *interaction-bounded-by-bind-lift-spmf* [*simp*]:

interaction-bounded-by consider (*lift-spmf p* \ggg *f*) *n* \longleftrightarrow $(\forall x \in \text{set-spmf } p. \text{interaction-bounded-by consider } (f x) n)$
 ⟨*proof*⟩

lemma *interaction-bounded-by-bind-lift-spmfI* [*interaction-bound*]:

$(\bigwedge x. x \in \text{set-spmf } p \implies \text{interaction-bounded-by consider } (f x) (n x))$
 $\implies \text{interaction-bounded-by consider } (\text{lift-spmf } p \ggg f)$ (*SUP x:set-spmf p. n x*)
 ⟨*proof*⟩

lemma *interaction-bounded-by-bind-DoneI* [*interaction-bound*]:

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*]:

$\llbracket b \implies \text{interaction-bounded-by consider } \text{gpv1 } n; \neg b \implies \text{interaction-bounded-by consider } \text{gpv2 } m \rrbracket$
 $\implies \text{interaction-bounded-by consider } (\text{if } b \text{ then } \text{gpv1} \text{ else } \text{gpv2})$ (if *b* then *n* else *m*)
 ⟨*proof*⟩

lemma *interaction-bounded-by-case-bool* [*interaction-bound*]:

$\llbracket b \implies \text{interaction-bounded-by consider } t \text{ bt}; \neg b \implies \text{interaction-bounded-by consider } f \text{ bf} \rrbracket$
 $\implies \text{interaction-bounded-by consider } (\text{case-bool } t \text{ } f \text{ } b)$ (if *b* then *bt* else *bf*)
 ⟨*proof*⟩

lemma *interaction-bounded-by-case-sum* [*interaction-bound*]:

$\llbracket \bigwedge y. x = \text{Inl } y \implies \text{interaction-bounded-by consider } (l y) (bl y);$
 $\bigwedge y. x = \text{Inr } y \implies \text{interaction-bounded-by consider } (r y) (br y) \rrbracket$
 $\implies \text{interaction-bounded-by consider } (\text{case-sum } l \text{ } r \text{ } x)$ (*case-sum bl br x*)
 ⟨*proof*⟩

lemma *interaction-bounded-by-case-prod* [*interaction-bound*]:

$(\bigwedge a \text{ } b. x = (a, b) \implies \text{interaction-bounded-by consider } (f a \text{ } b) (n a \text{ } b))$
 $\implies \text{interaction-bounded-by consider } (\text{case-prod } f \text{ } x)$ (*case-prod n x*)
 ⟨*proof*⟩

lemma *interaction-bounded-by-let* [*interaction-bound*]: — This rule unfolds let's

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*]: *interaction-bounded-by* P *gpv* n
 shows *interaction-bounded-by* P (*map-gpv* f *id* *gpv*) n
<proof>

abbreviation *interaction-any-bounded-by* :: ('*a*, '*out*, '*in*) *gpv* \Rightarrow *enat* \Rightarrow *bool*
where *interaction-any-bounded-by* \equiv *interaction-bounded-by* (λ -. *True*)

4.12 Typing

4.12.1 Interface between gpvs and rpvs / callees

lemma *is-empty-parametric* [*transfer-rule*]: *rel-fun* (*rel-set* A) (=) *Set.is-empty*
Set.is-empty
<proof>

typedef ('*call*, '*ret*) $\mathcal{I} = UNIV$:: ('*call* \Rightarrow '*ret set*) *set* *<proof>*

setup-lifting *type-definition- \mathcal{I}*

lemma *outs- \mathcal{I} -tparametric*:
 includes *lifting-syntax*
 assumes [*transfer-rule*]: *bi-total* A
 shows (($A \Longrightarrow rel\text{-set } B \Longrightarrow rel\text{-set } A$) (λ *resps*. {*out*. *resps out* \neq {}}})
 (λ *resps*. {*out*. *resps out* \neq {}}})
 <proof>

lift-definition *outs- \mathcal{I}* :: ('*call*, '*ret*) $\mathcal{I} \Rightarrow$ '*call set is* λ *resps*. {*out*. *resps out* \neq {}}}
parametric *outs- \mathcal{I} -tparametric* *<proof>*

lift-definition *responses- \mathcal{I}* :: ('*call*, '*ret*) $\mathcal{I} \Rightarrow$ '*call* \Rightarrow '*ret set is* λx . x **parametric**
id-transfer[*unfolded id-def*] *<proof>*

lift-definition *rel- \mathcal{I}* :: ('*call* \Rightarrow '*call'* \Rightarrow *bool*) \Rightarrow ('*ret* \Rightarrow '*ret'* \Rightarrow *bool*) \Rightarrow ('*call*,
'*ret*) $\mathcal{I} \Rightarrow$ ('*call'*, '*ret'*) $\mathcal{I} \Rightarrow$ *bool*
is $\lambda C R resp1 resp2$. *rel-set* C {*out*. *resp1 out* \neq {}} {*out*. *resp2 out* \neq {}} \wedge
rel-fun C (*rel-set* R) *resp1 resp2*
<proof>

lemma *rel- \mathcal{I}* [*intro?*]:
 [[*rel-set* C (*outs- \mathcal{I}* $\mathcal{I}1$) (*outs- \mathcal{I}* $\mathcal{I}2$); $\bigwedge x y$. $C x y \Longrightarrow rel\text{-set } R$ (*responses- \mathcal{I}* $\mathcal{I}1$
 x) (*responses- \mathcal{I}* $\mathcal{I}2$ y)]]
 $\Longrightarrow rel\text{- $\mathcal{I}$$ $C R \mathcal{I}1 \mathcal{I}2$
<proof>

lemma *rel- \mathcal{I} -eq* [*relator-eq*]: *rel- \mathcal{I}* (=) (=) (=)
<proof>

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

<proof>

lemma *rel-I-conversep1-eq* [*simp*]: $\text{rel-}\mathcal{I} \ C^{-1-1} \ (=) \ = \ (\text{rel-}\mathcal{I} \ C \ (=))^{-1-1}$
<proof>

lemma *rel-I-conversep2-eq* [*simp*]: $\text{rel-}\mathcal{I} \ (=) \ R^{-1-1} \ = \ (\text{rel-}\mathcal{I} \ (=) \ R)^{-1-1}$
<proof>

lemma *responses-I-empty-iff*: $\text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out} \ = \ \{\}$ \longleftrightarrow $\text{out} \notin \text{outs-}\mathcal{I} \ \mathcal{I}$
including *I.lifting* *<proof>*

lemma *in-outs-I-iff-responses-I*: $\text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \ \longleftrightarrow \ \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out} \neq \{\}$
<proof>

lift-definition *I-full* :: $(\text{'call}, \text{'ret}) \ \mathcal{I} \ \text{is} \ \lambda\text{-} \text{UNIV}$ *<proof>*

lemma *I-full-sel* [*simp*]:
 shows *outs-I-full*: $\text{outs-}\mathcal{I} \ \mathcal{I}\text{-full} \ = \ \text{UNIV}$
 and *responses-I-full*: $\text{responses-}\mathcal{I} \ \mathcal{I}\text{-full} \ x \ = \ \text{UNIV}$
<proof>

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

lemma *outs-I-parametric* [*transfer-rule*]: $(\text{rel-}\mathcal{I} \ C \ R \ ==\Rightarrow \ \text{rel-set} \ C) \ \text{outs-}\mathcal{I}$
outs-I
<proof>

lemma *responses-I-parametric* [*transfer-rule*]:
 $(\text{rel-}\mathcal{I} \ C \ R \ ==\Rightarrow \ C \ ==\Rightarrow \ \text{rel-set} \ R) \ \text{responses-}\mathcal{I} \ \text{responses-}\mathcal{I}$
<proof>

end

definition *I-trivial* :: $(\text{'out}, \text{'in}) \ \mathcal{I} \ \Rightarrow \ \text{bool}$
where *I-trivial* $\mathcal{I} \ \longleftrightarrow \ \text{outs-}\mathcal{I} \ \mathcal{I} \ = \ \text{UNIV}$

lemma *I-trivialI* [*intro?*]: $(\bigwedge x. x \in \text{outs-}\mathcal{I} \ \mathcal{I}) \ \Longrightarrow \ \mathcal{I}\text{-trivial} \ \mathcal{I}$
<proof>

lemma *I-trivialD*: $\mathcal{I}\text{-trivial} \ \mathcal{I} \ \Longrightarrow \ \text{outs-}\mathcal{I} \ \mathcal{I} \ = \ \text{UNIV}$
<proof>

lemma *I-trivial-I-full* [*simp*]: $\mathcal{I}\text{-trivial} \ \mathcal{I}\text{-full}$
<proof>

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

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 $results\text{-}gpv\ \Gamma\ gpv \equiv \{x. resultsp\text{-}gpv\ \Gamma\ x\ gpv\}$

lemma $resultsp\text{-}gpv\text{-}results\text{-}gpv\text{-}eq\ [pred\text{-}set\text{-}conv]: resultsp\text{-}gpv\ \Gamma\ x\ gpv \longleftrightarrow x \in results\text{-}gpv\ \Gamma\ gpv$

$\langle proof \rangle$

context begin

$\langle ML \rangle$

lemmas $intros\ [intro?]\ = resultsp\text{-}gpv.intros[to\text{-}set]$

and $Pure = Pure[to\text{-}set]$

and $IO = IO[to\text{-}set]$

and $induct\ [consumes\ 1, case\text{-}names\ Pure\ IO, induct\ set: results\text{-}gpv] = resultsp\text{-}gpv.induct[to\text{-}set]$

and $cases\ [consumes\ 1, case\text{-}names\ Pure\ IO, cases\ set: results\text{-}gpv] = resultsp\text{-}gpv.cases[to\text{-}set]$

and $simps = resultsp\text{-}gpv.simps[to\text{-}set]$

end

inductive-simps $results\text{-}gpv\text{-}GPV\ [to\text{-}set, simp]: resultsp\text{-}gpv\ \Gamma\ x\ (GPV\ gpv)$

end

lemma $results\text{-}gpv\text{-}Done\ [iff]: results\text{-}gpv\ \Gamma\ (Done\ x) = \{x\}$

$\langle proof \rangle$

lemma $results\text{-}gpv\text{-}Fail\ [iff]: results\text{-}gpv\ \Gamma\ Fail = \{\}$

$\langle proof \rangle$

lemma $results\text{-}gpv\text{-}Pause\ [simp]:$

$results\text{-}gpv\ \Gamma\ (Pause\ out\ c) = (\bigcup input \in responses\text{-}\mathcal{I}\ \Gamma\ out. results\text{-}gpv\ \Gamma\ (c\ input))$

$\langle proof \rangle$

lemma $results\text{-}gpv\text{-}lift\text{-}spmf\ [iff]: results\text{-}gpv\ \Gamma\ (lift\text{-}spmf\ p) = set\text{-}spmf\ p$

$\langle proof \rangle$

lemma $results\text{-}gpv\text{-}assert\text{-}gpv\ [simp]: results\text{-}gpv\ \Gamma\ (assert\text{-}gpv\ b) = (if\ b\ then\ \{\}\ else\ \{\})$

$\langle proof \rangle$

lemma *results-gpv-bind-gpv* [simp]:

$$\text{results-gpv } \Gamma (gpv \gg f) = (\bigcup x \in \text{results-gpv } \Gamma \text{ gpv. results-gpv } \Gamma (f x))$$

(is ?lhs = ?rhs)

<proof>

lemma *results-gpv-I-full*: *results-gpv I-full = results'-gpv*

<proof>

lemma *results'-bind-gpv* [simp]:

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

<proof>

lemma *results-gpv-map-gpv-id* [simp]: *results-gpv I (map-gpv f id gpv) = f ' results-gpv I gpv*

<proof>

lemma *results-gpv-map-gpv-id'* [simp]: *results-gpv I (map-gpv f (\lambda x. x) gpv) = f ' results-gpv 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]:

$$\text{results'-gpv } (\text{monad.bind-option Fail } x f) = (\bigcup y \in \text{set-option } x. \text{results'-gpv } (f y))$$

<proof>

lemma *bind-gpv-bind-option-assoc*:

$$\text{bind-gpv } (\text{monad.bind-option Fail } x f) g = \text{monad.bind-option Fail } x (\lambda x. \text{bind-gpv } (f x) g)$$

<proof>

4.12.2 Type judgements

coinductive *WT-gpv* :: ('out, 'in) $\mathcal{I} \Rightarrow$ ('a, 'out, 'in) *gpv* \Rightarrow bool (((-)/ \vdash g (-) \checkmark) [100, 0] 99)

for Γ

where

$$(\bigwedge \text{out } c. \text{IO out } c \in \text{set-spmf } gpv \implies \text{out} \in \text{outs-}\mathcal{I} \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \Gamma \text{ out. } \Gamma \vdash g c \text{ input } \checkmark))$$

$$\implies \Gamma \vdash g \text{ GPV } gpv \checkmark$$

lemma *WT-gpv-coinduct* [consumes 1, case-names *WT-gpv*, case-conclusion *WT-gpv out cont, coinduct pred: WT-gpv*]:

assumes *: *X gpv*

and *step*: $\bigwedge gpv \text{ out } c.$

$$\llbracket X gpv; \text{IO out } c \in \text{set-spmf } (\text{the-gpv } gpv) \rrbracket$$

$$\implies \text{out} \in \text{outs-}\mathcal{I} \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \Gamma \text{ out. } X (c \text{ input}) \vee \Gamma \vdash g c \text{ input}$$

\checkmark)
shows $\Gamma \vdash_g \text{gpv} \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-gpv-simps*:
 $\Gamma \vdash_g \text{GPV} \text{gpv} \checkmark \iff$
 $(\forall \text{out } c. \text{IO out } c \in \text{set-spmf } \text{gpv} \implies \text{out} \in \text{outs-}\mathcal{I} \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \Gamma$
 $\text{out}. \Gamma \vdash_g c \text{ input } \checkmark))$
 $\langle \text{proof} \rangle$

lemma *WT-gpvI*:
 $(\wedge \text{out } c. \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \implies \text{out} \in \text{outs-}\mathcal{I} \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I}$
 $\Gamma \text{ out}. \Gamma \vdash_g c \text{ input } \checkmark))$
 $\implies \Gamma \vdash_g \text{gpv} \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-gpvD*:
assumes $\Gamma \vdash_g \text{gpv} \checkmark$
shows *WT-gpv-OutD*: $\text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \implies \text{out} \in \text{outs-}\mathcal{I} \Gamma$
and *WT-gpv-ContD*: $\llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I}$
 $\Gamma \text{ out} \rrbracket \implies \Gamma \vdash_g c \text{ input } \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-gpv-mono*:
assumes *WT*: $\mathcal{I}1 \vdash_g \text{gpv} \checkmark$
and *outs*: $\text{outs-}\mathcal{I} \mathcal{I}1 \subseteq \text{outs-}\mathcal{I} \mathcal{I}2$
and *responses*: $\bigwedge x. x \in \text{outs-}\mathcal{I} \mathcal{I}1 \implies \text{responses-}\mathcal{I} \mathcal{I}2 x \subseteq \text{responses-}\mathcal{I} \mathcal{I}1 x$
shows $\mathcal{I}2 \vdash_g \text{gpv} \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-gpv-Done* [*iff*]: $\Gamma \vdash_g \text{Done } x \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-gpv-Fail* [*iff*]: $\Gamma \vdash_g \text{Fail} \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-gpv-PauseI*:
 $\llbracket \text{out} \in \text{outs-}\mathcal{I} \Gamma; \bigwedge \text{input}. \text{input} \in \text{responses-}\mathcal{I} \Gamma \text{ out} \implies \Gamma \vdash_g c \text{ input } \checkmark \rrbracket$
 $\implies \Gamma \vdash_g \text{Pause out } c \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-gpv-Pause* [*iff*]:
 $\Gamma \vdash_g \text{Pause out } c \checkmark \iff \text{out} \in \text{outs-}\mathcal{I} \Gamma \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \Gamma \text{ out}. \Gamma \vdash_g c$
 $\text{input } \checkmark)$
 $\langle \text{proof} \rangle$

lemma *WT-gpv-bindI*:
 $\llbracket \Gamma \vdash_g \text{gpv} \checkmark; \bigwedge x. x \in \text{results-gpv } \Gamma \text{gpv} \implies \Gamma \vdash_g f x \checkmark \rrbracket$
 $\implies \Gamma \vdash_g \text{gpv} \ggg f \checkmark$

<proof>

lemma *WT-gpv-bindD2*:

assumes *WT*: $\Gamma \vdash_g \text{gpv} \ggg f \checkmark$

and *x*: $x \in \text{results-gpv } \Gamma \text{ gpv}$

shows $\Gamma \vdash_g f x \checkmark$

<proof>

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*: $\bigwedge \text{gpv out } c. \llbracket X \text{ gpv}; \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}) \rrbracket$

$\implies \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \wedge (\forall \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out.}$

$X (c \text{ input}) \vee$

$\mathcal{I} \vdash_g c \text{ input} \checkmark \vee$

$(\exists (\text{gpv}' :: ('b, 'call, 'ret) \text{gpv}) f. c \text{ input} = \text{gpv}' \ggg f \wedge \mathcal{I} \vdash_g \text{gpv}' \checkmark \wedge$

$(\forall x \in \text{results-gpv } \mathcal{I} \ \text{gpv}'. X (f x))))$

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>

4.13 Sub-gpvs

context *begin*

qualified inductive *sub-gpvsp* $:: ('out, 'in) \mathcal{I} \Rightarrow ('a, 'out, 'in) \text{gpv} \Rightarrow ('a, 'out, 'in) \text{gpv} \Rightarrow \text{bool}$

for $\mathcal{I} \ x$

where

base:

$$\llbracket IO\ out\ c \in\ set\text{-}spm\!f\ (the\text{-}gpv\ gpv);\ input \in\ responses\text{-}\mathcal{I}\ \mathcal{I}\ out; x = c\ input \rrbracket$$

$$\implies\ sub\text{-}gpvsp\ \mathcal{I}\ x\ gpv$$
| *cont*:
$$\llbracket IO\ out\ c \in\ set\text{-}spm\!f\ (the\text{-}gpv\ gpv); input \in\ responses\text{-}\mathcal{I}\ \mathcal{I}\ out; sub\text{-}gpvsp\ \mathcal{I}\ x$$

$$(c\ input) \rrbracket$$

$$\implies\ sub\text{-}gpvsp\ \mathcal{I}\ x\ gpv$$

qualified lemma *sub-gpvsp-base*:

$$\llbracket IO\ out\ c \in\ set\text{-}spm\!f\ (the\text{-}gpv\ gpv); input \in\ responses\text{-}\mathcal{I}\ \mathcal{I}\ out \rrbracket$$

$$\implies\ sub\text{-}gpvsp\ \mathcal{I}\ (c\ input)\ gpv$$
<proof>

definition *sub-gpvs* :: (*'out*, *'in*) $\mathcal{I} \Rightarrow$ (*'a*, *'out*, *'in*) *gpv* \Rightarrow (*'a*, *'out*, *'in*) *gpv set*
where *sub-gpvs* $\mathcal{I}\ gpv \equiv \{x.\ sub\text{-}gpvsp\ \mathcal{I}\ x\ gpv\}$

lemma *sub-gpvsp-sub-gpvs-eq* [*pred-set-conv*]: *sub-gpvsp* $\mathcal{I}\ x\ gpv \longleftrightarrow x \in\ sub\text{-}gpvs$
 $\mathcal{I}\ gpv$
<proof>

context begin

<ML>

lemmas *intros* [*intro?*] = *sub-gpvsp.intros*[*to-set*]
and *base* = *sub-gpvsp-base*[*to-set*]
and *cont* = *cont*[*to-set*]
and *induct* [*consumes 1*, *case-names Pure IO*, *induct set: sub-gpvs*] = *sub-gpvsp.induct*[*to-set*]
and *cases* [*consumes 1*, *case-names Pure IO*, *cases set: sub-gpvs*] = *sub-gpvsp.cases*[*to-set*]
and *simps* = *sub-gpvsp.simps*[*to-set*]
end
end

lemma *WT-sub-gpvsD*:

assumes $\mathcal{I} \vdash_g\ gpv\ \checkmark$ **and** $gpv' \in\ sub\text{-}gpvs\ \mathcal{I}\ gpv$
shows $\mathcal{I} \vdash_g\ gpv'\ \checkmark$
<proof>

lemma *WT-sub-gpvsI*:

$$\llbracket \bigwedge\ out\ c.\ IO\ out\ c \in\ set\text{-}spm\!f\ (the\text{-}gpv\ gpv) \implies out \in\ outs\text{-}\mathcal{I}\ \Gamma;$$

$$\bigwedge\ gpv'.\ gpv' \in\ sub\text{-}gpvs\ \Gamma\ gpv \implies \Gamma \vdash_g\ gpv'\ \checkmark \rrbracket$$

$$\implies\ \Gamma \vdash_g\ gpv\ \checkmark$$
<proof>

4.14 Losslessness

A *gpv* is *lossless* iff we are guaranteed to get a result after a finite number of interactions that respect the interface. It is *colossless* if the interactions may go on for ever, but there is no non-termination.

We define both notions of losslessness simultaneously by mimicking what the

(co)inductive package would do internally. Thus, we get a constant which is parametrised by the choice of the fixpoint, i.e., for non-recursive gpv's, we can state and prove both versions of losslessness in one go.

context

fixes $co :: \text{bool}$ **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 $co' :: \text{bool}$
defines $F \equiv \lambda \text{gen-lossless-gpv } \text{gpv}. \exists pa. \text{gpv} = \text{GPV } pa \wedge$
 $\text{lossless-spmf } pa \wedge (\forall \text{out } c \text{ input}. \text{IO out } c \in \text{set-spmf } pa \longrightarrow \text{input} \in \text{responses-}\mathcal{I}$
 $\mathcal{I} \text{ out} \longrightarrow \text{gen-lossless-gpv } (c \text{ input}))$
and $co' \equiv co$ — We use a copy of co such that we can do case distinctions on co' without the simplifier rewriting the co in the local abbreviations for the constants.
begin

lemma *gen-lossless-gpv-mono*: $\text{mono } F$
 $\langle \text{proof} \rangle$

definition *gen-lossless-gpv* $:: ('a, 'out, 'in) \text{gpv} \Rightarrow \text{bool}$
where $\text{gen-lossless-gpv} = (\text{if } co' \text{ then } \text{gfp} \text{ else } \text{lfp}) F$

lemma *gen-lossless-gpv-unfold*: $\text{gen-lossless-gpv} = F \text{ gen-lossless-gpv}$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpv-True*: $co' = \text{True} \Longrightarrow \text{gen-lossless-gpv} \equiv \text{gfp } F$
and *gen-lossless-gpv-False*: $co' = \text{False} \Longrightarrow \text{gen-lossless-gpv} \equiv \text{lfp } F$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpv-cases* [*elim?*, *cases pred*]:
assumes $\text{gen-lossless-gpv } \text{gpv}$
obtains $(\text{gen-lossless-gpv}) p$ **where** $\text{gpv} = \text{GPV } p \text{ lossless-spmf } p$
 $\bigwedge \text{out } c \text{ input}. \llbracket \text{IO out } c \in \text{set-spmf } p; \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out} \rrbracket \Longrightarrow \text{gen-lossless-gpv}$
 $(c \text{ input})$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpvD*:
assumes $\text{gen-lossless-gpv } \text{gpv}$
shows $\text{gen-lossless-gpv-lossless-spmfD}$: $\text{lossless-spmf } (\text{the-gpv } \text{gpv})$
and $\text{gen-lossless-gpv-continuationD}$:
 $\llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out} \rrbracket \Longrightarrow \text{gen-lossless-gpv}$
 $(c \text{ input})$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpv-intros*:
 $\llbracket \text{lossless-spmf } p; \bigwedge \text{out } c \text{ input}. \llbracket \text{IO out } c \in \text{set-spmf } p; \text{input} \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out} \rrbracket \rrbracket \Longrightarrow$
 $\text{gen-lossless-gpv } (c \text{ input}) \rrbracket$
 $\Longrightarrow \text{gen-lossless-gpv } (\text{GPV } p)$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpvI* [intro?]:

$$\llbracket \text{lossless-spmf } (\text{the-gpv } \text{gpv});$$

$$\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out}$$

$$\rrbracket$$

$$\implies \text{gen-lossless-gpv } (c \text{ input}) \rrbracket$$

$$\implies \text{gen-lossless-gpv } \text{gpv}$$

<proof>

lemma *gen-lossless-gpv-simps*:

$$\text{gen-lossless-gpv } \text{gpv} \longleftrightarrow$$

$$(\exists p. \text{gpv} = \text{GPV } p \wedge \text{lossless-spmf } p \wedge (\forall \text{out } c \text{ input.}$$

$$\text{IO out } c \in \text{set-spmf } p \longrightarrow \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out} \longrightarrow \text{gen-lossless-gpv}$$

$$(c \text{ input})))$$

<proof>

lemma *gen-lossless-gpv-Done* [iff]: *gen-lossless-gpv* (Done x)
<proof>

lemma *gen-lossless-gpv-Fail* [iff]: \neg *gen-lossless-gpv* Fail
<proof>

lemma *gen-lossless-gpv-Pause* [simp]:

$$\text{gen-lossless-gpv } (\text{Pause out } c) \longleftrightarrow (\forall \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out. } \text{gen-lossless-gpv}$$

$$(c \text{ input}))$$

<proof>

lemma *gen-lossless-gpv-lift-spmf* [iff]: *gen-lossless-gpv* (lift-spmf p) \longleftrightarrow *lossless-spmf*
 p
<proof>

end

lemma *gen-lossless-gpv-assert-gpv* [iff]: *gen-lossless-gpv* co \mathcal{I} (assert-gpv b) \longleftrightarrow b
<proof>

abbreviation *lossless-gpv* :: ('out, 'in) $\mathcal{I} \Rightarrow$ ('a, 'out, 'in) *gpv* \Rightarrow *bool*
where *lossless-gpv* \equiv *gen-lossless-gpv* *False*

abbreviation *colossless-gpv* :: ('out, 'in) $\mathcal{I} \Rightarrow$ ('a, 'out, 'in) *gpv* \Rightarrow *bool*
where *colossless-gpv* \equiv *gen-lossless-gpv* *True*

lemma *lossless-gpv-induct* [consumes 1, case-names *lossless-gpv*, induct *pred*]:
assumes *: *lossless-gpv* \mathcal{I} *gpv*
and *step*: $\bigwedge p. \llbracket \text{lossless-spmf } p;$

$$\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } p; \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out} \rrbracket \implies$$

$$\text{lossless-gpv } \mathcal{I} (c \text{ input});$$

$$\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } p; \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out} \rrbracket \implies P (c$$

$$\text{input}) \rrbracket$$

$$\implies P (\text{GPV } p)$$

shows $P \text{ gpv}$
 $\langle \text{proof} \rangle$

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*: $\bigwedge \text{ gpv}. X \text{ gpv} \implies \text{lossless-spmf } (\text{the-gpv gpv}) \wedge (\forall \text{ out } c \text{ input}.$
 $\text{IO out } c \in \text{set-spmf } (\text{the-gpv gpv}) \longrightarrow \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} \longrightarrow X (c$
 $\text{input}) \vee \text{colossless-gpv } \mathcal{I} (c \text{ input}))$
shows *colossless-gpv* $\mathcal{I} \text{ gpv}$
 $\langle \text{proof} \rangle$

lemmas $\text{lossless-gpvI} = \text{gen-lossless-gpvI}[\mathbf{where} \text{ } co=False]$
and $\text{lossless-gpvD} = \text{gen-lossless-gpvD}[\mathbf{where} \text{ } co=False]$
and $\text{lossless-gpv-lossless-spmfD} = \text{gen-lossless-gpv-lossless-spmfD}[\mathbf{where} \text{ } co=False]$
and $\text{lossless-gpv-continuationD} = \text{gen-lossless-gpv-continuationD}[\mathbf{where} \text{ } co=False]$

lemmas $\text{colossless-gpvI} = \text{gen-lossless-gpvI}[\mathbf{where} \text{ } co=True]$
and $\text{colossless-gpvD} = \text{gen-lossless-gpvD}[\mathbf{where} \text{ } co=True]$
and $\text{colossless-gpv-lossless-spmfD} = \text{gen-lossless-gpv-lossless-spmfD}[\mathbf{where} \text{ } co=True]$
and $\text{colossless-gpv-continuationD} = \text{gen-lossless-gpv-continuationD}[\mathbf{where} \text{ } co=True]$

lemma *gen-lossless-bind-gpvI*:
assumes $\text{gen-lossless-gpv } co \ \mathcal{I} \ \text{ gpv} \wedge x. x \in \text{results-gpv } \mathcal{I} \ \text{ gpv} \implies \text{gen-lossless-gpv}$
 $co \ \mathcal{I} (f \ x)$
shows $\text{gen-lossless-gpv } co \ \mathcal{I} (\text{ gpv} \ggg f)$
 $\langle \text{proof} \rangle$

lemmas $\text{lossless-bind-gpvI} = \text{gen-lossless-bind-gpvI}[\mathbf{where} \text{ } co=False]$
and $\text{colossless-bind-gpvI} = \text{gen-lossless-bind-gpvI}[\mathbf{where} \text{ } co=True]$

lemma *gen-lossless-bind-gpvD1*:
assumes $\text{gen-lossless-gpv } co \ \mathcal{I} (\text{ gpv} \ggg f)$
shows $\text{gen-lossless-gpv } co \ \mathcal{I} \ \text{ gpv}$
 $\langle \text{proof} \rangle$

lemmas $\text{lossless-bind-gpvD1} = \text{gen-lossless-bind-gpvD1}[\mathbf{where} \text{ } co=False]$
and $\text{colossless-bind-gpvD1} = \text{gen-lossless-bind-gpvD1}[\mathbf{where} \text{ } co=True]$

lemma *gen-lossless-bind-gpvD2*:
assumes $\text{gen-lossless-gpv } co \ \mathcal{I} (\text{ gpv} \ggg f)$
and $x \in \text{results-gpv } \mathcal{I} \ \text{ gpv}$
shows $\text{gen-lossless-gpv } co \ \mathcal{I} (f \ x)$
 $\langle \text{proof} \rangle$

lemmas $\text{lossless-bind-gpvD2} = \text{gen-lossless-bind-gpvD2}[\mathbf{where} \text{ } co=False]$
and $\text{colossless-bind-gpvD2} = \text{gen-lossless-bind-gpvD2}[\mathbf{where} \text{ } co=True]$

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 $lossless-bind-gpv = gen-lossless-bind-gpv[\mathbf{where}\ co=False]$
 $\mathbf{and}\ colossless-bind-gpv = gen-lossless-bind-gpv[\mathbf{where}\ co=True]$

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*:
 $\llbracket rel-gpv''\ A\ C\ R\ gpv\ gpv';\ lossless-gpv\ \mathcal{I}'\ gpv';\ rel-\mathcal{I}\ C\ R\ \mathcal{I}\ \mathcal{I}' \rrbracket$
 $\implies lossless-gpv\ \mathcal{I}\ gpv$
 $\langle proof \rangle$

lemma *rel-gpv-lossless-gpvD1*:
 $\llbracket rel-gpv\ A\ C\ gpv\ gpv';\ lossless-gpv\ \mathcal{I}\ gpv;\ rel-\mathcal{I}\ C\ (=)\ \mathcal{I}\ \mathcal{I}' \rrbracket \implies lossless-gpv\ \mathcal{I}'\ gpv'$
 $\langle proof \rangle$

lemma *rel-gpv-lossless-gpvD2*:
 $\llbracket rel-gpv\ A\ C\ gpv\ gpv';\ lossless-gpv\ \mathcal{I}'\ gpv';\ rel-\mathcal{I}\ C\ (=)\ \mathcal{I}\ \mathcal{I}' \rrbracket$
 $\implies lossless-gpv\ \mathcal{I}\ gpv$
 $\langle proof \rangle$

lemma *rel-gpv''-colossless-gpvD1*:
assumes $rel: rel-gpv''\ A\ C\ R\ gpv\ gpv'$
and $gpv: colossless-gpv\ \mathcal{I}\ gpv$
and [*transfer-rule*]: $rel-\mathcal{I}\ C\ R\ \mathcal{I}\ \mathcal{I}'$
shows $colossless-gpv\ \mathcal{I}'\ gpv'$
 $\langle proof \rangle$

lemma *rel-gpv''-colossless-gpvD2*:
 $\llbracket rel-gpv''\ A\ C\ R\ gpv\ gpv';\ colossless-gpv\ \mathcal{I}'\ gpv';\ rel-\mathcal{I}\ C\ R\ \mathcal{I}\ \mathcal{I}' \rrbracket$
 $\implies colossless-gpv\ \mathcal{I}\ gpv$
 $\langle proof \rangle$

lemma *rel-gpv-colossless-gpvD1*:
 $\llbracket rel-gpv\ A\ C\ gpv\ gpv';\ colossless-gpv\ \mathcal{I}\ gpv;\ rel-\mathcal{I}\ C\ (=)\ \mathcal{I}\ \mathcal{I}' \rrbracket \implies colossless-gpv\ \mathcal{I}'\ gpv'$
 $\langle proof \rangle$

lemma *rel-gpv-colossless-gpvD2*:

$$\llbracket \text{rel-gpv } A \ C \ \text{gpv } \text{gpv}' ; \text{colossless-gpv } \mathcal{I}' \ \text{gpv}' ; \text{rel-}\mathcal{I} \ C \ (=) \ \mathcal{I} \ \mathcal{I}' \rrbracket$$

$$\implies \text{colossless-gpv } \mathcal{I} \ \text{gpv}$$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpv-parametric'*:

$$((=) \implies \text{rel-}\mathcal{I} \ C \ R \implies \text{rel-gpv}'' \ A \ C \ R \implies (=))$$

$$\text{gen-lossless-gpv } \text{gen-lossless-gpv}$$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpv-parametric [transfer-rule]*:

$$((=) \implies \text{rel-}\mathcal{I} \ C \ (=) \implies \text{rel-gpv } A \ C \implies (=))$$

$$\text{gen-lossless-gpv } \text{gen-lossless-gpv}$$
 $\langle \text{proof} \rangle$

end

lemma *gen-lossless-gpv-map-full [simp]*:

$$\text{gen-lossless-gpv } b \ \mathcal{I}\text{-full } (\text{map-gpv } f \ g \ \text{gpv}) = \text{gen-lossless-gpv } b \ \mathcal{I}\text{-full } \text{gpv}$$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *gen-lossless-gpv-map-id [simp]*:

$$\text{gen-lossless-gpv } b \ \mathcal{I} \ (\text{map-gpv } f \ \text{id } \text{gpv}) = \text{gen-lossless-gpv } b \ \mathcal{I} \ \text{gpv}$$
 $\langle \text{proof} \rangle$

lemma *results-gpv-try-gpv [simp]*:

$$\text{results-gpv } \mathcal{I} \ (\text{TRY } \text{gpv} \ \text{ELSE } \text{gpv}') =$$

$$\text{results-gpv } \mathcal{I} \ \text{gpv} \cup (\text{if } \text{colossless-gpv } \mathcal{I} \ \text{gpv} \ \text{then } \{\} \ \text{else } \text{results-gpv } \mathcal{I} \ \text{gpv}')$$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *results'-gpv-try-gpv [simp]*:

$$\text{results}'\text{-gpv } (\text{TRY } \text{gpv} \ \text{ELSE } \text{gpv}') =$$

$$\text{results}'\text{-gpv } \text{gpv} \cup (\text{if } \text{colossless-gpv } \mathcal{I}\text{-full } \text{gpv} \ \text{then } \{\} \ \text{else } \text{results}'\text{-gpv } \text{gpv}')$$
 $\langle \text{proof} \rangle$

lemma *outs'-gpv-try-gpv [simp]*:

$$\text{outs}'\text{-gpv } (\text{TRY } \text{gpv} \ \text{ELSE } \text{gpv}') =$$

$$\text{outs}'\text{-gpv } \text{gpv} \cup (\text{if } \text{colossless-gpv } \mathcal{I}\text{-full } \text{gpv} \ \text{then } \{\} \ \text{else } \text{outs}'\text{-gpv } \text{gpv}')$$
 $(\text{is } ?lhs = ?rhs)$
 $\langle \text{proof} \rangle$

lemma *pred-gpv-try [simp]*:

$$\text{pred-gpv } P \ Q \ (\text{try-gpv } \text{gpv} \ \text{gpv}') = (\text{pred-gpv } P \ Q \ \text{gpv} \wedge (\neg \text{colossless-gpv } \mathcal{I}\text{-full } \text{gpv} \longrightarrow \text{pred-gpv } P \ Q \ \text{gpv}'))$$
 $\langle \text{proof} \rangle$

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

assumes *lossless*: *lossless-gpv* \mathcal{I} *gpv*
and *WT*: $\mathcal{I} \vdash_g \text{gpv} \checkmark$
and *step*: $\bigwedge p. \llbracket$
 lossless-spmf p ;
 $\bigwedge \text{out } c. \text{IO out } c \in \text{set-spmf } p \implies \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I}$;
 $\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } p; \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} \rrbracket \implies \text{lossless-gpv } \mathcal{I} (c \text{ input})$;
 $\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } p; \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} \rrbracket \implies \mathcal{I} \vdash_g c \text{ input} \checkmark$;
 $\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } p; \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} \rrbracket \implies P (c \text{ input})$
 $\implies P (GPV \ p)$
shows $P \ \text{gpv}$
 $\langle \text{proof} \rangle$

lemma *lossless-gpv-induct-strong* [*consumes 1, case-names lossless-gpv*]:

assumes *gpv*: *lossless-gpv* \mathcal{I} *gpv*
and *step*:
 $\bigwedge p. \llbracket$ *lossless-spmf* p ;
 $\bigwedge \text{gpv}. \text{gpv} \in \text{sub-gpvs } \mathcal{I} (GPV \ p) \implies \text{lossless-gpv } \mathcal{I} \ \text{gpv}$;
 $\bigwedge \text{gpv}. \text{gpv} \in \text{sub-gpvs } \mathcal{I} (GPV \ p) \implies P \ \text{gpv} \rrbracket$
 $\implies P (GPV \ p)$
shows $P \ \text{gpv}$
 $\langle \text{proof} \rangle$

lemma *lossless-sub-gpvsI*:

assumes *spmf*: *lossless-spmf* (*the-gpv* *gpv*)
and *sub*: $\bigwedge \text{gpv}'. \text{gpv}' \in \text{sub-gpvs } \mathcal{I} \ \text{gpv} \implies \text{lossless-gpv } \mathcal{I} \ \text{gpv}'$
shows *lossless-gpv* \mathcal{I} *gpv*
 $\langle \text{proof} \rangle$

lemma *lossless-sub-gpvsD*:

assumes *lossless-gpv* \mathcal{I} *gpv* $\text{gpv}' \in \text{sub-gpvs } \mathcal{I} \ \text{gpv}$
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*: $\mathcal{I} \vdash_g \text{gpv} \checkmark$
and *step*: $\bigwedge p. \llbracket$ *lossless-spmf* p ;
 $\bigwedge \text{out } c. \text{IO out } c \in \text{set-spmf } p \implies \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I}$;
 $\bigwedge \text{gpv}. \text{gpv} \in \text{sub-gpvs } \mathcal{I} (GPV \ p) \implies \text{lossless-gpv } \mathcal{I} \ \text{gpv}$;
 $\bigwedge \text{gpv}. \text{gpv} \in \text{sub-gpvs } \mathcal{I} (GPV \ p) \implies \mathcal{I} \vdash_g \text{gpv} \checkmark$;
 $\bigwedge \text{gpv}. \text{gpv} \in \text{sub-gpvs } \mathcal{I} (GPV \ p) \implies P \ \text{gpv} \rrbracket$
 $\implies P (GPV \ p)$
shows $P \ \text{gpv}$
 $\langle \text{proof} \rangle$

lemma *try-gpv-gen-lossless*: — TODO: generalise to arbitrary typings ?

$gen\text{-}lossless\text{-}gpv\ b\ \mathcal{I}\text{-full}\ gpv \implies (TRY\ gpv\ ELSE\ gpv') = gpv$
 $\langle proof \rangle$

lemmas $try\text{-}gpv\text{-}lossless\ [simp] = try\text{-}gpv\text{-}gen\text{-}lossless[\mathbf{where}\ b=False]$
and $try\text{-}gpv\text{-}colossless\ [simp] = try\text{-}gpv\text{-}gen\text{-}lossless[\mathbf{where}\ b=True]$

lemma $try\text{-}gpv\text{-}bind\text{-}gen\text{-}lossless$: — TODO: generalise to arbitrary typings?

$gen\text{-}lossless\text{-}gpv\ b\ \mathcal{I}\text{-full}\ gpv \implies TRY\ bind\text{-}gpv\ gpv\ f\ ELSE\ gpv' = bind\text{-}gpv\ gpv$
 $(\lambda x. TRY\ f\ x\ ELSE\ gpv')$
 $\langle proof \rangle$

lemmas $try\text{-}gpv\text{-}bind\text{-}lossless = try\text{-}gpv\text{-}bind\text{-}gen\text{-}lossless[\mathbf{where}\ b=False]$
and $try\text{-}gpv\text{-}bind\text{-}colossless = try\text{-}gpv\text{-}bind\text{-}gen\text{-}lossless[\mathbf{where}\ b=True]$

lemma $try\text{-}gpv\text{-}cong$:

$\llbracket gpv = gpv''; \neg\ colossless\text{-}gpv\ \mathcal{I}\text{-full}\ gpv'' \implies gpv' = gpv''' \rrbracket$
 $\implies try\text{-}gpv\ gpv\ gpv' = try\text{-}gpv\ gpv''\ gpv'''$
 $\langle proof \rangle$

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 $catch\text{-}gpv\text{-}bind\text{-}gpv$: $catch\text{-}gpv\ (bind\text{-}gpv\ gpv\ f) = bind\text{-}gpv\ (catch\text{-}gpv\ gpv)$
 $(\lambda x. case\ x\ of\ None \Rightarrow Done\ None\ | Some\ x' \Rightarrow catch\text{-}gpv\ (f\ x'))$
 $\langle proof \rangle$

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*]:

$Some\ x \in results\text{-}gpv\ \mathcal{I}\ (catch\text{-}gpv\ gpv) \iff x \in results\text{-}gpv\ \mathcal{I}\ gpv$
<proof>

lemma *None-in-results-gpv-catch-gpv* [simp]:
 $None \in results\text{-}gpv\ \mathcal{I}\ (catch\text{-}gpv\ gpv) \longleftrightarrow \neg\ colossless\text{-}gpv\ \mathcal{I}\ gpv$
 ⟨proof⟩

lemma *results'-gpv-catch-gpv*:
 $results'\text{-}gpv\ (catch\text{-}gpv\ gpv) = Some\ 'results'\text{-}gpv\ gpv \cup (if\ colossless\text{-}gpv\ \mathcal{I}\\ full\ gpv\ then\ \{\}\ else\ \{None\})$
 ⟨proof⟩

lemma *Some-in-results'-gpv-catch-gpv* [simp]:
 $Some\ x \in results'\text{-}gpv\ (catch\text{-}gpv\ gpv) \longleftrightarrow x \in results'\text{-}gpv\ gpv$
 ⟨proof⟩

lemma *None-in-results'-gpv-catch-gpv* [simp]:
 $None \in results'\text{-}gpv\ (catch\text{-}gpv\ gpv) \longleftrightarrow \neg\ colossless\text{-}gpv\ \mathcal{I}\\ full\ gpv$
 ⟨proof⟩

lemma *results'-gpv-catch-gpvE*:
assumes $x \in results'\text{-}gpv\ (catch\text{-}gpv\ gpv)$
obtains $(Some)\ x'$
where $x = Some\ x'\ x' \in results'\text{-}gpv\ gpv$
 $| (colossless)\ x = None\ \neg\ colossless\text{-}gpv\ \mathcal{I}\\ full\ gpv$
 ⟨proof⟩

lemma *outs'-gpv-catch-gpv* [simp]: $outs'\text{-}gpv\ (catch\text{-}gpv\ gpv) = outs'\text{-}gpv\ gpv$
 ⟨proof⟩

lemma *pred-gpv-catch-gpv* [simp]: $pred\text{-}gpv\ (pred\text{-}option\ P)\ Q\ (catch\text{-}gpv\ gpv) = pred\text{-}gpv\ P\ Q\ gpv$
 ⟨proof⟩

abbreviation $bind\text{-}gpv' :: ('a, 'call, 'ret)\ gpv \Rightarrow ('a\ option \Rightarrow ('b, 'call, 'ret)\ gpv) \Rightarrow ('b, 'call, 'ret)\ gpv$
where $bind\text{-}gpv'\ gpv \equiv bind\text{-}gpv\ (catch\text{-}gpv\ gpv)$

lemma *bind-gpv'-assoc* [simp]: $bind\text{-}gpv'\ (bind\text{-}gpv'\ gpv\ f)\ g = bind\text{-}gpv'\ gpv\ (\lambda x. bind\text{-}gpv'\ (f\ x)\ g)$
 ⟨proof⟩

lemma *bind-gpv'-bind-gpv*: $bind\text{-}gpv'\ (bind\text{-}gpv\ gpv\ f)\ g = bind\text{-}gpv'\ gpv\ (case\ option\ (g\ None)\ (\lambda y. bind\text{-}gpv'\ (f\ y)\ g))$
 ⟨proof⟩

lemma *bind-gpv'-cong*:
 $\llbracket gpv = gpv'; \bigwedge x. x \in Some\ 'results'\text{-}gpv\ gpv' \vee (\neg\ colossless\text{-}gpv\ \mathcal{I}\\ full\ gpv \wedge x = None) \implies f\ x = f'\ x \rrbracket$

$\implies \text{bind-gpv}' \text{ gpv } f = \text{bind-gpv}' \text{ gpv}' f'$
 ⟨proof⟩

lemma *bind-gpv'-cong2*:

$\llbracket \text{gpv} = \text{gpv}' ; \bigwedge x. x \in \text{results}'\text{-gpv } \text{gpv}' \implies f (\text{Some } x) = f' (\text{Some } x); \neg$
colossless-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: $\bigwedge \text{gpv } \text{gpv}'. R \text{ gpv } \text{gpv}'$

$\implies \text{rel-spmf } (\text{rel-generat } (=) (=) (\text{rel-fun } (=) (\lambda \text{gpv } \text{gpv}'. R \text{ gpv } \text{gpv}' \vee \text{gpv} =$
 $\text{gpv}' \vee$

$(\exists \text{gpv2} :: ('b, 'call, 'ret) \text{ gpv}. \exists \text{gpv2}' :: ('c, 'call, 'ret) \text{ gpv}. \exists f f'. \text{gpv} =$
 $\text{bind-gpv } \text{gpv2 } f \wedge \text{gpv}' = \text{bind-gpv } \text{gpv2}' f' \wedge$

$\text{rel-gpv } (\lambda x y. R (f x) (f' y)) (=) \text{gpv2 } \text{gpv2}'))$

$(\text{the-gpv } \text{gpv}) (\text{the-gpv } \text{gpv}')$

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 (*spm*) *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*:

$\text{inline1 } \text{gpv } s =$

$\text{the-gpv } \text{gpv} \gg=$

$\text{case-generat } (\lambda x. \text{return-spmf } (\text{Inl } (x, s)))$

$(\lambda \text{out } \text{rpv}. \text{the-gpv } (\text{callee } s \text{ out}) \gg=$

$\text{case-generat } (\lambda(x, y). \text{inline1 } (\text{rpv } x) y)$

($\lambda out\ rpv'.\ return\text{-}spm\ f\ (Inr\ (out,\ rpv',\ rpv))$)
 $\langle proof \rangle$

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 $P\ (\lambda -.\ return\text{-}pm\ f\ None)$
and $\bigwedge inline1'.\ P\ inline1' \implies P\ (\lambda gpv\ s.\ the\text{-}gpv\ gpv \ggg case\text{-}generat\ (\lambda x.\$
 $return\text{-}spm\ f\ (Inl\ (x,\ s)))\ (\lambda out\ rpv.\ the\text{-}gpv\ (callee\ s\ out) \ggg case\text{-}generat\ (\lambda(x,\$
 $y).\ inline1'\ (rpv\ x)\ y)\ (\lambda out\ rpv'.\ return\text{-}spm\ f\ (Inr\ (out,\ rpv',\ rpv))))$)
shows $P\ inline1$
 $\langle proof \rangle$

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 $P\ (\lambda -.\ return\text{-}pm\ f\ None)$
and $\bigwedge inline1'.\ \llbracket \bigwedge gpv\ s.\ ord\text{-}spm\ f\ (=)\ (inline1'\ gpv\ s)\ (inline1\ gpv\ s); P\ inline1' \rrbracket$
 $\implies P\ (\lambda gpv\ s.\ the\text{-}gpv\ gpv \ggg case\text{-}generat\ (\lambda x.\ return\text{-}spm\ f\ (Inl\ (x,\ s)))\ (\lambda out$
 $rpv.\ the\text{-}gpv\ (callee\ s\ out) \ggg case\text{-}generat\ (\lambda(x,\ y).\ inline1'\ (rpv\ x)\ y)\ (\lambda out\ rpv'.$
 $return\text{-}spm\ f\ (Inr\ (out,\ rpv',\ rpv))))$)
shows $P\ inline1$
 $\langle proof \rangle$

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 $P\ (\lambda -.\ return\text{-}pm\ f\ None)$
and $\bigwedge inline1'.$
 $\llbracket \bigwedge gpv\ s.\ ord\text{-}spm\ f\ (=)\ (inline1'\ gpv\ s)\ (inline1\ gpv\ s);$
 $\bigwedge gpv\ s.\ ord\text{-}spm\ f\ (=)\ (inline1'\ gpv\ s)\ (the\text{-}gpv\ gpv \ggg case\text{-}generat\ (\lambda x.$
 $return\text{-}spm\ f\ (Inl\ (x,\ s)))\ (\lambda out\ rpv.\ the\text{-}gpv\ (callee\ s\ out) \ggg case\text{-}generat\ (\lambda(x,\$
 $y).\ inline1'\ (rpv\ x)\ y)\ (\lambda out\ rpv'.\ return\text{-}spm\ f\ (Inr\ (out,\ rpv',\ rpv))))$);
 $P\ inline1' \rrbracket$
 $\implies P\ (\lambda gpv\ s.\ the\text{-}gpv\ gpv \ggg case\text{-}generat\ (\lambda x.\ return\text{-}spm\ f\ (Inl\ (x,\ s)))\ (\lambda out$
 $rpv.\ the\text{-}gpv\ (callee\ s\ out) \ggg case\text{-}generat\ (\lambda(x,\ y).\ inline1'\ (rpv\ x)\ y)\ (\lambda out\ rpv'.$
 $return\text{-}spm\ f\ (Inr\ (out,\ rpv',\ rpv))))$)
shows $P\ inline1$
 $\langle proof \rangle$

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*

$:: ('a,\ 'call,\ 'ret)\ gpv \times 's + ('ret \Rightarrow ('a,\ 'call,\ 'ret)\ gpv) \times ('ret \times 's,\ 'call', 'ret')\ gpv$

$\Rightarrow ('a \times 's, 'call', 'ret')\text{ gpv}$
where
 \wedge state. the-gpv (inline-aux state) =
(case state of Inl (c, s) \Rightarrow map-spmf (λ result.
case result of Inl (x, s) \Rightarrow Pure (x, s)
| Inr (out, oracle, rpv) \Rightarrow IO out (λ input. inline-aux (Inr (rpv, oracle input))))
(inline1 c s)
| Inr (rpv, c) \Rightarrow
map-spmf (λ result.
case result of Inl (Inl (x, s)) \Rightarrow Pure (x, s)
| Inl (Inr (out, oracle, rpv)) \Rightarrow IO out (λ input. inline-aux (Inr (rpv, oracle
input))))
| Inr (out, c) \Rightarrow IO out (λ input. inline-aux (Inr (rpv, c input))))
(bind-spmf (the-gpv c) (λ generat. case generat of Pure (x, s') \Rightarrow (map-spmf Inl
(inline1 (rpv x) s'))
| IO out c \Rightarrow return-spmf (Inr (out, c)))
))

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)
<proof>

lemma inline-sel:

the-gpv (inline c s) =
map-spmf (λ result. case result of Inl xs \Rightarrow Pure xs
| Inr (out, oracle, rpv) \Rightarrow IO out (λ input. bind-gpv (oracle
input) ($\lambda(x, s')$. inline (rpv x) s')) (inline1 c s)
<proof>

lemma inline1-Fail [simp]: inline1 Fail s = return-pmf None
<proof>

lemma inline-Fail [simp]: inline Fail s = Fail
<proof>

lemma inline1-Done [simp]: inline1 (Done x) s = return-spmf (Inl (x, s))
<proof>

lemma inline-Done [simp]: inline (Done x) s = Done (x, s)
<proof>

lemma inline1-lift-spmf [simp]: inline1 (lift-spmf p) s = map-spmf (λx . Inl (x,
s)) p
<proof>

lemma *inline-lift-spmf* [*simp*]: *inline* (*lift-spmf* *p*) *s* = *lift-spmf* (*map-spmf* ($\lambda x. (x, s)$) *p*)
 ⟨*proof*⟩

lemma *inline1-Pause*:

inline1 (*Pause* *out* *c*) *s* =
the-gpv (*callee* *s* *out*) $\gg=$ ($\lambda \text{react}. \text{case react of Pure } (x, s') \Rightarrow \text{inline1 } (c \ x) \ s' \mid$
IO *out'* *c'* $\Rightarrow \text{return-spmf } (\text{Inr } (\text{out}', c', c))$)
 ⟨*proof*⟩

lemma *inline-Pause* [*simp*]:

inline (*Pause* *out* *c*) *s* = *callee* *s* *out* $\gg=$ ($\lambda(x, s'). \text{inline } (c \ x) \ s'$)
 ⟨*proof*⟩

lemma *inline1-bind-gpv*:

fixes *gpv* *f* *s*
defines [*simp*]: *inline11* \equiv *inline1* **and** [*simp*]: *inline12* \equiv *inline1* **and** [*simp*]:
inline13 \equiv *inline1*
shows *inline11* (*bind-gpv* *gpv* *f*) *s* = *bind-spmf* (*inline12* *gpv* *s*)
 ($\lambda \text{res}. \text{case res of Inl } (x, s') \Rightarrow \text{inline13 } (f \ x) \ s' \mid \text{Inr } (\text{out}, \text{rpv}', \text{rpv}) \Rightarrow$
return-spmf (*Inr* (*out*, *rpv'*, *bind-rpv* *rpv* *f*)))
 (**is** ?*lhs* = ?*rhs*)
 ⟨*proof*⟩

lemma *inline-bind-gpv* [*simp*]:

inline (*bind-gpv* *gpv* *f*) *s* = *bind-gpv* (*inline* *gpv* *s*) ($\lambda(x, s'). \text{inline } (f \ x) \ s'$)
 ⟨*proof*⟩

end

lemma *set-inline1-lift-spmf1*: *set-spmf* (*inline1* ($\lambda s \ x. \text{lift-spmf } (p \ s \ x)$) *gpv* *s*) \subseteq
range *Inl*
 ⟨*proof*⟩

lemma *in-set-inline1-lift-spmf1*: $y \in \text{set-spmf } (\text{inline1 } (\lambda s \ x. \text{lift-spmf } (p \ s \ x))$
gpv *s*) $\implies \exists r \ s'. y = \text{Inl } (r, s')$
 ⟨*proof*⟩

lemma *inline-lift-spmf1*:

fixes *p* **defines** *callee* $\equiv \lambda s \ c. \text{lift-spmf } (p \ s \ c)$
shows *inline* *callee* *gpv* *s* = *lift-spmf* (*map-spmf* *projl* (*inline1* *callee* *gpv* *s*))
 ⟨*proof*⟩

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

lemma *inline1-parametric'*:

($(S \text{====>} C \text{====>} \text{rel-gpv'' } (\text{rel-prod } R \ S) \ C' \ R') \text{====>} \text{rel-gpv'' } A \ C \ R$
 $\text{====>} S$
 $\text{====>} \text{rel-spmf } (\text{rel-sum } (\text{rel-prod } A \ S) \ (\text{rel-prod } C' \ (\text{rel-prod } (R' \ \text{====>})))$

$rel\text{-}gpv'' (rel\text{-}prod R S) C' R' (R ==> rel\text{-}gpv'' A C R))$
 $inline1 inline1$
 $(is (- ==> ?R) - -)$
 $\langle proof \rangle$

lemma $inline1\text{-}parametric [transfer\text{-}rule]$:
 $((S ==> C ==> rel\text{-}gpv (rel\text{-}prod (=) S) C') ==> rel\text{-}gpv A C ==> S$
 $==> rel\text{-}spmf (rel\text{-}sum (rel\text{-}prod A S) (rel\text{-}prod C' (rel\text{-}prod (rel\text{-}rpv (rel\text{-}prod$
 $(=) S) C') (rel\text{-}rpv A C))))$
 $inline1 inline1$
 $\langle proof \rangle$

lemma $inline\text{-}parametric'$:
notes $[transfer\text{-}rule] = inline1\text{-}parametric' the\text{-}gpv\text{-}parametric' corec\text{-}gpv\text{-}parametric'$
shows $((S ==> C ==> rel\text{-}gpv'' (rel\text{-}prod R S) C' R') ==> rel\text{-}gpv'' A$
 $C R ==> S ==> rel\text{-}gpv'' (rel\text{-}prod A S) C' R')$
 $inline inline$
 $\langle proof \rangle$

lemma $inline\text{-}parametric [transfer\text{-}rule]$:
 $((S ==> C ==> rel\text{-}gpv (rel\text{-}prod (=) S) C') ==> rel\text{-}gpv A C ==>$
 $S ==> rel\text{-}gpv (rel\text{-}prod A S) C')$
 $inline inline$
 $\langle proof \rangle$
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

$inline2 gpv s2 s1 =$
 $bind\text{-}spmf (the\text{-}gpv gpv)$
 $(case\text{-}generat (\lambda x. return\text{-}spmf (Inl (x, s2, s1)))$
 $(\lambda out rpv. bind\text{-}spmf (inline1 callee1 (callee2 s2 out) s1)$
 $(case\text{-}sum (\lambda((r2, s2), s1). inline2 (rpv r2) s2 s1)$
 $(\lambda(x, rpv'', rpv'). return\text{-}spmf (Inr (x, rpv'', rpv'))))))$

lemma $inline2\text{-}fixp\text{-}induct [case\text{-}names adm bottom step]$:
assumes $ccpo.admissible (fun\text{-}lub lub\text{-}spmf) (fun\text{-}ord (ord\text{-}spmf (=))) (\lambda inline2.$
 $P (\lambda gpv s2 s1. inline2 ((gpv, s2), s1)))$
and $P (\lambda - -. return\text{-}pmf None)$
and $\bigwedge inline2'. P inline2' \implies$
 $P (\lambda gpv s2 s1. bind\text{-}spmf (the\text{-}gpv gpv) (\lambda generat. case generat of$

$Pure\ x \Rightarrow return\text{-}spmf\ (Inl\ (x,\ s2,\ s1))$
 $| IO\ out\ rpv \Rightarrow bind\text{-}spmf\ (inline1\ callee1\ (callee2\ s2\ out)\ s1)\ (\lambda lr.\ case\ lr$
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 $inline1\ callee1\ (inline\ callee2\ gpv\ s2)\ s1 =$
 $map\text{-}spmf\ (map\text{-}sum\ (\lambda(x,\ (s2,\ s1)).\ ((x,\ s2),\ s1))$
 $\quad (\lambda(x,\ rpv'',\ rpv',\ rpv).\ (x,\ rpv'',\ \lambda r1.\ rpv'\ r1 \gg= (\lambda(r2,\ s2).\ inline\ callee2\ (rpv$
 $\quad r2)\ s2))))$
 $\quad (inline2\ gpv\ s2\ s1)$
(is $?lhs = ?rhs$)
 $\langle proof \rangle$

lemma *inline1-inline-conv-inline2'*:

$inline1\ (\lambda(s2,\ s1)\ c2.\ map\text{-}gpv\ (\lambda((r,\ s2),\ s1).\ (r,\ s2,\ s1))\ id\ (inline\ callee1$
 $(callee2\ s2\ c2)\ s1))\ gpv\ (s2,\ s1) =$
 $map\text{-}spmf\ (map\text{-}sum\ id\ (\lambda(x,\ rpv'',\ rpv',\ rpv).\ (x,\ \lambda r.\ bind\text{-}gpv\ (rpv''\ r)$
 $\quad (\lambda(r1,\ s1).\ map\text{-}gpv\ (\lambda((r2,\ s2),\ s1).\ (r2,\ s2,\ s1))\ id\ (inline\ callee1\ (rpv'$
 $\quad r1)\ s1)),\ rpv))$
 $\quad (inline2\ gpv\ s2\ s1)$
(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*: $y \in set\text{-}spmf\ (inline2\ (\lambda s\ x.\ lift\text{-}spmf\ (p\ s\ x))$
 $callee\ gpv\ s\ s') \Longrightarrow \exists r\ s\ s'. y = Inl\ (r,\ s,\ s')$
 $\langle proof \rangle$

context

fixes $consider' :: 'call \Rightarrow bool$
and $consider :: 'call' \Rightarrow bool$
and $callee :: 's \Rightarrow 'call \Rightarrow ('ret \times 's,\ 'call', 'ret')\ gpv$
notes $[[function\text{-}internals]]$

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 ($\lambda x. \text{return-spmf (Inl (x, s))}$)

($\lambda out \text{ rpv. the-gpv (callee s out) } \ggg$

case-generat ($\lambda(x, y). \text{inline1' (rpv x) y}$)

($\lambda out' \text{ rpv}'. \text{return-spmf (Inr (out, out', rpv', rpv))}$))

private lemma *inline1'-fixp-induct* [*case-names adm bottom step*]:

assumes *ccpo.admissible (fun-lub lub-spmf) (fun-ord (ord-spmf (=)))* ($\lambda \text{inline1'}$.

P ($\lambda \text{gpv s. inline1' (gpv, s)}$)

and *P* ($\lambda - . \text{return-pmf None}$)

and $\bigwedge \text{inline1'}. P \text{ inline1'} \implies P (\lambda \text{gpv s. the-gpv gpv } \ggg \text{ case-generat } (\lambda x. \text{return-spmf (Inl (x, s))}) (\lambda out \text{ rpv. the-gpv (callee s out) } \ggg \text{ case-generat } (\lambda(x, y). \text{inline1' (rpv x) y}) (\lambda out' \text{ rpv}'. \text{return-spmf (Inr (out, out', rpv', rpv))}))$)

shows *P inline1'*

<proof> **lemma** *inline1-conv-inline1'*: *inline1 callee gpv s = map-spmf (map-sum id snd) (inline1' gpv s)*

<proof>

context

fixes *q* :: *enat*

assumes *q*: $\bigwedge s \ x. \text{consider' } x \implies \text{interaction-bound consider (callee s x)} \leq q$

and *ignore*: $\bigwedge s \ x. \neg \text{consider' } x \implies \text{interaction-bound consider (callee s x)} = 0$

begin

private lemma *interaction-bound-inline1'-aux*:

interaction-bound consider' gpv $\leq p$

$\implies \text{set-spmf (inline1' gpv s)} \subseteq \{\text{Inr (out', out, c', rpv)} \mid \text{out' out c' rpv.}$

if consider' out'

then ($\forall \text{input. (if consider out then eSuc (interaction-bound consider (c' input)) else interaction-bound consider (c' input))} \leq q$) \wedge

($\forall x. \text{eSuc (interaction-bound consider' (rpv x))} \leq p$)

else $\neg \text{consider out} \wedge (\forall \text{input. interaction-bound consider (c' input)} = 0)$

$\wedge (\forall x. \text{interaction-bound consider' (rpv x)} \leq p)$

$\cup \text{range Inl}$

<proof>

lemma *interaction-bound-inline1'*:

$\llbracket \text{Inr (out', out, c', rpv)} \in \text{set-spmf (inline1' gpv s); interaction-bound consider' gpv} \leq p \rrbracket$

$\implies \text{if consider' out' then}$

(if consider out then eSuc (interaction-bound consider (c' input)) else

$interaction\text{-}bound\ consider\ (c'\ input) \leq q \wedge$
 $\quad eSuc\ (interaction\text{-}bound\ consider'\ (rpv\ x)) \leq p$
 $\quad else\ \neg\ consider\ out \wedge interaction\text{-}bound\ consider\ (c'\ input) = 0 \wedge interaction\text{-}bound$
 $consider'\ (rpv\ x) \leq p$
 <proof>

end

lemma *interaction-bounded-by-inline1*:

$\llbracket Inr\ (out',\ out,\ c',\ rpv) \in set\text{-}spmf\ (inline1'\ gpv\ s);$
 $\quad interaction\text{-}bounded\text{-}by\ consider'\ gpv\ p;$
 $\quad \bigwedge s\ x.\ consider'\ x \implies interaction\text{-}bounded\text{-}by\ consider\ (callee\ s\ x)\ q;$
 $\quad \bigwedge s\ x.\ \neg\ consider'\ x \implies interaction\text{-}bounded\text{-}by\ consider\ (callee\ s\ x)\ 0 \rrbracket$
 \implies if *consider'* *out'* then
 $\quad (if\ consider\ out\ then\ q \neq 0 \wedge interaction\text{-}bounded\text{-}by\ consider\ (c'\ input)\ (q$
 $- 1)$ else *interaction-bounded-by consider* (*c' input*) *q* \wedge
 $\quad p \neq 0 \wedge interaction\text{-}bounded\text{-}by\ consider'\ (rpv\ x)\ (p - 1)$
 else $\neg\ consider\ out \wedge interaction\text{-}bounded\text{-}by\ consider\ (c'\ input)\ 0 \wedge interaction\text{-}bounded\text{-}by$
 $consider'\ (rpv\ x)\ p$
 <proof>

declare *enat-0-iff* [simp]

lemma *interaction-bounded-by-inline* [*interaction-bound*]:

assumes *p*: *interaction-bounded-by consider'* *gpv p*
and *q*: $\bigwedge s\ x.\ consider'\ x \implies interaction\text{-}bounded\text{-}by\ consider\ (callee\ s\ x)\ q$
and *ignore*: $\bigwedge s\ x.\ \neg\ consider'\ x \implies interaction\text{-}bounded\text{-}by\ consider\ (callee\ s\ x)$
 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 \Rightarrow bool
and *consider* :: 'call' \Rightarrow bool
and *callee* :: 's \Rightarrow 'call \Rightarrow ('ret \times 's, 'call', 'ret') *gpv*
and *gpv* :: ('a, 'call, 'ret) *gpv*
assumes *p*: *interaction-bounded-by consider'* *gpv p*
and *q*: $\bigwedge s\ x.\ \llbracket I\ s;\ consider'\ x \rrbracket \implies interaction\text{-}bounded\text{-}by\ consider\ (callee\ s$
 $x)\ q$
and *ignore*: $\bigwedge s\ x.\ \llbracket I\ s;\ \neg\ consider'\ x \rrbracket \implies interaction\text{-}bounded\text{-}by\ consider$
 $(callee\ s\ x)\ 0$
and *I*: *I s*
and *invariant*: $\bigwedge s\ x\ y\ s'. \llbracket (y,\ s') \in results'\text{-}gpv\ (callee\ s\ x); I\ s \rrbracket \implies I\ s'$
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 $results: \bigwedge s x. x \in outs\text{-}\mathcal{I} \mathcal{I} \Longrightarrow results\text{-}gpv \mathcal{I}' (callee\ s\ x) \subseteq responses\text{-}\mathcal{I}$
 $\mathcal{I}\ x \times UNIV$
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* $\mathcal{I}' \ (\text{rpv } \text{input})$
 $\langle \text{proof} \rangle$

lemma *lossless-results-inline1*:
assumes *Inr* (*out*, *rpv*, *rpv'*) $\in \text{set-spmf } (\text{inline1 callee gpv } s)$
and (*x*, *s'*) $\in \text{results-gpv } \mathcal{I}' \ (\text{rpv } \text{input})$
and *input* $\in \text{responses-}\mathcal{I} \ \mathcal{I}' \ \text{out}$
shows *lossless-gpv* $\mathcal{I} \ (\text{rpv}' \ x)$
 $\langle \text{proof} \rangle$

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* $\mathcal{I}' \ (\text{inline callee gpv } s)$
 $\langle \text{proof} \rangle$

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

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* ⇒ 's ⇒ ('a × 's) *spmf*

where

exec-gpv *c* *s* =
the-gpv *c* ≫≧
case-generat (λ*x*. *return-spmf* (*x*, *s*))
(λ*out* *c*. *callee* *s* *out* ≫≧ (λ(*x*, *y*). *exec-gpv* (*c* *x*) *y*))

abbreviation *run-gpv* :: ('a, 'call, 'ret) *gpv* ⇒ 's ⇒ 'a *spmf*

where *run-gpv* *gpv* *s* ≡ *map-spmf* *fst* (*exec-gpv* *gpv* *s*)

lemma *exec-gpv-fixp-induct* [*case-names adm bottom step*]:

assumes *ccpo.admissible* (*fun-lub* *lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*f*. *P* (λ*c* *s*. *f* (*c*, *s*)))

and *P* (λ- -. *return-pmf* *None*)

and ∧ *exec-gpv*. *P* *exec-gpv* ⇒

P (λ*c* *s*. *the-gpv* *c* ≫≧ *case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out* *c*. *callee* *s* *out* ≫≧ (λ(*x*, *y*). *exec-gpv* (*c* *x*) *y*)))

shows *P* *exec-gpv*

⟨*proof*⟩

lemma *exec-gpv-fixp-induct-strong* [*case-names adm bottom step*]:

assumes *ccpo.admissible* (*fun-lub* *lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*f*. *P* (λ*c* *s*. *f* (*c*, *s*)))

and *P* (λ- -. *return-pmf* *None*)

and ∧ *exec-gpv'*. [∧ *c* *s*. *ord-spmf* (=) (*exec-gpv'* *c* *s*) (*exec-gpv* *c* *s*); *P* *exec-gpv'*]

⇒ *P* (λ*c* *s*. *the-gpv* *c* ≫≧ *case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out* *c*. *callee* *s* *out* ≫≧ (λ(*x*, *y*). *exec-gpv'* (*c* *x*) *y*)))

shows *P* *exec-gpv*

⟨*proof*⟩

lemma *exec-gpv-fixp-induct-strong2* [*case-names adm bottom step*]:

assumes *ccpo.admissible* (*fun-lub* *lub-spmf*) (*fun-ord* (*ord-spmf* (=))) (λ*f*. *P* (λ*c* *s*. *f* (*c*, *s*)))

and *P* (λ- -. *return-pmf* *None*)

and ∧ *exec-gpv'*.

[∧ *c* *s*. *ord-spmf* (=) (*exec-gpv'* *c* *s*) (*exec-gpv* *c* *s*);

∧ *c* *s*. *ord-spmf* (=) (*exec-gpv'* *c* *s*) (*the-gpv* *c* ≫≧ *case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out* *c*. *callee* *s* *out* ≫≧ (λ(*x*, *y*). *exec-gpv'* (*c* *x*) *y*)))];

P *exec-gpv'*]

⇒ *P* (λ*c* *s*. *the-gpv* *c* ≫≧ *case-generat* (λ*x*. *return-spmf* (*x*, *s*)) (λ*out* *c*. *callee* *s* *out* ≫≧ (λ(*x*, *y*). *exec-gpv'* (*c* *x*) *y*)))

shows *P* *exec-gpv*

⟨*proof*⟩

end

lemma *exec-gpv-conv-inline1*:

$exec\text{-}gpv\ callee\ gpv\ s = map\text{-}spmf\ projl\ (inline1\ (\lambda s\ c.\ lift\text{-}spmf\ (callee\ s\ c) :: (-, unit, unit)\ gpv)\ gpv\ s)$
 $\langle proof \rangle$

lemma $exec\text{-}gpv\text{-}simps$:

$exec\text{-}gpv\ callee\ gpv\ s =$
 $the\text{-}gpv\ gpv \gg=$
 $case\text{-}generat\ (\lambda x.\ return\text{-}spmf\ (x, s))$
 $(\lambda out\ rpv.\ callee\ s\ out \gg= (\lambda(x, y).\ exec\text{-}gpv\ callee\ (rpv\ x)\ y))$
 $\langle proof \rangle$

lemma $exec\text{-}gpv\text{-}lift\text{-}spmf\ [simp]$:

$exec\text{-}gpv\ callee\ (lift\text{-}spmf\ p)\ s = bind\text{-}spmf\ p\ (\lambda x.\ return\text{-}spmf\ (x, s))$
 $\langle proof \rangle$

lemma $exec\text{-}gpv\text{-}Done\ [simp]$: $exec\text{-}gpv\ callee\ (Done\ x)\ s = return\text{-}spmf\ (x, s)$

$\langle proof \rangle$

lemma $exec\text{-}gpv\text{-}Fail\ [simp]$: $exec\text{-}gpv\ callee\ Fail\ s = return\text{-}pmf\ None$

$\langle proof \rangle$

lemma $if\text{-}distrib\text{-}exec\text{-}gpv\ [if\text{-}distrib]$:

$exec\text{-}gpv\ callee\ (if\ b\ then\ x\ else\ y)\ s = (if\ b\ then\ exec\text{-}gpv\ callee\ x\ s\ else\ exec\text{-}gpv\ callee\ y\ s)$
 $\langle proof \rangle$

lemmas $exec\text{-}gpv\text{-}fixp\text{-}parallel\text{-}induct\ [case\text{-}names\ adm\ bottom\ step] =$

$parallel\text{-}fixp\text{-}induct\text{-}2\text{-}2[OF\ partial\text{-}function\text{-}definitions\text{-}spmf\ partial\text{-}function\text{-}definitions\text{-}spmf\ exec\text{-}gpv.mono\ exec\text{-}gpv.mono\ exec\text{-}gpv\text{-}def\ exec\text{-}gpv\text{-}def,\ unfolded\ lub\text{-}spmf\text{-}empty]$

context **includes** $lifting\text{-}syntax$ **begin**

lemma $exec\text{-}gpv\text{-}parametric'$:

$((S ==> CALL ==> rel\text{-}spmf\ (rel\text{-}prod\ R\ S)) ==> rel\text{-}gpv''\ A\ CALL\ R$
 $==> S ==> rel\text{-}spmf\ (rel\text{-}prod\ A\ S))$
 $exec\text{-}gpv\ exec\text{-}gpv$
 $\langle proof \rangle$

lemma $exec\text{-}gpv\text{-}parametric\ [transfer\text{-}rule]$:

$((S ==> CALL ==> rel\text{-}spmf\ (rel\text{-}prod\ ((=) :: 'ret \Rightarrow -)\ S)) ==> rel\text{-}gpv$
 $A\ CALL ==> S ==> rel\text{-}spmf\ (rel\text{-}prod\ A\ S))$
 $exec\text{-}gpv\ exec\text{-}gpv$
 $\langle proof \rangle$

end

lemma $exec\text{-}gpv\text{-}bind$: $exec\text{-}gpv\ callee\ (c \gg= f)\ s = exec\text{-}gpv\ callee\ c\ s \gg= (\lambda(x, s') \Rightarrow exec\text{-}gpv\ callee\ (f\ x)\ s')$

$\langle proof \rangle$

lemma *exec-gpv-map-gpv-id*:

exec-gpv oracle (map-gpv f id gpv) σ = map-spmf (apfst f) (exec-gpv oracle gpv σ)
 \langle proof \rangle

lemma *exec-gpv-Pause [simp]*:

exec-gpv callee (Pause out f) s = callee s out \gg ($\lambda(x, s').$ exec-gpv callee (f x) s')
 \langle proof \rangle

lemma *exec-gpv-bind-lift-spmf*:

exec-gpv callee (bind-gpv (lift-spmf p) f) s = bind-spmf p ($\lambda x.$ exec-gpv callee (f x) s)
 \langle proof \rangle

lemma *exec-gpv-bind-option [simp]*:

exec-gpv oracle (monad.bind-option Fail x f) s = monad.bind-option (return-pmf None) x ($\lambda a.$ exec-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-gpv A C gpv; pred-fun S (pred-fun C (pred-spmf (pred-prod ($\lambda.$ True) S))) callee; S s* \rrbracket
 \implies *pred-spmf (pred-prod A S) (exec-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-gpv callee (inline gpv c' s') s =*
*map-spmf ($\lambda(x, s', s).$ ((x, s'), s)) (exec-gpv ($\lambda(s', s)$ y. map-spmf ($\lambda((x, s'),$
s). (x, s', s)) (exec-gpv callee (gpv s' y) s)) c' (s', s))
(is ?lhs = ?rhs)
 \langle proof \rangle*

lemma *ord-spmf-exec-gpv*:

assumes *callee*: $\bigwedge s x.$ *ord-spmf (=) (callee1 s x) (callee2 s x)*
shows *ord-spmf (=) (exec-gpv callee1 gpv s) (exec-gpv callee2 gpv s)*
 \langle proof \rangle

context **fixes** *callee* :: ('call, 'ret, 's) callee **notes** \llbracket *function-internals* \rrbracket **begin**

partial-function (*spmf*) *execp-resumption* :: ('a, 'call, 'ret) *resumption* \Rightarrow 's \Rightarrow ('a \times 's) *spmf*

where

execp-resumption r s = (case r of resumption.Done x \Rightarrow return-pmf (map-option

$(\lambda a. (a, s)) x$
 $| \text{resumption.Pause out } c \Rightarrow \text{bind-spmf (callee } s \text{ out) } (\lambda(\text{input}, s'). \text{excep-resumption (} c \text{ input) } s')$

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

lemma *excep-resumption-ABORT* [simp]: *excep-resumption ABORT s = return-pmf None*
 $\langle \text{proof} \rangle$

lemma *excep-resumption-DONE* [simp]: *excep-resumption (DONE x) s = return-spmf (x, s)*
 $\langle \text{proof} \rangle$

lemma *exec-gpv-lift-resumption*: *exec-gpv callee (lift-resumption r) s = excep-resumption r s*
 $\langle \text{proof} \rangle$

lemma *mcont2mcont-excep-resumption* [THEN *spmf.mcont2mcont, cont-intro, simp*]:
shows *mcont-excep-resumption*:
mcont resumption-lub resumption-ord lub-spmf (ord-spmf (=)) $(\lambda r. \text{excep-resumption } r \text{ } s)$
 $\langle \text{proof} \rangle$

lemma *excep-resumption-bind* [simp]:
excep-resumption (r \ggg f) s = excep-resumption r s \ggg $(\lambda(x, s'). \text{excep-resumption (} f \text{ } x) \text{ } s')$
 $\langle \text{proof} \rangle$

lemma *pred-spmf-excep-resumption*:
 $\bigwedge A. \llbracket \text{pred-resumption } A \text{ } C \text{ } r; \text{pred-fun } S \text{ (pred-fun } C \text{ (pred-spmf (pred-prod } (\lambda-. \text{True) } S))) \text{ } \text{callee; } S \text{ } s \rrbracket$
 $\implies \text{pred-spmf (pred-prod } A \text{ } S) (\text{excep-resumption } r \text{ } s)$
 $\langle \text{proof} \rangle$

end

inductive *WT-callee* :: $('call, 'ret) \mathcal{I} \Rightarrow ('call \Rightarrow ('ret \times 's) \text{spmf}) \Rightarrow \text{bool } ((-)$
 $\vdash_c / (-) \checkmark [100, 0] 99)$

for \mathcal{I} *callee*

where

WT-callee:

$\llbracket \bigwedge \text{call } ret \text{ } s. \llbracket \text{call} \in \text{outs-}\mathcal{I} \text{ } \mathcal{I}; (ret, s) \in \text{set-spmf (callee call)} \rrbracket \implies ret \in \text{responses-}\mathcal{I} \text{ } \mathcal{I} \text{ call} \rrbracket$

$\implies \mathcal{I} \vdash_c \text{callee} \checkmark$

lemmas *WT-calleeI = WT-callee*

hide-fact *WT-callee*

lemma *WT-calleeD*: $\llbracket \mathcal{I} \vdash c \text{ callee } \checkmark; (ret, s) \in \text{set-spmf } (\text{callee } out); out \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies ret \in \text{responses-}\mathcal{I} \mathcal{I} \text{ out}$
 $\langle \text{proof} \rangle$

lemma *WT-callee-full* [*intro!*, *simp*]: $\mathcal{I}\text{-full} \vdash c \text{ callee } \checkmark$
 $\langle \text{proof} \rangle$

lemma *WT-callee-parametric* [*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique R*
shows $(\text{rel-}\mathcal{I} \ C \ R \implies (C \implies \text{rel-spmf } (\text{rel-prod } R \ S)) \implies (=))$
 $WT\text{-callee } WT\text{-callee}$
 $\langle \text{proof} \rangle$

locale *callee-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-invariant-on* = *callee-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-invariant*: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies I \ s'$
and *WT-callee*: $\bigwedge s. I \ s \implies \mathcal{I} \vdash c \text{ callee } s \checkmark$
begin

lemma *callee-invariant'*: $\llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies I \ s' \wedge y \in \text{responses-}\mathcal{I} \mathcal{I} \ x$
 $\langle \text{proof} \rangle$

lemma *exec-gpv-invariant'*:
 $\llbracket I \ s; \mathcal{I} \vdash g \text{ gpv } \checkmark \rrbracket \implies \text{set-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s) \subseteq \{(x, s'). I \ s'\}$
 $\langle \text{proof} \rangle$

lemma *exec-gpv-invariant*:
 $\llbracket (x, s') \in \text{set-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s); I \ s; \mathcal{I} \vdash g \text{ gpv } \checkmark \rrbracket \implies I \ s'$
 $\langle \text{proof} \rangle$

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

and $WT: \mathcal{I} \vdash g \text{ gpv } \checkmark$
and $I: I \ s$
shows $\text{set-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s) \subseteq \{(x, s'). \text{count } s' \leq n + \text{count } s\}$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-exec-gpv-count:*

fixes count
assumes $\text{bound: interaction-bounded-by consider gpv } n$
and $xs': (x, s') \in \text{set-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s)$
and $\text{count: } \bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; \text{consider } x; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{eSuc } (\text{count } s)$
and $\text{ignore: } \bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and $WT: \mathcal{I} \vdash g \text{ gpv } \checkmark$
and $I: I \ s$
shows $\text{count } s' \leq n + \text{count } s$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by'-exec-gpv-count:*

fixes count
assumes $\text{bound: interaction-bounded-by' consider gpv } n$
and $xs': (x, s') \in \text{set-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s)$
and $\text{count: } \bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; \text{consider } x; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{Suc } (\text{count } s)$
and $\text{ignore: } \bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and $\text{outs: } \mathcal{I} \vdash g \text{ gpv } \checkmark$
and $I: I \ s$
shows $\text{count } s' \leq n + \text{count } s$
 $\langle \text{proof} \rangle$

lemma *pred-spmf-calleeI:* $\llbracket I \ s; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{pred-spmf } (\text{pred-prod } (\lambda \cdot \text{True}) \ I) \ (\text{callee } s \ x)$
 $\langle \text{proof} \rangle$

lemma *lossless-exec-gpv:*

assumes $\text{gpv: lossless-gpv } \mathcal{I} \ \text{gpv}$
and $\text{callee: } \bigwedge s \ \text{out}. \llbracket \text{out} \in \text{outs-}\mathcal{I} \mathcal{I}; I \ s \rrbracket \implies \text{lossless-spmf } (\text{callee } s \ \text{out})$
and $WT\text{-gpv: } \mathcal{I} \vdash g \text{ gpv } \checkmark$
and $I: I \ s$
shows $\text{lossless-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s)$
 $\langle \text{proof} \rangle$

lemma *in-set-spmf-exec-gpv-into-results-gpv:*

assumes $*$: $(x, s') \in \text{set-spmf } (\text{exec-gpv } \text{callee } \text{gpv } s)$
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*:

callee-invariant-on = (λ callee $I \mathcal{I}$.

$(\forall s \in \text{Collect } I. \forall x \in \text{outs-}\mathcal{I} \mathcal{I}. \forall (y, s') \in \text{set-spmf } (\text{callee } s x). I s') \wedge$

$(\forall s \in \text{Collect } I. \mathcal{I} \vdash c \text{ callee } s \checkmark)$)

\langle proof \rangle

lemma *callee-invariant-on-parametric* [transfer-rule]: **includes** *lifting-syntax*

assumes [transfer-rule]: *bi-unique* R *bi-total* S

shows $((S \text{====>} C \text{====>} \text{rel-spmf } (\text{rel-prod } R S)) \text{====>} (S \text{====>} (=))$
 $\text{====>} \text{rel-}\mathcal{I} C R \text{====>} (=))$

callee-invariant-on callee-invariant-on

\langle proof \rangle

lemma *callee-invariant-on-cong*:

$\llbracket I = I'; \text{outs-}\mathcal{I} \mathcal{I} = \text{outs-}\mathcal{I} \mathcal{I}' \rrbracket$

$\wedge s x. \llbracket I' s; x \in \text{outs-}\mathcal{I} \mathcal{I}' \rrbracket \implies \text{set-spmf } (\text{callee } s x) \subseteq \text{responses-}\mathcal{I} \mathcal{I} x \times$
 $\text{Collect } I' \longleftrightarrow \text{set-spmf } (\text{callee}' s x) \subseteq \text{responses-}\mathcal{I} \mathcal{I}' x \times \text{Collect } I' \rrbracket$

$\implies \text{callee-invariant-on callee } I \mathcal{I} = \text{callee-invariant-on callee}' I' \mathcal{I}'$

\langle proof \rangle

abbreviation *callee-invariant* :: $(s \Rightarrow a \Rightarrow (b \times s) \text{ spmf}) \Rightarrow (s \Rightarrow \text{bool}) \Rightarrow$
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 proof \rangle

lemma *callee-invariant-on-return-spmf* [simp]:

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 proof \rangle

lemma *callee-invariant-return-spmf* [simp]:

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 proof \rangle

lemma *callee-invariant-restrict-relp*:

includes *lifting-syntax*

assumes $(S \text{====>} C \text{====>} \text{rel-spmf } (\text{rel-prod } R S))$ *callee1 callee2*

and *callee-invariant callee1* $I1$

and *callee-invariant callee2* $I2$

shows $((S \upharpoonright I1 \otimes I2) \text{====>} C \text{====>} \text{rel-spmf } (\text{rel-prod } R (S \upharpoonright I1 \otimes I2)))$
callee1 callee2

\langle proof \rangle

lemma *callee-invariant-on-True* [simp]: *callee-invariant-on callee* $(\lambda\cdot$. *True*) \mathcal{I}

$\longleftrightarrow (\forall s. \mathcal{I} \vdash c \text{ callee } s \checkmark)$
 $\langle \text{proof} \rangle$

lemma *lossless-exec-gpv*:

$\llbracket \text{lossless-gpv } \mathcal{I} \text{ gpv}; \bigwedge s \text{ out}. \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \implies \text{lossless-spmf } (\text{callee } s \text{ out});$
 $\mathcal{I} \vdash g \text{ gpv } \checkmark; \bigwedge s. \mathcal{I} \vdash c \text{ callee } s \checkmark \rrbracket$
 $\implies \text{lossless-spmf } (\text{exec-gpv callee gpv } s)$
 $\langle \text{proof} \rangle$

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

primcorec *restrict-gpv* $:: ('a, 'out, 'in) \text{ gpv} \Rightarrow ('a, 'out, 'in) \text{ gpv}$

where

$\text{restrict-gpv gpv} = \text{GPV } ($
 $\text{map-pmf } (\text{case-option None } (\text{case-generat } (\text{Some } \circ \text{Pure})$
 $(\lambda \text{out } c. \text{if } \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I} \text{ then Some } (\text{IO out } (\lambda \text{input}. \text{if } \text{input} \in \text{responses-}\mathcal{I}$
 $\mathcal{I} \text{ out then restrict-gpv } (c \text{ input}) \text{ else Fail}))$
 $\text{else None}))$
 $(\text{the-gpv gpv}))$

lemma *restrict-gpv-Done* [*simp*]: $\text{restrict-gpv } (\text{Done } x) = \text{Done } x$
 $\langle \text{proof} \rangle$

lemma *restrict-gpv-Fail* [*simp*]: $\text{restrict-gpv } \text{Fail} = \text{Fail}$
 $\langle \text{proof} \rangle$

lemma *restrict-gpv-Pause* [*simp*]: $\text{restrict-gpv } (\text{Pause out } c) = (\text{if } \text{out} \in \text{outs-}\mathcal{I} \ \mathcal{I}$
 $\text{then Pause out } (\lambda \text{input}. \text{if } \text{input} \in \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out then restrict-gpv } (c \text{ input})$
 $\text{else Fail}) \text{ else Fail})$
 $\langle \text{proof} \rangle$

lemma *restrict-gpv-bind* [*simp*]: $\text{restrict-gpv } (\text{bind-gpv gpv } f) = \text{bind-gpv } (\text{restrict-gpv}$
 $\text{gpv}) (\lambda x. \text{restrict-gpv } (f \ x))$
 $\langle \text{proof} \rangle$

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 $\text{exec-gpv callee } (\text{restrict-gpv gpv}) \ s = \text{exec-gpv callee gpv } \ s$
 $\langle \text{proof} \rangle$

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

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

lemma *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 *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 *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 *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 *gen-colossless-restrict-gpv [simp]*:
 $\mathcal{I} \vdash_g \text{ gpv } \checkmark \implies \text{gen-lossless-gpv } b \mathcal{I} (\text{restrict-gpv } gpv) \longleftrightarrow \text{gen-lossless-gpv } b \mathcal{I}$
 gpv
 ⟨proof⟩

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

lemma *interaction-bounded-by-restrict-gpvI [interaction-bound, simp]*:
 $\text{interaction-bounded-by consider } gpv \ n \implies \text{interaction-bounded-by consider } (\text{restrict-gpv}$
 $gpv) \ n$
 ⟨proof⟩

end

lemma *restrict-gpv-parametric'*:
includes *lifting-syntax*
notes $[\text{transfer-rule}] = \text{the-gpv-parametric' Fail-parametric' corec-gpv-parametric'}$
assumes $[\text{transfer-rule}]$: *bi-unique C 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 *restrict-gpv-parametric [transfer-rule]*: **includes** *lifting-syntax* **shows**
 $\text{bi-unique } C \implies (\text{rel-}\mathcal{I} \ C \ (=) \implies \text{rel-gpv } A \ C \implies \text{rel-gpv } A \ C) \ \text{restrict-gpv}$
 restrict-gpv
 ⟨proof⟩

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

lemma (**in** *callee-invariant-on*) *exec-gpv-restrict-gpv-invariant*:
assumes $\mathcal{I} \vdash g \text{ gpv } \surd$ **and** $I \text{ s}$
shows $\text{exec-gpv } \text{callee } (\text{restrict-gpv } \mathcal{I} \text{ gpv}) \text{ s} = \text{exec-gpv } \text{callee } \text{gpv } \text{s}$
 $\langle \text{proof} \rangle$

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

inductive *finite-gpv* $:: ('a, 'out, 'in) \text{ gpv} \Rightarrow \text{bool}$
where

finite-gpvI:
 $(\bigwedge \text{out } c \text{ input. } \llbracket \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \text{ } \mathcal{I} \text{ out} \rrbracket$
 $\Longrightarrow \text{finite-gpv } (c \text{ input})) \Longrightarrow \text{finite-gpv } \text{gpv}$

lemmas *finite-gpv-induct*[*consumes 1, case-names finite-gpv, induct pred*] = *finite-gpv.induct*

lemma *finite-gpvD*: $\llbracket \text{finite-gpv } \text{gpv}; \text{IO out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \text{ } \mathcal{I} \text{ out} \rrbracket \Longrightarrow \text{finite-gpv } (c \text{ input})$
 $\langle \text{proof} \rangle$

lemma *finite-gpv-Fail* [*simp*]: *finite-gpv Fail*
 $\langle \text{proof} \rangle$

lemma *finite-gpv-Done* [*simp*]: *finite-gpv (Done x)*
 $\langle \text{proof} \rangle$

lemma *finite-gpv-Pause* [*simp*]: *finite-gpv (Pause x c) \longleftrightarrow $(\forall \text{input} \in \text{responses-}\mathcal{I} \text{ } \mathcal{I} \text{ x. } \text{finite-gpv } (c \text{ input}))$*
 $\langle \text{proof} \rangle$

lemma *finite-gpv-lift-spmf* [*simp*]: *finite-gpv (lift-spmf p)*
 $\langle \text{proof} \rangle$

lemma *finite-gpv-bind* [*simp*]:
 $\text{finite-gpv } (\text{gpv} \gg= f) \longleftrightarrow \text{finite-gpv } \text{gpv} \wedge (\forall x \in \text{results-gpv } \mathcal{I} \text{ gpv. } \text{finite-gpv } (f \text{ x}))$
(is *?lhs = ?rhs*
 $\langle \text{proof} \rangle$

end

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

lemma *finite-gpv-rel''D1*:

assumes $rel\text{-}gpv'' A C R gpv gpv'$ **and** $finite\text{-}gpv \mathcal{I} gpv$ **and** $\mathcal{I}: rel\text{-}\mathcal{I} C R \mathcal{I} \mathcal{I}'$
shows $finite\text{-}gpv \mathcal{I}' gpv'$
 $\langle proof \rangle$

lemma $finite\text{-}gpv\text{-}relD1$: $\llbracket rel\text{-}gpv A C gpv gpv'; finite\text{-}gpv \mathcal{I} gpv; rel\text{-}\mathcal{I} C (=) \mathcal{I} \mathcal{I} \rrbracket \implies finite\text{-}gpv \mathcal{I} gpv'$
 $\langle proof \rangle$

lemma $finite\text{-}gpv\text{-}rel''D2$: $\llbracket rel\text{-}gpv'' A C R gpv gpv'; finite\text{-}gpv \mathcal{I} gpv'; rel\text{-}\mathcal{I} C R \mathcal{I}' \mathcal{I} \rrbracket \implies finite\text{-}gpv \mathcal{I}' gpv$
 $\langle proof \rangle$

lemma $finite\text{-}gpv\text{-}relD2$: $\llbracket rel\text{-}gpv A C gpv gpv'; finite\text{-}gpv \mathcal{I} gpv'; rel\text{-}\mathcal{I} C (=) \mathcal{I} \mathcal{I} \rrbracket \implies finite\text{-}gpv \mathcal{I} gpv$
 $\langle proof \rangle$

lemma $finite\text{-}gpv\text{-}parametric'$: $(rel\text{-}\mathcal{I} C R \implies rel\text{-}gpv'' A C R \implies (=))$
 $finite\text{-}gpv \text{ finite-gpv}$
 $\langle proof \rangle$

lemma $finite\text{-}gpv\text{-}parametric$ [transfer-rule]: $(rel\text{-}\mathcal{I} C (=) \implies rel\text{-}gpv A C \implies (=))$
 $finite\text{-}gpv \text{ finite-gpv}$
 $\langle proof \rangle$

end

lemma $finite\text{-}gpv\text{-}map$ [simp]: $finite\text{-}gpv \mathcal{I} (map\text{-}gpv f id gpv) = finite\text{-}gpv \mathcal{I} gpv$
 $\langle proof \rangle$

lemma $finite\text{-}gpv\text{-}assert$ [simp]: $finite\text{-}gpv \mathcal{I} (assert\text{-}gpv b)$
 $\langle proof \rangle$

lemma $finite\text{-}gpv\text{-}try$ [simp]:
 $finite\text{-}gpv \mathcal{I} (TRY gpv ELSE gpv') \longleftrightarrow finite\text{-}gpv \mathcal{I} gpv \wedge (colossless\text{-}gpv \mathcal{I} gpv \vee finite\text{-}gpv \mathcal{I} gpv')$
 $(is ?lhs = -)$
 $\langle proof \rangle$

lemma $lossless\text{-}gpv\text{-}conv\text{-}finite$:
 $lossless\text{-}gpv \mathcal{I} gpv \longleftrightarrow finite\text{-}gpv \mathcal{I} gpv \wedge colossless\text{-}gpv \mathcal{I} gpv$
 $(is ?loss \longleftrightarrow ?fin \wedge ?co)$
 $\langle proof \rangle$

lemma $colossless\text{-}gpv\text{-}try$ [simp]:
 $colossless\text{-}gpv \mathcal{I} (TRY gpv ELSE gpv') \longleftrightarrow colossless\text{-}gpv \mathcal{I} gpv \vee colossless\text{-}gpv \mathcal{I} gpv'$
 $(is ?lhs \longleftrightarrow ?gpv \vee ?gpv')$
 $\langle proof \rangle$

lemma *lossless-gpv-try* [simp]:
 $\text{lossless-gpv } \mathcal{I} \text{ (TRY } gpv \text{ ELSE } gpv') \longleftrightarrow$
 $\text{finite-gpv } \mathcal{I} \text{ } gpv \wedge (\text{lossless-gpv } \mathcal{I} \text{ } gpv \vee \text{lossless-gpv } \mathcal{I} \text{ } gpv')$
 ⟨proof⟩

lemma *interaction-any-bounded-by-imp-finite*:
assumes *interaction-any-bounded-by* gpv (enat n)
shows *finite-gpv* \mathcal{I} -full gpv
 ⟨proof⟩

lemma *finite-restrict-gpvI* [simp]: $\text{finite-gpv } \mathcal{I}' \text{ } gpv \implies \text{finite-gpv } \mathcal{I}' \text{ (restrict-gpv } \mathcal{I} \text{ } gpv)$
 ⟨proof⟩

lemma *interaction-bounded-by-exec-gpv-bad-count*:
fixes *count* **and** *bad* **and** $n :: \text{enat}$ **and** $k :: \text{real}$
assumes *bound*: *interaction-bounded-by* *consider* gpv n
and *good*: $\neg \text{bad } s$
and *count*: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf (callee } s \ x); \text{ consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{Suc (count } s)$
and *ignore*: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf (callee } s \ x); \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and *bad*: $\bigwedge s' \ x. \llbracket \neg \text{bad } s'; \text{count } s' < n + \text{count } s; \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{spmf (map-spmf (bad } \circ \text{snd) (callee } s' \ x)) \ \text{True} \leq k$
and *consider*: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf (callee } s \ x); \neg \text{bad } s; \text{bad } s'; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{consider } x$
and *k-nonneg*: $k \geq 0$
and *WT-gpv*: $\mathcal{I} \vdash_g gpv \checkmark$
and *WT-callee*: $\bigwedge s. \mathcal{I} \vdash_c \text{callee } s \checkmark$
shows $\text{spmf (map-spmf (bad } \circ \text{snd) (exec-gpv callee } gpv \ s)) \ \text{True} \leq \text{ennreal } k * n$
 ⟨proof⟩

context *callee-invariant-on* **begin**

lemma *interaction-bounded-by-exec-gpv-bad-count*:
includes *lifting-syntax*
fixes *count* **and** *bad* **and** $n :: \text{enat}$
assumes *bound*: *interaction-bounded-by* *consider* gpv n
and $I: I \ s$
and *good*: $\neg \text{bad } s$
and *count*: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf (callee } s \ x); I \ s; \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{Suc (count } s)$
and *ignore*: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf (callee } s \ x); I \ s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and *bad*: $\bigwedge s' \ x. \llbracket I \ s'; \neg \text{bad } s'; \text{count } s' < n + \text{count } s; \text{consider } x; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{spmf (map-spmf (bad } \circ \text{snd) (callee } s' \ x)) \ \text{True} \leq k$
and *consider*: $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf (callee } s \ x); I \ s; \neg \text{bad } s; \text{bad } s'; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \implies \text{consider } x$

and $k\text{-nonneg}$: $k \geq 0$
and $WT\text{-gpv}$: $\mathcal{I} \vdash g \text{ gpv } \checkmark$
shows $\text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{exec-gpv } \text{callee } \text{gpv } s)) \text{ True} \leq \text{ennreal } k * n$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by'-exec-gpv-bad-count*:

fixes count **and** bad **and** $n :: \text{nat}$
assumes bound : *interaction-bounded-by'* $\text{consider } \text{gpv } n$
and I : $I \ s$
and good : $\neg \text{bad } s$
and count : $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; \text{consider } x; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{Suc } (\text{count } s)$
and ignore : $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; \neg \text{consider } x; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{count } s' \leq \text{count } s$
and bad : $\bigwedge s' \ x. \llbracket I \ s'; \neg \text{bad } s'; \text{count } s' < n + \text{count } s; \text{consider } x; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{callee } s' \ x)) \text{ True} \leq k$
and consider : $\bigwedge s \ x \ y \ s'. \llbracket (y, s') \in \text{set-spmf } (\text{callee } s \ x); I \ s; \neg \text{bad } s; \text{bad } s'; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{consider } x$
and $k\text{-nonneg}$: $k \geq 0$
and $WT\text{-gpv}$: $\mathcal{I} \vdash g \text{ gpv } \checkmark$
shows $\text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{exec-gpv } \text{callee } \text{gpv } s)) \text{ True} \leq k * n$
 $\langle \text{proof} \rangle$

lemma *interaction-bounded-by-exec-gpv-bad*:

assumes $\text{interaction-any-bounded-by}$ $\text{gpv } n$
and $I \ s \neg \text{bad } s$
and bad : $\bigwedge s \ x. \llbracket I \ s; \neg \text{bad } s; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \implies \text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{callee } s \ x)) \text{ True} \leq k$
and $k\text{-nonneg}$: $0 \leq k$
and $WT\text{-gpv}$: $\mathcal{I} \vdash g \text{ gpv } \checkmark$
shows $\text{spmf } (\text{map-spmf } (\text{bad} \circ \text{snd}) (\text{exec-gpv } \text{callee } \text{gpv } s)) \text{ True} \leq k * n$
 $\langle \text{proof} \rangle$

end

end

4.18 Expectation transformer semantics

theory *GPV-Expectation* **imports**

Generative-Probabilistic-Value

begin

lemma *le-enn2realI*: $\llbracket \text{ennreal } x \leq y; y = \top \implies x \leq 0 \rrbracket \implies x \leq \text{enn2real } y$
 $\langle \text{proof} \rangle$

lemma *enn2real-leD*: $\llbracket \text{enn2real } x < y; x \neq \top \rrbracket \implies x < \text{ennreal } y$
 $\langle \text{proof} \rangle$

lemma *ennreal-mult-le-self2I*: $\llbracket y > 0 \implies x \leq 1 \rrbracket \implies x * y \leq y$ **for** $x\ y :: \text{ennreal}$
 <proof>

lemma *ennreal-leI*: $x \leq \text{enn2real } y \implies \text{ennreal } x \leq y$
 <proof>

lemma *enn2real-INF*: $\llbracket A \neq \{\}; \forall x \in A. f\ x < \top \rrbracket \implies \text{enn2real } (\text{INF } x:A. f\ x) =$
 $(\text{INF } x:A. \text{enn2real } (f\ x))$
 <proof>

lemma *monotone-times-ennreal1*: *monotone* $(\leq) (\leq) (\lambda x. x * y :: \text{ennreal})$
 <proof>

lemma *monotone-times-ennreal2*: *monotone* $(\leq) (\leq) (\lambda x. y * x :: \text{ennreal})$
 <proof>

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

lemma *mcont-times-ennreal1*: *mcont* *Sup* $(\leq) \text{Sup } (\leq) (\lambda y. x * y :: \text{ennreal})$
 <proof>

lemma *mcont-times-ennreal2*: *mcont* *Sup* $(\leq) \text{Sup } (\leq) (\lambda y. y * x :: \text{ennreal})$
 <proof>

lemma *mcont2mcont-times-ennreal* [*cont-intro*, *simp*]:
 $\llbracket \text{mcont } \text{lub } \text{ord } \text{Sup } (\leq) (\lambda x. f\ x);$
 $\text{mcont } \text{lub } \text{ord } \text{Sup } (\leq) (\lambda x. g\ x) \rrbracket$
 $\implies \text{mcont } \text{lub } \text{ord } \text{Sup } (\leq) (\lambda x. f\ x * g\ x :: \text{ennreal})$
 <proof>

lemma *ereal-INF-cmult*: $0 < c \implies (\text{INF } i:I. c * f\ i) = \text{ereal } c * (\text{INF } i:I. f\ i)$
 <proof>

lemma *ereal-INF-multc*: $0 < c \implies (\text{INF } i:I. f\ i * c) = (\text{INF } i:I. f\ i) * \text{ereal } c$
 <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 $c * (\text{INF } i:I. f\ i) = (\text{INF } i:I. c * f\ i :: \text{ennreal})$
 <proof> **including** *ennreal.lifting*
 <proof>

lemma *pmf-map-spmf-None*: *pmf* $(\text{map-spmf } f\ p) \text{None} = \text{pmf } p \text{None}$

<proof>

lemma *nn-integral-try-spmf*:

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

<proof>

lemma *INF-UNION*: $(INF\ z : \bigcup x \in A. B\ x. f\ z) = (INF\ x:A. INF\ z:B\ x. f\ z)$ **for**
 $f :: - \Rightarrow 'b::complete\ lattice$

<proof>

definition *nn-integral-spmf* :: $'a\ spmf \Rightarrow ('a \Rightarrow ennreal) \Rightarrow ennreal$ **where**
 $nn\text{-integral-spmf}\ p = nn\text{-integral (measure-spmf}\ p)$

lemma *nn-integral-spmf-parametric* [*transfer-rule*]:

includes *lifting-syntax*

shows $(rel\text{-spmf}\ A\ ==>\ (A\ ==>\ (=))\ ==>\ (=))\ nn\text{-integral-spmf}\ nn\text{-integral-spmf}$
<proof>

lemma *weight-spmf-mcont2mcont* [*THEN lfp.mcont2mcont, cont-intro*]:

shows $weight\text{-spmf-mcont: mcont (lub-spmf) (ord-spmf (=))\ Sup\ (\le)\ (\lambda p. en\text{-}$
 $nreal (weight\text{-spmf}\ p))$

<proof>

lemma *mono2mono-nn-integral-spmf* [*THEN lfp.mono2mono, cont-intro*]:

shows $monotone\text{-nn-integral-spmf: monotone (ord-spmf (=))\ (\le)\ (\lambda p. integral^N$
 $(measure-spmf\ p)\ f)$

<proof>

lemma *cont-nn-integral-spmf*:

$cont\ lub\text{-spmf (ord-spmf (=))\ Sup (\le)\ (\lambda p :: 'a\ spmf. nn-integral (measure-spmf}$
 $p)\ f)$

<proof>

lemma *mcont2mcont-nn-integral-spmf* [*THEN lfp.mcont2mcont, cont-intro*]:

shows *mcont-nn-integral-spmf*:

$mcont\ lub\text{-spmf (ord-spmf (=))\ Sup (\le)\ (\lambda p :: 'a\ spmf. nn-integral (measure-spmf}$
 $p)\ f)$

<proof>

lemma *nn-integral-mono2mono*:

assumes $\bigwedge x. x \in space\ M \implies monotone\ ord\ (\le)\ (\lambda f. F\ f\ x)$

shows $monotone\ ord\ (\le)\ (\lambda f. nn\text{-integral}\ M\ (F\ f))$

<proof>

lemma *nn-integral-mono-lfp* [*partial-function-mono*]:

— `Partial_Function.mono_tac` does not like conditional assumptions (more

precisely the case splitter)

$(\bigwedge x. \text{ lfp.mono-body } (\lambda f. F f x)) \implies \text{ lfp.mono-body } (\lambda f. \text{ nn-integral } M (F f))$
 $\langle \text{proof} \rangle$

lemma *INF-mono-lfp* [*partial-function-mono*]:

$(\bigwedge x. \text{ lfp.mono-body } (\lambda f. F f x)) \implies \text{ lfp.mono-body } (\lambda f. \text{ INF } x:M. F f x)$
 $\langle \text{proof} \rangle$

lemmas *parallel-fixp-induct-1-2* = *parallel-fixp-induct-uc*[

of - - - $\lambda x. x - \lambda x. x$ *case-prod* - *curry*,

where $P = \lambda f g. P f (curry g)$,

unfolded case-prod-curry *curry-case-prod* *curry-K*,

OF - - - - - *refl* *refl*]

for P

lemma *monotone-ennreal-add1*: *monotone* (\leq) (\leq) $(\lambda x. x + y :: \text{ ennreal})$

$\langle \text{proof} \rangle$

lemma *monotone-ennreal-add2*: *monotone* (\leq) (\leq) $(\lambda y. x + y :: \text{ ennreal})$

$\langle \text{proof} \rangle$

lemma *mono2mono-ennreal-add*[*THEN* *lfp.mono2mono2*, *cont-intro*, *simp*]:

shows *monotone-eadd*: *monotone* $(\text{rel-prod } (\leq) (\leq)) (\leq) (\lambda(x, y). x + y :: \text{ ennreal})$

$\langle \text{proof} \rangle$

lemma *ennreal-add-partial-function-mono* [*partial-function-mono*]:

$\llbracket \text{ monotone } (\text{fun-ord } (\leq)) (\leq) f; \text{ monotone } (\text{fun-ord } (\leq)) (\leq) g \rrbracket$

$\implies \text{ monotone } (\text{fun-ord } (\leq)) (\leq) (\lambda x. f x + g x :: \text{ ennreal})$

$\langle \text{proof} \rangle$

context

fixes *fail* :: *ennreal*

and $\mathcal{I} :: ('out, 'ret) \mathcal{I}$

and $f :: 'a \Rightarrow \text{ ennreal}$

notes $\llbracket \text{function-internals} \rrbracket$

begin

partial-function (*lfp-strong*) *expectation-gpv* :: $('a, 'out, 'ret) \text{ gpv} \Rightarrow \text{ ennreal}$

where

expectation-gpv *gpv* =

$(\int^+ \text{ generat. } (\text{case generat of Pure } x \Rightarrow f x$

$| \text{ IO out } c \Rightarrow \text{ INF } r:\text{responses-}\mathcal{I} \ \mathcal{I} \ \text{out. } \text{ expectation-gpv } (c \ r))$

$\partial \text{measure-spmf } (\text{the-gpv } \text{gpv}))$

$+ \text{ fail } * \text{ pmf } (\text{the-gpv } \text{gpv}) \ \text{None}$

lemma *expectation-gpv-fixp-induct* [*case-names adm bottom step*]:

assumes *lfp.admissible* P

and $P (\lambda-. 0)$

and $\bigwedge \text{expectation-gpv}' . \llbracket \bigwedge \text{gpv} . \text{expectation-gpv}' \text{ gpv} \leq \text{expectation-gpv} \text{ gpv} ; P \text{ expectation-gpv}' \rrbracket \implies$
 $P (\lambda \text{gpv} . (\int^+ \text{generat} . (\text{case generat of Pure } x \Rightarrow f x \mid \text{IO out } c \Rightarrow \text{INF } r : \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} . \text{expectation-gpv}' (c \ r)) \ \partial \text{measure-spmf} (\text{the-gpv} \ \text{gpv})) + \text{fail} * \text{pmf} (\text{the-gpv} \ \text{gpv}) \ \text{None})$
shows $P \text{ expectation-gpv}$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-Done* [simp]: $\text{expectation-gpv} (\text{Done } x) = f x$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-Fail* [simp]: $\text{expectation-gpv} \text{Fail} = \text{fail}$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-lift-spmf* [simp]:
 $\text{expectation-gpv} (\text{lift-spmf } p) = (\int^+ x . f x \ \partial \text{measure-spmf } p) + \text{fail} * \text{pmf } p \ \text{None}$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-Pause* [simp]:
 $\text{expectation-gpv} (\text{Pause out } c) = (\text{INF } r : \text{responses-}\mathcal{I} \ \mathcal{I} \text{ out} . \text{expectation-gpv} (c \ r))$
 $\langle \text{proof} \rangle$

end

context begin

private definition *weight-spmf'* $p = \text{weight-spmf } p$

lemmas *weight-spmf'-parametric* = *weight-spmf-parametric*[folded *weight-spmf'-def*]

lemma *expectation-gpv-parametric'*:

includes *lifting-syntax* **notes** *weight-spmf'-parametric*[*transfer-rule*]

shows $((=) \implies \text{rel-}\mathcal{I} \ C \ R \implies (A \implies (=)) \implies \text{rel-gpv}'' \ A \ C \ R \implies (=)) \ \text{expectation-gpv} \ \text{expectation-gpv}$
 $\langle \text{proof} \rangle$

end

lemma *expectation-gpv-parametric* [*transfer-rule*]:

includes *lifting-syntax*

shows $((=) \implies \text{rel-}\mathcal{I} \ C \ (=) \implies (A \implies (=)) \implies \text{rel-gpv} \ A \ C \implies (=)) \ \text{expectation-gpv} \ \text{expectation-gpv}$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-cong*:

fixes *fail fail'*

assumes *fail*: $\text{fail} = \text{fail}'$

and $\mathcal{I} : \mathcal{I} = \mathcal{I}'$

and *gpv*: $\text{gpv} = \text{gpv}'$

and *f*: $\bigwedge x . x \in \text{results-gpv} \ \mathcal{I}' \ \text{gpv}' \implies f x = g x$

shows $\text{expectation-gpv} \ \text{fail} \ \mathcal{I} \ f \ \text{gpv} = \text{expectation-gpv} \ \text{fail}' \ \mathcal{I}' \ g \ \text{gpv}'$
 $\langle \text{proof} \rangle$

lemma *expectation-gpv-cong-fail*:

colossless-gpv \mathcal{I} *gpv* \implies *expectation-gpv fail* \mathcal{I} *f gpv* = *expectation-gpv fail'* \mathcal{I} *f gpv* **for** *fail*
(*proof*)

lemma *expectation-gpv-mono*:

fixes *fail fail'*
assumes *fail*: *fail* \leq *fail'*
and *fg*: *f* \leq *g*
shows *expectation-gpv fail* \mathcal{I} *f gpv* \leq *expectation-gpv fail'* \mathcal{I} *g gpv*
(*proof*)

lemma *expectation-gpv-mono-strong*:

fixes *fail fail'*
assumes *fail*: \neg *colossless-gpv* \mathcal{I} *gpv* \implies *fail* \leq *fail'*
and *fg*: $\bigwedge x. x \in$ *results-gpv* \mathcal{I} *gpv* \implies *f x* \leq *g x*
shows *expectation-gpv fail* \mathcal{I} *f gpv* \leq *expectation-gpv fail'* \mathcal{I} *g gpv*
(*proof*)

lemma *expectation-gpv-bind* [*simp*]:

fixes \mathcal{I} *f g fail*
defines *expectation-gpv1* \equiv *expectation-gpv fail* \mathcal{I} *f*
and *expectation-gpv2* \equiv *expectation-gpv fail* \mathcal{I} (*expectation-gpv fail* \mathcal{I} *f* \circ *g*)
shows *expectation-gpv1* (*bind-gpv gpv g*) = *expectation-gpv2 gpv* (**is** ?*lhs* = ?*rhs*)
(*proof*)

lemma *expectation-gpv-try-gpv* [*simp*]:

fixes *fail* \mathcal{I} *f gpv'*
defines *expectation-gpv1* \equiv *expectation-gpv fail* \mathcal{I} *f*
and *expectation-gpv2* \equiv *expectation-gpv* (*expectation-gpv fail* \mathcal{I} *f gpv'*) \mathcal{I} *f*
shows *expectation-gpv1* (*try-gpv gpv gpv'*) = *expectation-gpv2 gpv*
(*proof*)

lemma *expectation-gpv-restrict-gpv*:

$\mathcal{I} \vdash_g$ *gpv* $\checkmark \implies$ *expectation-gpv fail* \mathcal{I} *f* (*restrict-gpv* \mathcal{I} *gpv*) = *expectation-gpv fail* \mathcal{I} *f gpv* **for** *fail*
(*proof*)

lemma *expectation-gpv-const-le*: $\mathcal{I} \vdash_g$ *gpv* $\checkmark \implies$ *expectation-gpv fail* \mathcal{I} ($\lambda-. c$) *gpv* \leq *max c fail* **for** *fail*
(*proof*)

lemma *expectation-gpv-no-results*:

\llbracket *results-gpv* \mathcal{I} *gpv* = $\{\}$; $\mathcal{I} \vdash_g$ *gpv* $\checkmark \rrbracket \implies$ *expectation-gpv 0* \mathcal{I} *f gpv* = 0
(*proof*)

lemma *expectation-gpv-cmult*:

fixes *fail*
assumes 0 < *c* **and** *c* \neq \top

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

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 $\text{expectation-gpv } 0 \mathcal{I} f \text{ gpv} \leq \int^+ (x, s). f x \partial \text{measure-spmf } (\text{exec-gpv callee gpv } s)$
 $\langle \text{proof} \rangle$

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]*: $\text{weight-gpv } \mathcal{I} (\text{Done } x) = 1$
 $\langle \text{proof} \rangle$

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

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

lemma *weight-gpv-Pause [simp]*:

$(\bigwedge r. r \in \text{responses-}\mathcal{I} \mathcal{I} \text{out} \implies \mathcal{I} \vdash_g c r \checkmark)$
 $\implies \text{weight-gpv } \mathcal{I} (\text{Pause out } c) = (\text{if out} \in \text{outs-}\mathcal{I} \mathcal{I} \text{ then } \text{INF } r:\text{responses-}\mathcal{I} \mathcal{I} \text{out. weight-gpv } \mathcal{I} (c r) \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *weight-gpv-nonneg*: $0 \leq \text{weight-gpv } \mathcal{I} \text{gpv}$
 $\langle \text{proof} \rangle$

lemma *weight-gpv-le-1*: $\mathcal{I} \vdash_g \text{gpv } \checkmark \implies \text{weight-gpv } \mathcal{I} \text{gpv} \leq 1$
 $\langle \text{proof} \rangle$

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 $\text{weight-gpv } \mathcal{I} \text{gpv} \leq \text{weight-spmf } (\text{exec-gpv callee gpv } s)$
 $\langle \text{proof} \rangle$

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 $\text{weight-gpv } \mathcal{I} \text{gpv} \leq \text{weight-spmf } (\text{exec-gpv callee gpv } s)$
including *lifting-syntax*

<proof>

4.19 Probabilistic termination

definition $p\text{gen-lossless-gpv} :: \text{ennreal} \Rightarrow ('c, 'r) \mathcal{I} \Rightarrow ('a, 'c, 'r) \text{gpv} \Rightarrow \text{bool}$
where $p\text{gen-lossless-gpv fail } \mathcal{I} \text{ gpv} = (\text{expectation-gpv fail } \mathcal{I} (\lambda\cdot. 1) \text{ gpv} = 1)$ **for fail**

abbreviation $p\text{lossless-gpv} :: ('c, 'r) \mathcal{I} \Rightarrow ('a, 'c, 'r) \text{gpv} \Rightarrow \text{bool}$
where $p\text{lossless-gpv} \equiv p\text{gen-lossless-gpv } 0$

abbreviation $p\text{finite-gpv} :: ('c, 'r) \mathcal{I} \Rightarrow ('a, 'c, 'r) \text{gpv} \Rightarrow \text{bool}$
where $p\text{finite-gpv} \equiv p\text{gen-lossless-gpv } 1$

lemma $p\text{gen-lossless-gpvI}$ [*intro?*]: $\text{expectation-gpv fail } \mathcal{I} (\lambda\cdot. 1) \text{ gpv} = 1 \implies p\text{gen-lossless-gpv fail } \mathcal{I} \text{ gpv}$ **for fail**
<proof>

lemma $p\text{gen-lossless-gpvD}$: $p\text{gen-lossless-gpv fail } \mathcal{I} \text{ gpv} \implies \text{expectation-gpv fail } \mathcal{I} (\lambda\cdot. 1) \text{ gpv} = 1$ **for fail**
<proof>

lemma $\text{lossless-imp-plossless-gpv}$:
assumes $\text{lossless-gpv } \mathcal{I} \text{ gpv } \mathcal{I} \vdash_g \text{gpv } \checkmark$
shows $p\text{lossless-gpv } \mathcal{I} \text{ gpv}$
<proof>

lemma $\text{finite-imp-pfinite-gpv}$:
assumes $\text{finite-gpv } \mathcal{I} \text{ gpv } \mathcal{I} \vdash_g \text{gpv } \checkmark$
shows $p\text{finite-gpv } \mathcal{I} \text{ gpv}$
<proof>

lemma $p\text{lossless-gpv-lossless-spmfD}$:
assumes $\text{lossless: } p\text{lossless-gpv } \mathcal{I} \text{ gpv}$
and $WT: \mathcal{I} \vdash_g \text{gpv } \checkmark$
shows $\text{lossless-spmf } (\text{the-gpv } \text{gpv})$
<proof>

lemma
shows $p\text{lossless-gpv-ContD}$:
[[$p\text{lossless-gpv } \mathcal{I} \text{ gpv}$; $IO \text{ out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I}$
 $\text{out}; \mathcal{I} \vdash_g \text{gpv } \checkmark$]]
 $\implies p\text{lossless-gpv } \mathcal{I} (c \text{ input})$
and $p\text{finite-gpv-ContD}$:
[[$p\text{finite-gpv } \mathcal{I} \text{ gpv}$; $IO \text{ out } c \in \text{set-spmf } (\text{the-gpv } \text{gpv}); \text{input} \in \text{responses-}\mathcal{I} \mathcal{I}$
 $\text{out}; \mathcal{I} \vdash_g \text{gpv } \checkmark$]]
 $\implies p\text{finite-gpv } \mathcal{I} (c \text{ input})$
<proof>

lemma *plossless-iff-colossless-pfinite*:

assumes $WT: \mathcal{I} \vdash g \text{ gpv } \checkmark$

shows $plossless\text{-gpv } \mathcal{I} \text{ gpv} \longleftrightarrow colossless\text{-gpv } \mathcal{I} \text{ gpv} \wedge pfinite\text{-gpv } \mathcal{I} \text{ gpv}$

<proof>

lemma *pgen-lossless-gpv-Done* [simp]: $pgen\text{-lossless-gpv } fail \ \mathcal{I} \ (Done \ x) \ \mathbf{for} \ fail$

<proof>

lemma *pgen-lossless-gpv-Fail* [simp]: $pgen\text{-lossless-gpv } fail \ \mathcal{I} \ Fail \longleftrightarrow fail = 1 \ \mathbf{for} \ fail$

<proof>

lemma *pgen-lossless-gpv-PauseI* [simp, intro!]:

$\llbracket out \in outs\text{-}\mathcal{I} \ \mathcal{I}; \bigwedge r. r \in responses\text{-}\mathcal{I} \ \mathcal{I} \ out \implies pgen\text{-lossless-gpv } fail \ \mathcal{I} \ (c \ r) \rrbracket$

$\implies pgen\text{-lossless-gpv } fail \ \mathcal{I} \ (Pause \ out \ c) \ \mathbf{for} \ fail$

<proof>

lemma *pgen-lossless-gpv-bindI* [simp, intro!]:

$\llbracket pgen\text{-lossless-gpv } fail \ \mathcal{I} \ \text{gpv}; \bigwedge x. x \in results\text{-gpv } \mathcal{I} \ \text{gpv} \implies pgen\text{-lossless-gpv} \ fail \ \mathcal{I} \ (f \ x) \rrbracket$

$\implies pgen\text{-lossless-gpv } fail \ \mathcal{I} \ (bind\text{-gpv } \text{gpv} \ f) \ \mathbf{for} \ fail$

<proof>

lemma *pgen-lossless-gpv-lift-spmf* [simp]:

$pgen\text{-lossless-gpv } fail \ \mathcal{I} \ (lift\text{-spmf } p) \longleftrightarrow lossless\text{-spmf } p \vee fail = 1 \ \mathbf{for} \ fail$

<proof>

lemma *expectation-gpv-top-pfinite*:

assumes $pfinite\text{-gpv } \mathcal{I} \ \text{gpv}$

shows $expectation\text{-gpv } \top \ \mathcal{I} \ (\lambda\cdot. \top) \ \text{gpv} = \top$

<proof>

lemma *pfinite-INF-le-expectation-gpv*:

fixes $fail \ \mathcal{I} \ \text{gpv} \ f$

defines $c \equiv min \ (INF \ x:results\text{-gpv } \mathcal{I} \ \text{gpv}. \ f \ x) \ fail$

assumes $fin: pfinite\text{-gpv } \mathcal{I} \ \text{gpv}$

shows $c \leq expectation\text{-gpv } fail \ \mathcal{I} \ f \ \text{gpv} \ (\mathbf{is} \ ?lhs \leq ?rhs)$

<proof>

lemma *plossless-INF-le-expectation-gpv*:

fixes $fail$

assumes $plossless\text{-gpv } \mathcal{I} \ \text{gpv} \ \mathbf{and} \ \mathcal{I} \vdash g \ \text{gpv} \ \checkmark$

shows $(INF \ x:results\text{-gpv } \mathcal{I} \ \text{gpv}. \ f \ x) \leq expectation\text{-gpv } fail \ \mathcal{I} \ f \ \text{gpv} \ (\mathbf{is} \ ?lhs \leq ?rhs)$

<proof>

lemma *expectation-gpv-le-inline*:

fixes \mathcal{I}'

defines $expectation-gpv2 \equiv expectation-gpv\ 0\ \mathcal{I}'$
assumes $callee: \bigwedge s\ x. x \in outs\ \mathcal{I}\ \mathcal{I} \implies plossless-gpv\ \mathcal{I}'\ (callee\ s\ x)$
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 $callee': \bigwedge s\ x. x \in outs\ \mathcal{I}\ \mathcal{I} \implies set-spmf\ (callee\ s\ x) \subseteq responses\ \mathcal{I}\ \mathcal{I}\ x \times UNIV$
shows $lossless-spmf\ (exec-gpv\ callee\ gpv\ s)$
 $\langle proof \rangle$

end

theory *GPV-Bisim* **imports**

GPV-Expectation

begin

4.20 Bisimulation for oracles

Bisimulation is a consequence of parametricity

lemma *exec-gpv-oracle-bisim'*:

assumes $*: X\ s1\ s2$
and $bisim: \bigwedge s1\ s2\ x. X\ s1\ s2 \implies rel-spmf\ (\lambda(a, s1') (b, s2'). a = b \wedge X\ s1'\ s2')\ (oracle1\ s1\ x)\ (oracle2\ s2\ x)$
shows $rel-spmf\ (\lambda(a, s1') (b, s2'). a = b \wedge X\ s1'\ s2')\ (exec-gpv\ oracle1\ gpv\ s1)\ (exec-gpv\ oracle2\ gpv\ s2)$
 $\langle proof \rangle$

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*: $\bigwedge x\ s1'\ s2'.\ \llbracket X\ s1'\ s2'; (x, s1') \in \text{set-spmf } (\text{exec-gpv } \text{oracle1 } \text{gpv } s1);$
 $(x, s2') \in \text{set-spmf } (\text{exec-gpv } \text{oracle2 } \text{gpv } s2) \rrbracket \implies R\ (x, s1')\ (x, s2')$
shows $\text{rel-spmf } R\ (\text{exec-gpv } \text{oracle1 } \text{gpv } s1)\ (\text{exec-gpv } \text{oracle2 } \text{gpv } s2)$
 $\langle \text{proof} \rangle$

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 $\text{run-gpv } \text{oracle1 } \text{gpv } s1 = \text{run-gpv } \text{oracle2 } \text{gpv } s2$
 $\langle \text{proof} \rangle$

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

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

lemma *exec-until-bad-fixp-induct* [*case-names adm bottom step*]:

assumes $\text{ccpo.admissible } (\text{fun-lub } \text{lub-spmf})\ (\text{fun-ord } (\text{ord-spmf } (=)))\ (\lambda f.\ P$
 $(\lambda \text{gpv } s1\ s2.\ f\ ((\text{gpv}, s1), s2)))$
and $P\ (\lambda - -.\ \text{return-pmf } \text{None})$
and $\bigwedge \text{exec-until-bad}'.\ P\ \text{exec-until-bad}' \implies$
 $P\ (\lambda \text{gpv } s1\ s2.\ \text{if } \text{bad1 } s1 \vee \text{bad2 } s2 \text{ then } \text{pair-spmf } (\text{exec-gpv } \text{oracle1 } \text{gpv } s1)$
 $(\text{exec-gpv } \text{oracle2 } \text{gpv } s2))$
 $\text{else } \text{bind-spmf } (\text{the-gpv } \text{gpv})\ (\lambda \text{generat.}$
 $\text{case } \text{generat of Pure } x \Rightarrow \text{return-spmf } ((x, s1), (x, s2))$
 $| \text{IO out } f \Rightarrow \text{bind-spmf } (\text{joint-oracle } (s1, s2)\ \text{out})\ (\lambda((x, s1'), (y, s2')).$
 $\text{if } \text{bad1 } s1' \vee \text{bad2 } s2' \text{ then } \text{pair-spmf } (\text{exec-gpv } \text{oracle1 } (f\ x)\ s1')\ (\text{exec-gpv } \text{oracle2 } (f\ y)\ s2')$
 $\text{else } \text{exec-until-bad}'\ (f\ x)\ s1'\ s2'))$
shows $P\ \text{exec-until-bad}$

<proof>

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 $\text{bad}: \text{bad1 } s1 = \text{bad2 } s2$
and $\text{bisim}: \bigwedge s1 \ s2 \ x. \llbracket X \ s1 \ s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \Longrightarrow \text{rel-spmf } (\lambda(a, s1') (b, s2').$
 $\text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then X-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2'))$
 $(\text{oracle1 } s1 \ x) (\text{oracle2 } s2 \ x)$
and $\text{bad-sticky1}: \bigwedge s2. \text{bad2 } s2 \Longrightarrow \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge$
 $X\text{-bad } s1 \ s2) \ \mathcal{I}$
and $\text{bad-sticky2}: \bigwedge s1. \text{bad1 } s1 \Longrightarrow \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge$
 $X\text{-bad } s1 \ s2) \ \mathcal{I}$
and $\text{lossless1}: \bigwedge s1 \ x. \llbracket \text{bad1 } s1; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \Longrightarrow \text{lossless-spmf } (\text{oracle1 } s1 \ x)$
and $\text{lossless2}: \bigwedge s2 \ x. \llbracket \text{bad2 } s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \Longrightarrow \text{lossless-spmf } (\text{oracle2 } s2 \ x)$
and $\text{lossless}: \text{plossless-gpv } \mathcal{I} \ \text{gpv}$
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 $\text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then X-bad}$
 $s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2')) (\text{exec-gpv oracle1 gpv } s1) (\text{exec-gpv oracle2 gpv}$
 $s2)$
(is $\text{rel-spmf } ?R \ ?p \ ?q)$
<proof>

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 $\text{bad}: \text{bad1 } s1 = \text{bad2 } s2$
and $\text{bisim}: \bigwedge s1 \ s2 \ x. \llbracket X \ s1 \ s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \Longrightarrow \text{rel-spmf } (\lambda(a, s1') (b, s2').$
 $\text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then X-bad } s1' \ s2' \text{ else } a = b \wedge X \ s1' \ s2'))$
 $(\text{oracle1 } s1 \ x) (\text{oracle2 } s2 \ x)$
and $\text{bad-sticky1}: \bigwedge s2. \text{bad2 } s2 \Longrightarrow \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge$
 $X\text{-bad } s1 \ s2) \ \mathcal{I}$
and $\text{bad-sticky2}: \bigwedge s1. \text{bad1 } s1 \Longrightarrow \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge$
 $X\text{-bad } s1 \ s2) \ \mathcal{I}$
and $\text{lossless1}: \bigwedge s1 \ x. \llbracket \text{bad1 } s1; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \Longrightarrow \text{lossless-spmf } (\text{oracle1 } s1 \ x)$
and $\text{lossless2}: \bigwedge s2 \ x. \llbracket \text{bad2 } s2; x \in \text{outs-}\mathcal{I} \ \mathcal{I} \rrbracket \Longrightarrow \text{lossless-spmf } (\text{oracle2 } s2 \ x)$
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 $\text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then X-bad}$

$s1' s2' \text{ else } a = b \wedge X s1' s2')$ ($\text{exec-gpv oracle1 gpv s1}$) ($\text{exec-gpv oracle2 gpv s2}$)
 ⟨proof⟩

lemma *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 *: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and $\text{bad}: \text{bad1 } s1 = \text{bad2 } s2$
and $\text{bisim}: \bigwedge s1 s2 x. \llbracket X s1 s2; x \in \text{outs-}\mathcal{I} \mathcal{I}; I1 s1; I2 s2 \rrbracket \Longrightarrow \text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then } X\text{-bad } s1' s2' \text{ else } a = b \wedge X s1' s2')) (\text{oracle1 } s1 x) (\text{oracle2 } s2 x)$
and $\text{bad-sticky1}: \bigwedge s2. \llbracket \text{bad2 } s2; I2 s2 \rrbracket \Longrightarrow \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge X\text{-bad } s1 s2) \mathcal{I}$
and $\text{bad-sticky2}: \bigwedge s1. \llbracket \text{bad1 } s1; I1 s1 \rrbracket \Longrightarrow \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge X\text{-bad } s1 s2) \mathcal{I}$
and $\text{lossless1}: \bigwedge s1 x. \llbracket \text{bad1 } s1; I1 s1; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \Longrightarrow \text{lossless-spmf } (\text{oracle1 } s1 x)$
and $\text{lossless2}: \bigwedge s2 x. \llbracket \text{bad2 } s2; I2 s2; x \in \text{outs-}\mathcal{I} \mathcal{I} \rrbracket \Longrightarrow \text{lossless-spmf } (\text{oracle2 } s2 x)$
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 $\text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then } X\text{-bad } s1' s2' \text{ else } a = b \wedge X s1' s2')) (\text{exec-gpv oracle1 gpv s1}) (\text{exec-gpv oracle2 gpv s2})$
including *lifting-syntax*
 ⟨proof⟩

lemma *exec-gpv-oracle-bisim-bad*:

assumes *: *if bad2 s2 then X-bad s1 s2 else X s1 s2*
and $\text{bad}: \text{bad1 } s1 = \text{bad2 } s2$
and $\text{bisim}: \bigwedge s1 s2 x. X s1 s2 \Longrightarrow \text{rel-spmf } (\lambda(a, s1') (b, s2'). \text{bad1 } s1' = \text{bad2 } s2' \wedge (\text{if bad2 } s2' \text{ then } X\text{-bad } s1' s2' \text{ else } a = b \wedge X s1' s2')) (\text{oracle1 } s1 x) (\text{oracle2 } s2 x)$
and $\text{bad-sticky1}: \bigwedge s2. \text{bad2 } s2 \Longrightarrow \text{callee-invariant-on oracle1 } (\lambda s1. \text{bad1 } s1 \wedge X\text{-bad } s1 s2) \mathcal{I}$
and $\text{bad-sticky2}: \bigwedge s1. \text{bad1 } s1 \Longrightarrow \text{callee-invariant-on oracle2 } (\lambda s2. \text{bad2 } s2 \wedge X\text{-bad } s1 s2) \mathcal{I}$
and $\text{lossless1}: \bigwedge s1 x. \text{bad1 } s1 \Longrightarrow \text{lossless-spmf } (\text{oracle1 } s1 x)$
and $\text{lossless2}: \bigwedge s2 x. \text{bad2 } s2 \Longrightarrow \text{lossless-spmf } (\text{oracle2 } s2 x)$
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 $WT\text{-}gpv: \mathcal{I} \vdash g \text{ } gpv \checkmark$
and $R: \bigwedge a \ s1 \ b \ s2. \llbracket bad1 \ s1 = bad2 \ s2; \neg bad2 \ s2 \implies a = b \wedge X \ s1 \ s2; bad2 \ s2 \implies X\text{-}bad \ s1 \ s2 \rrbracket \implies R \ (a, \ s1) \ (b, \ s2)$
shows $rel\text{-}spmf \ R \ (exec\text{-}gpv \ oracle1 \ gpv \ s1) \ (exec\text{-}gpv \ oracle2 \ gpv \ s2)$
 $\langle proof \rangle$

lemma $exec\text{-}gpv\text{-}oracle\text{-}bisim\text{-}bad\text{-}full$:

assumes $X \ s1 \ s2$
and $bad1 \ s1 = bad2 \ s2$
and $\bigwedge s1 \ s2 \ x. X \ s1 \ s2 \implies rel\text{-}spmf \ (\lambda(a, \ s1') \ (b, \ s2')). \ bad1 \ s1' = bad2 \ s2' \wedge (\neg bad2 \ s2' \longrightarrow a = b \wedge X \ s1' \ s2')$ $(oracle1 \ s1 \ x) \ (oracle2 \ s2 \ x)$
and $callee\text{-}invariant \ oracle1 \ bad1$
and $callee\text{-}invariant \ oracle2 \ bad2$
and $\bigwedge s1 \ x. bad1 \ s1 \implies lossless\text{-}spmf \ (oracle1 \ s1 \ x)$
and $\bigwedge s2 \ x. bad2 \ s2 \implies lossless\text{-}spmf \ (oracle2 \ s2 \ x)$
and $lossless\text{-}gpv \ \mathcal{I}\text{-}full \ gpv$
and $R: \bigwedge a \ s1 \ b \ s2. \llbracket bad1 \ s1 = bad2 \ s2; \neg bad2 \ s2 \implies a = b \wedge X \ s1 \ s2 \rrbracket \implies R \ (a, \ s1) \ (b, \ s2)$
shows $rel\text{-}spmf \ R \ (exec\text{-}gpv \ oracle1 \ gpv \ s1) \ (exec\text{-}gpv \ oracle2 \ gpv \ s2)$
 $\langle proof \rangle$

lemma $max\text{-}enn2ereal$: $max \ (enn2ereal \ x) \ (enn2ereal \ y) = enn2ereal \ (max \ x \ y)$
including $ennreal.lifting \ \langle proof \rangle$

lemma $identical\text{-}until\text{-}bad$:

assumes $bad\text{-}eq: map\text{-}spmf \ bad \ p = map\text{-}spmf \ bad \ q$
and $not\text{-}bad: measure \ (measure\text{-}spmf \ (map\text{-}spmf \ (\lambda x. \ (f \ x, \ bad \ x)) \ p)) \ (A \times \{False\}) = measure \ (measure\text{-}spmf \ (map\text{-}spmf \ (\lambda x. \ (f \ x, \ bad \ x)) \ q)) \ (A \times \{False\})$
shows $|measure \ (measure\text{-}spmf \ (map\text{-}spmf \ f \ p)) \ A - measure \ (measure\text{-}spmf \ (map\text{-}spmf \ f \ q)) \ A| \leq spmf \ (map\text{-}spmf \ bad \ p) \ True$
 $\langle proof \rangle$

lemma (**in** $callee\text{-}invariant\text{-}on$) $exec\text{-}gpv\text{-}bind\text{-}materialize$:

fixes $f :: 's \Rightarrow 'r \ spmf$
and $g :: 'x \times 's \Rightarrow 'r \Rightarrow 'y \ spmf$
and $s :: 's$
defines $exec\text{-}gpv2 \equiv exec\text{-}gpv$
assumes $cond: \bigwedge s \ x \ y \ s'. \llbracket (y, \ s') \in set\text{-}spmf \ (callee \ s \ x); I \ s \rrbracket \implies f \ s = f \ s'$
and $\mathcal{I}: \mathcal{I} = \mathcal{I}\text{-}full$
shows $bind\text{-}spmf \ (exec\text{-}gpv \ callee \ gpv \ s) \ (\lambda as. \ bind\text{-}spmf \ (f \ (snd \ as)) \ (g \ as)) = exec\text{-}gpv2 \ (\lambda(r, \ s) \ x. \ bind\text{-}spmf \ (callee \ s \ x) \ (\lambda(y, \ s'). \ if \ I \ s' \wedge \ r = None \ then \ map\text{-}spmf \ (\lambda r. \ (y, \ (Some \ r, \ s')))) \ (f \ s') \ else \ return\text{-}spmf \ (y, \ (r, \ s')))) \ gpv \ (None, \ s)$
 $\ggg (\lambda(a, \ r, \ s). \ case \ r \ of \ None \Rightarrow \ bind\text{-}spmf \ (f \ s) \ (g \ (a, \ s)) \ | \ Some \ r' \Rightarrow \ g \ (a, \ s) \ r')$
(is $?lhs = ?rhs$ **is** $- = bind\text{-}spmf \ (exec\text{-}gpv2 \ ?callee2 \ - \ -) \ -)$
 $\langle proof \rangle$

primcorec $gpv\text{-}stop :: ('a, 'c, 'r) \ gpv \Rightarrow ('a \ option, 'c, 'r \ option) \ gpv$

where

$the-gpv (gpv-stop\ gpv) =$
 $map-spmf (map-generat\ Some\ id\ (\lambda rpv\ input.\ case\ input\ of\ None\ \Rightarrow\ Done\ None$
 $| Some\ input'\ \Rightarrow\ gpv-stop\ (rpv\ input'))))$
 $(the-gpv\ gpv)$

lemma $gpv-stop-Done$ [simp]: $gpv-stop (Done\ x) = Done (Some\ x)$
<proof>

lemma $gpv-stop-Fail$ [simp]: $gpv-stop\ Fail = Fail$
<proof>

lemma $gpv-stop-Pause$ [simp]: $gpv-stop (Pause\ out\ rpv) = Pause\ out\ (\lambda input.\ case$
 $input\ of\ None\ \Rightarrow\ Done\ None\ | Some\ input'\ \Rightarrow\ gpv-stop\ (rpv\ input'))$
<proof>

lemma $gpv-stop-lift-spmf$ [simp]: $gpv-stop (lift-spmf\ p) = lift-spmf (map-spmf$
 $Some\ p)$
<proof>

lemma $gpv-stop-bind$ [simp]:
 $gpv-stop (bind-gpv\ gpv\ f) = bind-gpv (gpv-stop\ gpv) (\lambda x.\ case\ x\ of\ None\ \Rightarrow\ Done$
 $None\ | Some\ x'\ \Rightarrow\ gpv-stop (f\ x'))$
<proof>

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

lemma $gpv-stop-parametric'$:
notes [transfer-rule] = $the-gpv-parametric'\ the-gpv-parametric'\ Done-parametric'$
 $corec-gpv-parametric'$
shows $(rel-gpv''\ A\ C\ R\ ==>\ rel-gpv'' (rel-option\ A)\ C\ (rel-option\ R))\ gpv-stop$
 $gpv-stop$
<proof>

lemma $gpv-stop-parametric$ [transfer-rule]:
shows $(rel-gpv\ A\ C\ ==>\ rel-gpv (rel-option\ A)\ C)\ gpv-stop\ gpv-stop$
<proof>

lemma $gpv-stop-transfer$:
 $(rel-gpv''\ A\ B\ C\ ==>\ rel-gpv'' (pcr-Some\ A)\ B\ (pcr-Some\ C)) (\lambda x.\ x)\ gpv-stop$
<proof>

end

lemma $gpv-stop-map'$ [simp]:
 $gpv-stop (map-gpv'\ f\ g\ h\ gpv) = map-gpv' (map-option\ f)\ g (map-option\ h)$
 $(gpv-stop\ gpv)$
<proof>

lemma *interaction-bound-gpv-stop* [simp]:
interaction-bound consider (gpv-stop gpv) = interaction-bound consider gpv
 ⟨proof⟩

abbreviation *exec-gpv-stop* :: ('s ⇒ 'c ⇒ ('r option × 's) spmf) ⇒ ('a, 'c, 'r)
gpv ⇒ 's ⇒ ('a option × 's) spmf
where *exec-gpv-stop callee gpv* ≡ *exec-gpv callee (gpv-stop gpv)*

abbreviation *inline-stop* :: ('s ⇒ 'c ⇒ ('r option × 's, 'c', 'r') gpv) ⇒ ('a, 'c,
 'r) gpv ⇒ 's ⇒ ('a option × 's, 'c', 'r') gpv
where *inline-stop callee gpv* ≡ *inline callee (gpv-stop gpv)*

context

fixes *joint-oracle* :: 's1 ⇒ 's2 ⇒ 'c ⇒ (('r option × 's1) option × ('r option ×
 's2) option) pmf
and *callee1* :: 's1 ⇒ 'c ⇒ ('r option × 's1) spmf
notes [[*function-internals*]]
begin

partial-function (*spmf*) *exec-until-stop* :: ('a option, 'c, 'r) gpv ⇒ 's1 ⇒ 's2 ⇒
 bool ⇒ ('a option × 's1 × 's2) spmf

where

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

end

lemma *ord-spmf-exec-gpv-stop*:

fixes *callee1* :: ('c, 'r option, 's) callee
and *callee2* :: ('c, 'r option, 's) callee
and *S* :: 's ⇒ 's ⇒ bool
and *gpv* :: ('a, 'c, 'r) gpv
assumes *bisim*:

```

     $\bigwedge s1\ s2\ x. \llbracket S\ s1\ s2; \neg\ stop\ s2 \rrbracket \implies$ 
     $ord\text{-}spmf\ (\lambda(r1, s1')\ (r2, s2').\ le\text{-}option\ r2\ r1 \wedge S\ s1'\ s2' \wedge (r2 = None \wedge r1$ 
 $\neq None \iff stop\ s2'))$ 
    (callee1 s1 x) (callee2 s2 x)
    and init: S s1 s2
    and go:  $\neg\ stop\ s2$ 
    and sticking:  $\bigwedge s1\ s2\ x\ y\ s1'. \llbracket (y, s1') \in set\text{-}spmf\ (callee1\ s1\ x); S\ s1\ s2; stop$ 
 $s2 \rrbracket \implies S\ s1'\ s2$ 
    shows  $ord\text{-}spmf\ (rel\text{-}prod\ (ord\text{-}option\ \top)^{-1-1}\ S)\ (exec\text{-}gpv\text{-}stop\ callee1\ gpv\ s1)$ 
    (exec-gpv-stop callee2 gpv s2)
    <proof>

end
theory GPV-Applicative imports
  Generative-Probabilistic-Value
  SPMF-Applicative
begin

```

4.21 Applicative instance for $(-, 'out, 'in)\ gpv$

```

definition ap-gpv :: ('a  $\Rightarrow$  'b, 'out, 'in) gpv  $\Rightarrow$  ('a, 'out, 'in) gpv  $\Rightarrow$  ('b, 'out, 'in)
gpv
where ap-gpv f x = bind-gpv f ( $\lambda f'.\ bind\text{-}gpv\ x\ (\lambda x'.\ Done\ (f'\ x'))$ )

```

```

adhoc-overloading Applicative.ap ap-gpv

```

```

abbreviation (input) pure-gpv :: 'a  $\Rightarrow$  ('a, 'out, 'in) gpv
where pure-gpv  $\equiv Done$ 

```

```

context includes applicative-syntax begin

```

```

lemma ap-gpv-id: pure-gpv ( $\lambda x.\ x$ )  $\diamond x = x$ 
<proof>

```

```

lemma ap-gpv-comp: pure-gpv ( $\circ$ )  $\diamond u \diamond v \diamond w = u \diamond (v \diamond w)$ 
<proof>

```

```

lemma ap-gpv-homo: pure-gpv f  $\diamond pure\text{-}gpv\ x = pure\text{-}gpv\ (f\ x)$ 
<proof>

```

```

lemma ap-gpv-interchange:  $u \diamond pure\text{-}gpv\ x = pure\text{-}gpv\ (\lambda f.\ f\ x) \diamond u$ 
<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*:
 $\text{exec-gpv } \text{callee } (f \diamond x) \sigma =$
 $\text{exec-gpv } \text{callee } f \sigma \gg (\lambda(f', \sigma'). \text{pure-spmf } (\lambda(x', \sigma''). (f' x', \sigma'')) \diamond \text{exec-gpv}$
 $\text{callee } x \sigma')$
 ⟨proof⟩

lemma *exec-gpv-ap-pure* [simp]:
 $\text{exec-gpv } \text{callee } (\text{pure-gpv } f \diamond x) \sigma = \text{pure-spmf } (\text{apfst } f) \diamond \text{exec-gpv } \text{callee } x \sigma$
 ⟨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 '

⟨ML⟩

typ (' σ ', '*call*', '*ret*') *oracle*

5.1 Shared state

context **includes** *I.lifting* *lifting-syntax* **begin**

lift-definition *plus-I* :: ('*out*', '*ret*') *I* \Rightarrow ('*out'*', '*ret'*') *I* \Rightarrow ('*out*' + '*out'*', '*ret*' + '*ret'*') *I* (**infix** $\oplus_{\mathcal{I}}$ 500)

is $\lambda \text{resp1 resp2. } \lambda \text{out. case out of Inl out'} \Rightarrow \text{Inl ' resp1 out' | Inr out'} \Rightarrow \text{Inr ' resp2 out'}$ ⟨proof⟩

lemma *plus-I-sel* [simp]:

shows *outs-plus-I*: $\text{outs-I } (\text{plus-I } \mathcal{I}l \mathcal{I}r) = \text{outs-I } \mathcal{I}l <+> \text{outs-I } \mathcal{I}r$

and *responses-plus-I-Inl*: $\text{responses-I } (\text{plus-I } \mathcal{I}l \mathcal{I}r) (\text{Inl } x) = \text{Inl ' responses-I } \mathcal{I}l x$

and *responses-plus-I-Inr*: $\text{responses-I } (\text{plus-I } \mathcal{I}l \mathcal{I}r) (\text{Inr } y) = \text{Inr ' responses-I } \mathcal{I}r y$

⟨proof⟩

lemma *vimage-Inl-Plus* [*simp*]: $\text{Inl } - ' (A <+> B) = A$
and *vimage-Inr-Plus* [*simp*]: $\text{Inr } - ' (A <+> B) = B$
 ⟨*proof*⟩

lemma *vimage-Inl-image-Inr*: $\text{Inl } - ' \text{Inr } ' A = \{\}$
and *vimage-Inr-image-Inl*: $\text{Inr } - ' \text{Inl } ' A = \{\}$
 ⟨*proof*⟩

lemma *plus-I-parametric* [*transfer-rule*]:
 $(\text{rel-}\mathcal{I} \ C \ R \ ==\!>\! \text{rel-}\mathcal{I} \ C' \ R' \ ==\!>\! \text{rel-}\mathcal{I} \ (\text{rel-sum } C \ C') \ (\text{rel-sum } R \ R')) \ \text{plus-}\mathcal{I}$
 $\text{plus-}\mathcal{I}$
 ⟨*proof*⟩

lifting-update $\mathcal{I}.\text{lifting}$
lifting-forget $\mathcal{I}.\text{lifting}$

lemma *I-trivial-plus-I* [*simp*]: $\mathcal{I}\text{-trivial } (\mathcal{I}_1 \oplus_{\mathcal{I}} \mathcal{I}_2) \longleftrightarrow \mathcal{I}\text{-trivial } \mathcal{I}_1 \wedge \mathcal{I}\text{-trivial } \mathcal{I}_2$
 ⟨*proof*⟩

end

context

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

begin

primrec *plus-oracle* :: $'a + 'c \Rightarrow (('b + 'd) \times 's)$ *spmf*

where

$\text{plus-oracle } (\text{Inl } a) = \text{map-spmf } (\text{apfst } \text{Inl}) (\text{left } s \ a)$
 $|\ \text{plus-oracle } (\text{Inr } b) = \text{map-spmf } (\text{apfst } \text{Inr}) (\text{right } s \ b)$

lemma *lossless-plus-oracleI* [*intro, simp*]:

$\llbracket \bigwedge a. x = \text{Inl } a \Rightarrow \text{lossless-spmf } (\text{left } s \ a);$
 $\bigwedge b. x = \text{Inr } b \Rightarrow \text{lossless-spmf } (\text{right } s \ b) \rrbracket$
 $\Rightarrow \text{lossless-spmf } (\text{plus-oracle } x)$
 ⟨*proof*⟩

lemma *plus-oracle-split*:

$P (\text{plus-oracle } lr) \longleftrightarrow$
 $(\forall x. lr = \text{Inl } x \longrightarrow P (\text{map-spmf } (\text{apfst } \text{Inl}) (\text{left } s \ x))) \wedge$
 $(\forall y. lr = \text{Inr } y \longrightarrow P (\text{map-spmf } (\text{apfst } \text{Inr}) (\text{right } s \ y)))$
 ⟨*proof*⟩

lemma *plus-oracle-split-asm*:

$P (\text{plus-oracle } lr) \longleftrightarrow$
 $\neg ((\exists x. lr = \text{Inl } x \wedge \neg P (\text{map-spmf } (\text{apfst } \text{Inl}) (\text{left } s \ x))) \vee$
 $(\exists y. lr = \text{Inr } y \wedge \neg P (\text{map-spmf } (\text{apfst } \text{Inr}) (\text{right } s \ y))))$

<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) $I (\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r) \iff$

callee-invariant-on left $I \mathcal{I}l \wedge$ *callee-invariant-on* right $I \mathcal{I}r$

(**is** $?lhs \iff ?rhs$)

<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** $?lhs \iff ?rhs$)

<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*:

$\llbracket \bigwedge q1' q2'. \llbracket q1 = \text{Inl } q1'; q2 = \text{Inl } q2' \rrbracket \implies \text{rel-spmf } (\text{rel-prod } B \ S) \ (\text{left1 } s1 \ q1') \ (\text{left2 } s2 \ q2') \rrbracket$;
 $\llbracket \bigwedge q1' q2'. \llbracket q1 = \text{Inr } q1'; q2 = \text{Inr } q2' \rrbracket \implies \text{rel-spmf } (\text{rel-prod } D \ S) \ (\text{right1 } s1 \ q1') \ (\text{right2 } s2 \ q2') \rrbracket$;
 $S \ s1 \ s2; \text{rel-sum } A \ C \ q1 \ q2 \llbracket$
 $\implies \text{rel-spmf } (\text{rel-prod } (\text{rel-sum } B \ D) \ S) \ ((\text{left1 } \oplus_O \ \text{right1}) \ s1 \ q1) \ ((\text{left2 } \oplus_O \ \text{right2}) \ s2 \ q2)$
 $\langle \text{proof} \rangle$

end

5.2 Shared state with aborts

context

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

begin

primrec *plus-oracle-stop* :: 'a + 'c \Rightarrow (('b + 'd) option \times 's) 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 *lossless-plus-oracle-stopI* [*intro*, *simp*]:

$\llbracket \bigwedge a. x = \text{Inl } a \implies \text{lossless-spmf } (\text{left } s \ a);$
 $\llbracket \bigwedge b. x = \text{Inr } b \implies \text{lossless-spmf } (\text{right } s \ b) \rrbracket$
 $\implies \text{lossless-spmf } (\text{plus-oracle-stop } x)$
 $\langle \text{proof} \rangle$

lemma *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 *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** \oplus_O^S 500)

5.3 Disjoint state

context

fixes $left :: ('s1, 'a, 'b) oracle'$
and $right :: ('s2, 'c, 'd) oracle'$
begin

fun $parallel-oracle :: ('s1 \times 's2, 'a + 'c, 'b + 'd) oracle'$

where

$parallel-oracle (s1, s2) (Inl a) = map-spmf (map-prod Inl (\lambda s1'. (s1', s2))) (left s1 a)$
 $| parallel-oracle (s1, s2) (Inr b) = map-spmf (map-prod Inr (Pair s1)) (right s2 b)$

lemma $parallel-oracle-def$:

$parallel-oracle = (\lambda (s1, s2). case-sum (\lambda a. map-spmf (map-prod Inl (\lambda s1'. (s1', s2))) (left s1 a)) (\lambda b. map-spmf (map-prod Inr (Pair s1)) (right s2 b)))$
 $\langle proof \rangle$

lemma $lossless-parallel-oracle [simp]$:

$lossless-spmf (parallel-oracle s1s2 xy) \longleftrightarrow$
 $(\forall x. xy = Inl x \longrightarrow lossless-spmf (left (fst s1s2) x)) \wedge$
 $(\forall y. xy = Inr y \longrightarrow lossless-spmf (right (snd s1s2) y))$
 $\langle proof \rangle$

lemma $parallel-oracle-split$:

$P (parallel-oracle s1s2 lr) \longleftrightarrow$
 $(\forall s1 s2 x. s1s2 = (s1, s2) \longrightarrow lr = Inl x \longrightarrow P (map-spmf (map-prod Inl (\lambda s1'. (s1', s2))) (left s1 x))) \wedge$
 $(\forall s1 s2 y. s1s2 = (s1, s2) \longrightarrow lr = Inr y \longrightarrow P (map-spmf (map-prod Inr (Pair s1)) (right s2 y)))$
 $\langle proof \rangle$

lemma $parallel-oracle-split-asm$:

$P (parallel-oracle s1s2 lr) \longleftrightarrow$
 $\neg ((\exists s1 s2 x. s1s2 = (s1, s2) \wedge lr = Inl x \wedge \neg P (map-spmf (map-prod Inl (\lambda s1'. (s1', s2))) (left s1 x))) \vee$
 $(\exists s1 s2 y. s1s2 = (s1, s2) \wedge lr = Inr y \wedge \neg P (map-spmf (map-prod Inr (Pair s1)) (right s2 y))))$
 $\langle proof \rangle$

lemma $WT-parallel-oracle [intro!, simp]$:

$\llbracket \mathcal{I}l \vdash_c left sl \checkmark; \mathcal{I}r \vdash_c right sr \checkmark \rrbracket \Longrightarrow plus\text{-}\mathcal{I} \mathcal{I}l \mathcal{I}r \vdash_c parallel-oracle (sl, sr)$
 \checkmark
 $\langle proof \rangle$

lemma $callee-invariant-parallel-oracleI [simp, intro]$:

assumes $callee-invariant-on left \mathcal{I}l \mathcal{I}l$ $callee-invariant-on right \mathcal{I}r \mathcal{I}r$
shows $callee-invariant-on parallel-oracle (pred-prod \mathcal{I}l \mathcal{I}r) (\mathcal{I}l \oplus_{\mathcal{I}} \mathcal{I}r)$
 $\langle proof \rangle$

end

lemma *parallel-oracle-parametric*:

includes *lifting-syntax* **shows**

(($S1 \text{ ====> } CALL1 \text{ ====> } \text{rel-spmf } (\text{rel-prod } (=) S1)$)
====> ($S2 \text{ ====> } CALL2 \text{ ====> } \text{rel-spmf } (\text{rel-prod } (=) S2)$)
====> $\text{rel-prod } S1 S2 \text{ ====> } \text{rel-sum } CALL1 CALL2 \text{ ====> } \text{rel-spmf } (\text{rel-prod } (=) (\text{rel-prod } S1 S2))$)
parallel-oracle parallel-oracle
<proof>

5.4 Indexed oracles

definition *family-oracle* :: ($'i \Rightarrow ('s, 'a, 'b) \text{ oracle}'$) $\Rightarrow ('i \Rightarrow 's, 'i \times 'a, 'b) \text{ oracle}'$
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*:

lossless-spmf (*family-oracle* $f s ix$) $\longleftrightarrow \text{lossless-spmf } (f (fst ix) (s (fst ix))) (snd ix)$
<proof>

5.5 State extension

definition *extend-state-oracle* :: ($'call, 'ret, 's) \text{ callee} \Rightarrow ('call, 'ret, 's' \times 's) \text{ callee}$
(\dagger - [1000] 1000)

where *extend-state-oracle* $\text{callee} = (\lambda(s', s) x. \text{map-spmf } (\lambda(y, s). (y, (s', s)))) (\text{callee } s x)$

lemma *extend-state-oracle-simps* [*simp*]:

extend-state-oracle $\text{callee } (s', s) x = \text{map-spmf } (\lambda(y, s). (y, (s', s))) (\text{callee } s x)$
<proof>

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

lemma *extend-state-oracle-parametric* [*transfer-rule*]:

(($S \text{ ====> } C \text{ ====> } \text{rel-spmf } (\text{rel-prod } R S)$)
====> $\text{rel-prod } S' S \text{ ====> } C$
====> $\text{rel-spmf } (\text{rel-prod } R (\text{rel-prod } S' S))$)
extend-state-oracle extend-state-oracle
<proof>

lemma *extend-state-oracle-transfer*:

(($S \text{ ====> } C \text{ ====> } \text{rel-spmf } (\text{rel-prod } R S)$)
====> $\text{rel-prod2 } S \text{ ====> } C \text{ ====> } \text{rel-spmf } (\text{rel-prod } R (\text{rel-prod2 } S))$)
($\lambda \text{oracle. oracle}$) *extend-state-oracle*
<proof>

end

lemma *callee-invariant-extend-state-oracle-const* [simp]:
callee-invariant † *oracle* ($\lambda(s', s). I s'$)
 ‹*proof*›

lemma *callee-invariant-extend-state-oracle-const'*:
callee-invariant † *oracle* ($\lambda s. I (fst s)$)
 ‹*proof*›

definition *lift-stop-oracle* :: ('call, 'ret, 's) *callee* \Rightarrow ('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*:
 (($S \text{====>} C \text{====> rel-spmf (rel-prod R S)}$) ====> ($S \text{====>} C \text{====> rel-spmf (rel-prod (pcr-Some R) S)}$))
 ($\lambda x. x$) *lift-stop-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-intercept* :: 's \Rightarrow 'x1 + 'x2 \Rightarrow (('y1 + 'y2) \times 's, 'call, 'ret) *gpv*
where

plus-intercept s (Inl x) = *map-gpv* (*apfst Inl*) *id* (*left s x*)
 | *plus-intercept s (Inr x)* = *map-gpv* (*apfst Inr*) *id* (*right s x*)

end

lemma *plus-intercept-parametric* [transfer-rule]:
includes *lifting-syntax shows*
 (($S \text{====>} X1 \text{====> rel-gpv (rel-prod Y1 S) C}$)
 ====> ($S \text{====>} X2 \text{====> rel-gpv (rel-prod Y2 S) C}$)
 ====> $S \text{====>} \text{rel-sum } X1 \ X2 \text{====>} \text{rel-gpv (rel-prod (rel-sum } Y1 \ Y2) S) C$)
plus-intercept plus-intercept
 ‹*proof*›

lemma *interaction-bounded-by-plus-intercept* [*interaction-bound*]:
fixes *left right*
shows $\llbracket \bigwedge x'. x = \text{Inl } x' \implies \text{interaction-bounded-by } P \text{ (left } s \ x') \text{ (n } x')$;
 $\bigwedge y. x = \text{Inr } y \implies \text{interaction-bounded-by } P \text{ (right } s \ y) \text{ (m } y) \rrbracket$
 $\implies \text{interaction-bounded-by } P \text{ (plus-intercept left right } s \ x) \text{ (case } x \text{ of Inl } x \Rightarrow n$
 $x \mid \text{Inr } y \Rightarrow m \ y)$
<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* :: $'s \Rightarrow 'x1 + 'x2 \Rightarrow (('y1 + 'y2) \text{ option} \times 's, 'call,$
 $'ret) \text{ gpv}$

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**
 $((S \text{ =====> } X1 \text{ =====> } \text{rel-gpv } (\text{rel-prod } (\text{rel-option } Y1) \ S) \ C)$
 $\text{=====> } (S \text{ =====> } X2 \text{ =====> } \text{rel-gpv } (\text{rel-prod } (\text{rel-option } Y2) \ S) \ C)$
 $\text{=====> } S \text{ =====> } \text{rel-sum } X1 \ X2 \ \text{=====> } \text{rel-gpv } (\text{rel-prod } (\text{rel-option } (\text{rel-sum}$
 $Y1 \ Y2)) \ S) \ C)$
 $\text{plus-intercept-stop plus-intercept-stop}$
<proof>

end

7 Cyclic groups

theory *Cyclic-Group* **imports**

HOL-Algebra.Coset

begin

record $'a \ \text{cyclic-group} = 'a \ \text{monoid} +$
 $\text{generator} :: 'a \ (\mathbf{g1})$

locale *cyclic-group* = *group* G

for $G :: ('a, 'b) \ \text{cyclic-group-scheme} \ (\mathbf{structure})$

+

assumes *generator-closed* [*intro, simp*]: $\text{generator } G \in \text{carrier } G$

and *generator*: $\text{carrier } G \subseteq \text{range } (\lambda n :: \text{nat. generator } G \ [\wedge]_G \ n)$

begin

lemma *generatorE* [*elim?*]:

assumes $x \in \text{carrier } G$

obtains $n :: \text{nat}$ **where** $x = \text{generator } G \ [\wedge] \ n$

<proof>

lemma *inj-on-generator*: *inj-on* $(([\wedge]) \ \mathbf{g}) \ \{\cdot \cdot < \text{order } G\}$

<proof>

lemma *carrier-conv-generator*:

finite $(\text{carrier } G) \implies \text{carrier } G = (\lambda n. \ \mathbf{g} \ [\wedge] \ n) \ \cdot \ \{\cdot \cdot < \text{order } G\}$

<proof>

lemma *bij-betw-generator-carrier*:

finite $(\text{carrier } G) \implies \text{bij-betw} \ (\lambda n :: \text{nat}. \ \mathbf{g} \ [\wedge] \ n) \ \{\cdot \cdot < \text{order } G\} \ (\text{carrier } G)$

<proof>

end

lemma (**in monoid**) *order-in-range-Suc*: $\text{order } G \in \text{range } \text{Suc} \longleftrightarrow \text{finite} \ (\text{carrier } G)$

<proof>

end

theory *Cyclic-Group-SPMF* **imports**

Cyclic-Group

HOL-Probability.SPMF

begin

definition *sample-uniform* $:: \text{nat} \Rightarrow \text{nat } \text{spmf}$

where *sample-uniform* $n = \text{spmf-of-set} \ \{\cdot \cdot < n\}$

lemma *spmf-sample-uniform*: $\text{spmf} \ (\text{sample-uniform } n) \ x = \text{indicator} \ \{\cdot \cdot < n\} \ x / n$

<proof>

lemma *weight-sample-uniform*: $\text{weight-spmf} \ (\text{sample-uniform } n) = \text{indicator} \ (\text{range } \text{Suc}) \ n$

<proof>

lemma *weight-sample-uniform-0* [*simp*]: $\text{weight-spmf} \ (\text{sample-uniform } 0) = 0$

<proof>

lemma *weight-sample-uniform-gt-0* [*simp*]: $0 < n \implies \text{weight-spmf} \ (\text{sample-uniform } n) = 1$

<proof>

lemma *lossless-sample-uniform* [simp]: $\text{lossless-spmf } (\text{sample-uniform } n) \longleftrightarrow 0 < n$
 <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
 $\text{map-spmf } (\lambda x. \mathbf{g} [\hat{\cdot}] x \otimes c) (\text{sample-uniform } (\text{order } G)) =$
 $\text{map-spmf } (\lambda x. \mathbf{g} [\hat{\cdot}] x) (\text{sample-uniform } (\text{order } G))$
 (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.