

Effect Polymorphism in Higher-Order Logic

Andreas Lochbihler

May 26, 2024

Abstract

The notion of a *monad* cannot be expressed within higher-order logic (HOL) due to type system restrictions. We show that if a monad is used with values of only one type, this notion *can* be formalised in HOL. Based on this idea, we develop a library of effect specifications and implementations of monads and monad transformers. Hence, we can abstract over the concrete monad in HOL definitions and thus use the same definition for different (combinations of) effects. We illustrate the usefulness of effect polymorphism with a monadic interpreter for a simple language.

Contents

1	Preliminaries	2
2	Locales for monomorphic monads	5
2.1	Plain monad	5
2.2	State	6
2.3	Failure	7
2.4	Exception	8
2.5	Reader	9
2.6	Probability	9
2.7	Nondeterministic choice	12
2.7.1	Binary choice	12
2.7.2	Countable choice	13
2.8	Writer monad	14
2.9	Resumption monad	14
2.10	Commutative monad	15
2.11	Discardable monad	15
2.12	Duplicable monad	15
3	Monad implementations	15
3.1	Identity monad	15
3.1.1	Plain monad	16

3.2	Probability monad	16
3.3	Resumption	16
3.3.1	Plain monad	17
3.4	Failure and exception monad transformer	17
3.4.1	Plain monad, failure, and exceptions	19
3.4.2	Reader	20
3.4.3	State	20
3.4.4	Probability	21
3.4.5	Writer	22
3.4.6	Binary Non-determinism	22
3.4.7	Countable Non-determinism	23
3.4.8	Resumption	23
3.4.9	Commutativity	24
3.4.10	Duplicability	24
3.4.11	Parametricity	24
3.5	Reader monad transformer	25
3.5.1	Plain monad and ask	26
3.5.2	Failure	27
3.5.3	State	28
3.5.4	Probability	28
3.5.5	Binary Non-determinism	29
3.5.6	Countable Non-determinism	30
3.5.7	Resumption	30
3.5.8	Writer	31
3.5.9	Commutativity	31
3.5.10	Discardability	31
3.5.11	Duplicability	31
3.5.12	Parametricity	31
3.6	Unbounded non-determinism	33
3.7	Non-determinism transformer	33
3.7.1	Generic implementation	35
3.7.2	Parametricity	39
3.7.3	Implementation using lists	40
3.7.4	Implementation using multisets	41
3.7.5	Implementation using finite sets	43
3.7.6	Implementation using countable sets	45
3.8	State transformer	48
3.8.1	Plain monad, get, and put	49
3.8.2	Failure	50
3.8.3	Reader	50
3.8.4	Probability	51
3.8.5	Writer	52
3.8.6	Binary Non-determinism	52
3.8.7	Countable Non-determinism	53

3.8.8	Resumption	53
3.8.9	Parametricity	54
3.9	Writer monad transformer	55
3.9.1	Failure	56
3.9.2	State	56
3.9.3	Probability	57
3.9.4	Reader	58
3.9.5	Resumption	58
3.9.6	Binary Non-determinism	59
3.9.7	Countable Non-determinism	59
3.9.8	Parametricity	60
3.10	Continuation monad transformer	61
3.10.1	CallCC	61
3.10.2	Plain monad	61
3.10.3	Failure	62
3.10.4	State	62
4	Locales for monad homomorphisms	63
5	Switching between monads	66
5.1	Embedding Identity into Probability	66
5.2	State and Reader	66
5.3	- <i>spmf</i> and (-, - <i>prob</i>) <i>optionT</i>	68
5.4	Probabilities and countable non-determinism	69
6	Overloaded monad operations	70
6.1	Identity monad	71
6.2	Probability monad	71
6.3	Nondeterminism monad transformer	72
6.4	State monad transformer	74
6.5	Failure and Exception monad transformer	78
6.6	Reader monad transformer	81
6.7	Writer monad transformer	85
6.8	Continuation monad transformer	89
7	Examples	90
7.1	Monadic interpreter	90
7.1.1	Basic interpreter	90
7.1.2	Memoisation	91
7.1.3	Probabilistic interpreter	93
7.1.4	Moving between monad instances	95
7.2	Non-deterministic interpreter	98
7.3	Towers of Hanoi	100
7.4	Fast product	101

```

theory Monomorphic-Monad imports
  HOL-Probability.Probability
  HOL-Library.Multiset
  HOL-Library.Countable-Set-Type
begin

```

1 Preliminaries

```

lemma (in comp-fun-idem) fold-set-union:
   $\llbracket \text{finite } A; \text{finite } B \rrbracket \implies \text{Finite-Set.fold } f\ x\ (A \cup B) = \text{Finite-Set.fold } f\ (\text{Finite-Set.fold } f\ x\ A)\ B$ 
  <proof>

```

```

lemma (in comp-fun-idem) ffold-set-union:  $\text{ffold } f\ x\ (A \mid\cup\mid B) = \text{ffold } f\ (\text{ffold } f\ x\ A)\ B$ 
including fset.lifting <proof>

```

```

lemma relcompp-top-top [simp]: top OO top = top
<proof>

```

<ML>

```

named-theorems monad-unfold Defining equations for overloaded monad operations

```

```

context includes lifting-syntax begin

```

```

inductive rel-itself :: 'a itself  $\Rightarrow$  'b itself  $\Rightarrow$  bool
where rel-itself TYPE(-) TYPE(-)

```

```

lemma type-parametric [transfer-rule]: rel-itself TYPE('a) TYPE('b)
<proof>

```

```

lemma plus-multiset-parametric [transfer-rule]:
   $(\text{rel-mset } A \implies \text{rel-mset } A \implies \text{rel-mset } A)\ (+)\ (+)$ 
  <proof>

```

```

lemma Mempty-parametric [transfer-rule]: rel-mset A {#} {#}
  <proof>

```

```

lemma fold-mset-parametric:
  assumes 12:  $(A \implies B \implies B)\ f1\ f2$ 
  and comp-fun-commute f1 comp-fun-commute f2
  shows  $(B \implies \text{rel-mset } A \implies B)\ (\text{fold-mset } f1)\ (\text{fold-mset } f2)$ 
  <proof>

```

```

lemma rel-fset-induct [consumes 1, case-names empty step, induct pred: rel-fset]:
  assumes XY: rel-fset A X Y

```

and empty: $P \{\{\}\} \{\{\}\}$
and step: $\bigwedge X Y x y. \llbracket \text{rel-fset } A X Y; P X Y; A x y; x \notin X \vee y \notin Y \rrbracket \implies$
 $P (\text{finsert } x X) (\text{finsert } y Y)$
shows $P X Y$
 $\langle \text{proof} \rangle$

lemma *ffold-parametric:*
assumes $1\dot{2}: (A \implies B \implies B) f1 f2$
and *comp-fun-idem* $f1$ *comp-fun-idem* $f2$
shows $(B \implies \text{rel-fset } A \implies B) (\text{ffold } f1) (\text{ffold } f2)$
 $\langle \text{proof} \rangle$

end

lemma *rel-set-Grp:* $\text{rel-set } (\text{BNF-Def.Grp } A f) = \text{BNF-Def.Grp } \{X. X \subseteq A\}$
 $(\text{image } f)$
 $\langle \text{proof} \rangle$

context includes *cset.lifting* **begin**

lemma *cUNION-assoc:* $cUNION (cUNION A f) g = cUNION A (\lambda x. cUNION$
 $(f x) g)$
 $\langle \text{proof} \rangle$

lemma *cUnion-empty [simp]:* $cUnion \text{ empty} = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *cUNION-empty [simp]:* $cUNION \text{ empty } f = \text{empty}$
 $\langle \text{proof} \rangle$

lemma *cUnion-cinsert:* $cUnion (\text{cinsert } x A) = cUn x (cUnion A)$
 $\langle \text{proof} \rangle$

lemma *cUNION-cinsert:* $cUNION (\text{cinsert } x A) f = cUn (f x) (cUNION A f)$
 $\langle \text{proof} \rangle$

lemma *cUnion-csingle [simp]:* $cUnion (\text{csingle } x) = x$
 $\langle \text{proof} \rangle$

lemma *cUNION-csingle [simp]:* $cUNION (\text{csingle } x) f = f x$
 $\langle \text{proof} \rangle$

lemma *cUNION-csingle2 [simp]:* $cUNION A \text{ csingle} = A$
 $\langle \text{proof} \rangle$

lemma *cUNION-cUn:* $cUNION (cUn A B) f = cUn (cUNION A f) (cUNION B$
 $f)$
 $\langle \text{proof} \rangle$

lemma *cUNION-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 $(rel\text{-}cset\ A\ ==> (A\ ==> rel\text{-}cset\ B)\ ==> rel\text{-}cset\ B)\ cUNION\ cUNION$
 $\langle proof \rangle$

end

locale *three* =
fixes *tytok* :: 'a *itself*
assumes *ex-three*: $\exists x\ y\ z :: 'a. x \neq y \wedge x \neq z \wedge y \neq z$
begin

definition *threes* :: 'a \times 'a \times 'a **where**
 $threes = (SOME\ (x,\ y,\ z). x \neq y \wedge x \neq z \wedge y \neq z)$
definition *three₁* :: 'a (**1**) **where** **1** = *fst* *threes*
definition *three₂* :: 'a (**2**) **where** **2** = *fst* (*snd* *threes*)
definition *three₃* :: 'a (**3**) **where** **3** = *snd* (*snd* (*threes*))

lemma *three-neq-aux*: **1** \neq **2** **1** \neq **3** **2** \neq **3**
 $\langle proof \rangle$

lemmas *three-neq* [*simp*] = *three-neq-aux* *three-neq-aux*[*symmetric*]

inductive *rel-12-23* :: 'a \Rightarrow 'a \Rightarrow *bool* **where**
 $rel\text{-}12\text{-}23\ \mathbf{1}\ \mathbf{2}$
 $| rel\text{-}12\text{-}23\ \mathbf{2}\ \mathbf{3}$

lemma *bi-unique-rel-12-23* [*simp*, *transfer-rule*]: *bi-unique* *rel-12-23*
 $\langle proof \rangle$

inductive *rel-12-21* :: 'a \Rightarrow 'a \Rightarrow *bool* **where**
 $rel\text{-}12\text{-}21\ \mathbf{1}\ \mathbf{2}$
 $| rel\text{-}12\text{-}21\ \mathbf{2}\ \mathbf{1}$

lemma *bi-unique-rel-12-21* [*simp*, *transfer-rule*]: *bi-unique* *rel-12-21*
 $\langle proof \rangle$

end

lemma *bernoulli-pmf-0*: *bernoulli-pmf* 0 = *return-pmf* *False*
 $\langle proof \rangle$

lemma *bernoulli-pmf-1*: *bernoulli-pmf* 1 = *return-pmf* *True*
 $\langle proof \rangle$

lemma *bernoulli-Not*: *map-pmf* *Not* (*bernoulli-pmf* *r*) = *bernoulli-pmf* (1 - *r*)
 $\langle proof \rangle$

lemma *pmf-eqI-avoid*: $p = q$ **if** $\bigwedge i. i \neq x \implies pmf\ p\ i = pmf\ q\ i$
 $\langle proof \rangle$

2 Locales for monomorphic monads

2.1 Plain monad

type-synonym ('a, 'm) bind = 'm \Rightarrow ('a \Rightarrow 'm) \Rightarrow 'm
type-synonym ('a, 'm) return = 'a \Rightarrow 'm

locale monad-base =
 fixes return :: ('a, 'm) return
 and bind :: ('a, 'm) bind
begin

primrec sequence :: 'm list \Rightarrow ('a list \Rightarrow 'm) \Rightarrow 'm
where

 sequence [] f = f []
 | sequence (x # xs) f = bind x (λa . sequence xs (f \circ (#) a))

definition lift :: ('a \Rightarrow 'a) \Rightarrow 'm \Rightarrow 'm
where lift f x = bind x (λx . return (f x))

end

declare
 monad-base.sequence.simps [code]
 monad-base.lift-def [code]

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

lemma sequence-parametric [transfer-rule]:
 ((M \Longrightarrow (A \Longrightarrow M) \Longrightarrow M) \Longrightarrow list-all2 M \Longrightarrow (list-all2 A
 \Longrightarrow M) \Longrightarrow M) monad-base.sequence monad-base.sequence
 <proof>

lemma lift-parametric [transfer-rule]:
 ((A \Longrightarrow M) \Longrightarrow (M \Longrightarrow (A \Longrightarrow M) \Longrightarrow M) \Longrightarrow (A \Longrightarrow
 A) \Longrightarrow M \Longrightarrow M) monad-base.lift monad-base.lift
 <proof>

end

locale monad = monad-base return bind
 for return :: ('a, 'm) return
 and bind :: ('a, 'm) bind
 +
 assumes bind-assoc: $\bigwedge(x :: 'm) f g$. bind (bind x f) g = bind x (λy . bind (f y)
 g)
 and return-bind: $\bigwedge x f$. bind (return x) f = f x
 and bind-return: $\bigwedge x$. bind x return = x
begin

lemma *bind-lift* [*simp*]: $\text{bind } (\text{lift } f \ x) \ g = \text{bind } x \ (g \circ f)$
 ⟨*proof*⟩

lemma *lift-bind* [*simp*]: $\text{lift } f \ (\text{bind } m \ g) = \text{bind } m \ (\lambda x. \text{lift } f \ (g \ x))$
 ⟨*proof*⟩

end

2.2 State

type-synonym (*'s*, *'m*) *get* = (*'s* \Rightarrow *'m*) \Rightarrow *'m*
type-synonym (*'s*, *'m*) *put* = *'s* \Rightarrow *'m* \Rightarrow *'m*

locale *monad-state-base* = *monad-base* *return* *bind*
for *return* :: (*'a*, *'m*) *return*
and *bind* :: (*'a*, *'m*) *bind*
 +
fixes *get* :: (*'s*, *'m*) *get*
and *put* :: (*'s*, *'m*) *put*
begin

definition *update* :: (*'s* \Rightarrow *'s*) \Rightarrow *'m* \Rightarrow *'m*
where *update* *f* *m* = *get* ($\lambda s. \text{put } (f \ s) \ m$)

end

declare *monad-state-base.update-def* [*code*]

lemma *update-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 (((*S* ====> *M*) ====> *M*) ====> (*S* ====> *M* ====> *M*) ====> (*S* ====>
S) ====> *M* ====> *M*)
monad-state-base.update monad-state-base.update
 ⟨*proof*⟩

locale *monad-state* = *monad-state-base* *return* *bind* *get* *put* + *monad* *return* *bind*
for *return* :: (*'a*, *'m*) *return*
and *bind* :: (*'a*, *'m*) *bind*
and *get* :: (*'s*, *'m*) *get*
and *put* :: (*'s*, *'m*) *put*
 +
assumes *put-get*: $\bigwedge f. \text{put } s \ (\text{get } f) = \text{put } s \ (f \ s)$
and *get-get*: $\bigwedge f. \text{get } (\lambda s. \text{get } (f \ s)) = \text{get } (\lambda s. f \ s \ s)$
and *put-put*: $\text{put } s \ (\text{put } s' \ m) = \text{put } s' \ m$
and *get-put*: $\text{get } (\lambda s. \text{put } s \ m) = m$
and *get-const*: $\bigwedge m. \text{get } (\lambda -. \ m) = m$
and *bind-get*: $\bigwedge f \ g. \text{bind } (\text{get } f) \ g = \text{get } (\lambda s. \text{bind } (f \ s) \ g)$
and *bind-put*: $\bigwedge f. \text{bind } (\text{put } s \ m) \ f = \text{put } s \ (\text{bind } m \ f)$
begin

lemma *put-update*: $put\ s\ (update\ f\ m) = put\ (f\ s)\ m$
<proof>

lemma *update-put*: $update\ f\ (put\ s\ m) = put\ s\ m$
<proof>

lemma *bind-update*: $bind\ (update\ f\ m)\ g = update\ f\ (bind\ m\ g)$
<proof>

lemma *update-get*: $update\ f\ (get\ g) = get\ (update\ f\ \circ\ g\ \circ\ f)$
<proof>

lemma *update-const*: $update\ (\lambda\cdot.\ s)\ m = put\ s\ m$
<proof>

lemma *update-update*: $update\ f\ (update\ g\ m) = update\ (g\ \circ\ f)\ m$
<proof>

lemma *update-id*: $update\ id\ m = m$
<proof>

end

2.3 Failure

type-synonym *'m fail* = *'m*

locale *monad-fail-base* = *monad-base* *return* *bind*
 for *return* :: (*'a*, *'m*) *return*
 and *bind* :: (*'a*, *'m*) *bind*
 +
 fixes *fail* :: *'m fail*
begin

definition *assert* :: (*'a* \Rightarrow *bool*) \Rightarrow *'m* \Rightarrow *'m*
where *assert* *P* *m* = *bind* *m* ($\lambda x.$ *if* *P* *x* *then* *return* *x* *else* *fail*)

end

declare *monad-fail-base.assert-def* [*code*]

lemma *assert-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 $((A\ ==>\ M)\ ==>\ (M\ ==>\ (A\ ==>\ M)\ ==>\ M)\ ==>\ M\ ==>\ M\ ==>\ M)$
 $(A\ ==>\ (=))\ ==>\ M\ ==>\ M$
 monad-fail-base.assert *monad-fail-base.assert*
<proof>

locale *monad-fail* = *monad-fail-base* *return* *bind* *fail* + *monad* *return* *bind*
 for *return* :: (*'a*, *'m*) *return*

```

and bind :: ('a, 'm) bind
and fail :: 'm fail
+
assumes fail-bind:  $\bigwedge f. \text{bind fail } f = \text{fail}$ 
begin

```

```

lemma assert-fail: assert P fail = fail
<proof>

```

```

end

```

2.4 Exception

```

type-synonym 'm catch = 'm  $\Rightarrow$  'm  $\Rightarrow$  'm

```

```

locale monad-catch-base = monad-fail-base return bind fail
for return :: ('a, 'm) return
and bind :: ('a, 'm) bind
and fail :: 'm fail
+
fixes catch :: 'm catch

```

```

locale monad-catch = monad-catch-base return bind fail catch + monad-fail return
bind fail
for return :: ('a, 'm) return
and bind :: ('a, 'm) bind
and fail :: 'm fail
and catch :: 'm catch
+
assumes catch-return: catch (return x) m = return x
and catch-fail: catch fail m = m
and catch-fail2: catch m fail = m
and catch-assoc: catch (catch m m') m'' = catch m (catch m' m'')

```

```

locale monad-catch-state = monad-catch return bind fail catch + monad-state re-
turn bind get put
for return :: ('a, 'm) return
and bind :: ('a, 'm) bind
and fail :: 'm fail
and catch :: 'm catch
and get :: ('s, 'm) get
and put :: ('s, 'm) put
+
assumes catch-get: catch (get f) m = get ( $\lambda s. \text{catch } (f s) m$ )
and catch-put: catch (put s m) m' = put s (catch m m')
begin

```

```

lemma catch-update: catch (update f m) m' = update f (catch m m')
<proof>

```

end

2.5 Reader

As `ask` takes a continuation, we have to restate the monad laws for `ask`

type-synonym $(\prime r, \prime m)$ `ask` = $(\prime r \Rightarrow \prime m) \Rightarrow \prime m$

locale `monad-reader-base` = `monad-base return bind`

for `return` :: $(\prime a, \prime m)$ `return`

and `bind` :: $(\prime a, \prime m)$ `bind`

+

fixes `ask` :: $(\prime r, \prime m)$ `ask`

locale `monad-reader` = `monad-reader-base return bind ask + monad return bind`

for `return` :: $(\prime a, \prime m)$ `return`

and `bind` :: $(\prime a, \prime m)$ `bind`

and `ask` :: $(\prime r, \prime m)$ `ask`

+

assumes `ask-ask`: $\bigwedge f. \text{ask } (\lambda r. \text{ask } (f r)) = \text{ask } (\lambda r. f r r)$

and `ask-const`: $\text{ask } (\lambda-. m) = m$

and `bind-ask`: $\bigwedge f g. \text{bind } (\text{ask } f) g = \text{ask } (\lambda r. \text{bind } (f r) g)$

and `bind-ask2`: $\bigwedge f. \text{bind } m (\lambda x. \text{ask } (f x)) = \text{ask } (\lambda r. \text{bind } m (\lambda x. f x r))$

begin

lemma `ask-bind`: $\text{ask } (\lambda r. \text{bind } (f r) (g r)) = \text{bind } (\text{ask } f) (\lambda x. \text{ask } (\lambda r. g r x))$

<proof>

end

locale `monad-reader-state` =

`monad-reader return bind ask +`

`monad-state return bind get put`

for `return` :: $(\prime a, \prime m)$ `return`

and `bind` :: $(\prime a, \prime m)$ `bind`

and `ask` :: $(\prime r, \prime m)$ `ask`

and `get` :: $(\prime s, \prime m)$ `get`

and `put` :: $(\prime s, \prime m)$ `put`

+

assumes `ask-get`: $\bigwedge f. \text{ask } (\lambda r. \text{get } (f r)) = \text{get } (\lambda s. \text{ask } (\lambda r. f r s))$

and `put-ask`: $\bigwedge f. \text{put } s (\text{ask } f) = \text{ask } (\lambda r. \text{put } s (f r))$

2.6 Probability

type-synonym $(\prime p, \prime m)$ `sample` = $\prime p \text{ pmf} \Rightarrow (\prime p \Rightarrow \prime m) \Rightarrow \prime m$

locale `monad-prob-base` = `monad-base return bind`

for `return` :: $(\prime a, \prime m)$ `return`

and `bind` :: $(\prime a, \prime m)$ `bind`

+
fixes *sample* :: ('p, 'm) *sample*

locale *monad-prob* = *monad return bind* + *monad-prob-base return bind sample*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
and *sample* :: ('p, 'm) *sample*

+
assumes *sample-const*: $\bigwedge p m. \text{sample } p (\lambda-. m) = m$
and *sample-return-pmf*: $\bigwedge x f. \text{sample } (\text{return-pmf } x) f = f x$
and *sample-bind-pmf*: $\bigwedge p f g. \text{sample } (\text{bind-pmf } p f) g = \text{sample } p (\lambda x. \text{sample } (f x) g)$
and *sample-commute*: $\bigwedge p q f. \text{sample } p (\lambda x. \text{sample } q (f x)) = \text{sample } q (\lambda y. \text{sample } p (\lambda x. f x y))$
 — We'd like to state that we can combine independent samples rather than just commute them, but that's not possible with a monomorphic sampling operation
and *bind-sample1*: $\bigwedge p f g. \text{bind } (\text{sample } p f) g = \text{sample } p (\lambda x. \text{bind } (f x) g)$
and *bind-sample2*: $\bigwedge m f p. \text{bind } m (\lambda y. \text{sample } p (f y)) = \text{sample } p (\lambda x. \text{bind } m (\lambda y. f y x))$
and *sample-parametric*: $\bigwedge R. \text{bi-unique } R \implies \text{rel-fun } (\text{rel-pmf } R) (\text{rel-fun } (\text{rel-fun } R (=)) (=)) \text{ sample sample}$
begin

lemma *sample-cong*: $(\bigwedge x. x \in \text{set-pmf } p \implies f x = g x) \implies \text{sample } p f = \text{sample } q g$ **if** $p = q$
 ⟨*proof*⟩

end

We can implement binary probabilistic choice using *sample* provided that the sample space contains at least three elements.

locale *monad-prob3* = *monad-prob return bind sample* + *three TYPE('p)*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
and *sample* :: ('p, 'm) *sample*
begin

definition *pchoose* :: $\text{real} \Rightarrow 'm \Rightarrow 'm \Rightarrow 'm$ **where**
pchoose $r m m' = \text{sample } (\text{map-pmf } (\lambda b. \text{if } b \text{ then } \mathbf{1} \text{ else } \mathbf{2}) (\text{bernoulli-pmf } r)) (\lambda x. \text{if } x = \mathbf{1} \text{ then } m \text{ else } m')$

abbreviation *pchoose-syntax* :: $'m \Rightarrow \text{real} \Rightarrow 'm \Rightarrow 'm$ (- ◁ - ▷ - [100, 0, 100] 99) **where**
 $m \triangleleft r \triangleright m' \equiv \text{pchoose } r m m'$

lemma *pchoose-0*: $m \triangleleft 0 \triangleright m' = m'$
 ⟨*proof*⟩

lemma *pchoose-1*: $m \triangleleft 1 \triangleright m' = m$

<proof>

lemma *pchoose-idemp*: $m \triangleleft r \triangleright m = m$
<proof>

lemma *pchoose-bind1*: $\text{bind } (m \triangleleft r \triangleright m') f = \text{bind } m f \triangleleft r \triangleright \text{bind } m' f$
<proof>

lemma *pchoose-bind2*: $\text{bind } m (\lambda x. f x \triangleleft p \triangleright g x) = \text{bind } m f \triangleleft p \triangleright \text{bind } m g$
<proof>

lemma *pchoose-commute*: $m \triangleleft 1 - r \triangleright m' = m' \triangleleft r \triangleright m$
<proof>

lemma *pchoose-assoc*: $m \triangleleft p \triangleright (m' \triangleleft q \triangleright m'') = (m \triangleleft r \triangleright m') \triangleleft s \triangleright m''$ (is
?lhs = ?rhs)
if $\min 1 (\max 0 p) = \min 1 (\max 0 r) * \min 1 (\max 0 s)$
and $1 - \min 1 (\max 0 s) = (1 - \min 1 (\max 0 p)) * (1 - \min 1 (\max 0 q))$
<proof>

lemma *pchoose-assoc'*: $m \triangleleft p \triangleright (m' \triangleleft q \triangleright m'') = (m \triangleleft r \triangleright m') \triangleleft s \triangleright m''$
if $p = r * s$ **and** $1 - s = (1 - p) * (1 - q)$
and $0 \leq p \leq 1$ $0 \leq q \leq 1$ $0 \leq r \leq 1$ $0 \leq s \leq 1$
<proof>

end

locale *monad-state-prob* = *monad-state return bind get put + monad-prob return*
bind sample
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
and *get* :: ('s, 'm) *get*
and *put* :: ('s, 'm) *put*
and *sample* :: ('p, 'm) *sample*
+
assumes *sample-get*: $\text{sample } p (\lambda x. \text{get } (f x)) = \text{get } (\lambda s. \text{sample } p (\lambda x. f x s))$
begin

lemma *sample-put*: $\text{sample } p (\lambda x. \text{put } s (m x)) = \text{put } s (\text{sample } p m)$
<proof>

lemma *sample-update*: $\text{sample } p (\lambda x. \text{update } f (m x)) = \text{update } f (\text{sample } p m)$
<proof>

end

2.7 Nondeterministic choice

2.7.1 Binary choice

type-synonym $'m \text{ alt} = 'm \Rightarrow 'm \Rightarrow 'm$

locale *monad-alt-base* = *monad-base return bind*

for *return* :: ($'a, 'm$) *return*

and *bind* :: ($'a, 'm$) *bind*

+

fixes *alt* :: $'m \text{ alt}$

locale *monad-alt* = *monad return bind* + *monad-alt-base return bind alt*

for *return* :: ($'a, 'm$) *return*

and *bind* :: ($'a, 'm$) *bind*

and *alt* :: $'m \text{ alt}$

+ — Laws taken from Gibbons, Hinze: Just do it

assumes *alt-assoc*: $\text{alt } (m1 \ m2) \ m3 = \text{alt } m1 \ (\text{alt } m2 \ m3)$

and *bind-alt1*: $\text{bind } (m \ m') \ f = \text{alt } (\text{bind } m \ f) \ (\text{bind } m' \ f)$

locale *monad-fail-alt* = *monad-fail return bind fail* + *monad-alt return bind alt*

for *return* :: ($'a, 'm$) *return*

and *bind* :: ($'a, 'm$) *bind*

and *fail* :: $'m \text{ fail}$

and *alt* :: $'m \text{ alt}$

+

assumes *alt-fail1*: $\text{alt } \text{fail } m = m$

and *alt-fail2*: $\text{alt } m \ \text{fail} = m$

begin

lemma *assert-alt*: $\text{assert } P \ (\text{alt } m \ m') = \text{alt } (\text{assert } P \ m) \ (\text{assert } P \ m')$

<proof>

end

locale *monad-state-alt* = *monad-state return bind get put* + *monad-alt return bind alt*

for *return* :: ($'a, 'm$) *return*

and *bind* :: ($'a, 'm$) *bind*

and *get* :: ($'s, 'm$) *get*

and *put* :: ($'s, 'm$) *put*

and *alt* :: $'m \text{ alt}$

+

assumes *alt-get*: $\text{alt } (\text{get } f) \ (\text{get } g) = \text{get } (\lambda x. \text{alt } (f \ x) \ (g \ x))$

and *alt-put*: $\text{alt } (\text{put } s \ m) \ (\text{put } s \ m') = \text{put } s \ (\text{alt } m \ m')$

— Unlike for *sample*, we must require both *alt-get* and *alt-put* because we do not require that *bind* right-distributes over *alt*.

begin

lemma *alt-update*: $\text{alt } (\text{update } f \ m) \ (\text{update } f \ m') = \text{update } f \ (\text{alt } m \ m')$

<proof>

end

2.7.2 Countable choice

type-synonym ('c, 'm) *altc* = 'c *cset* \Rightarrow ('c \Rightarrow 'm) \Rightarrow 'm

locale *monad-altc-base* = *monad-base* *return* *bind*

for *return* :: ('a, 'm) *return*

and *bind* :: ('a, 'm) *bind*

+

fixes *altc* :: ('c, 'm) *altc*

begin

definition *fail* :: 'm *fail* **where** *fail* = *altc* *cempty* (λ -. *undefined*)

end

declare *monad-altc-base.fail-def* [*code*]

locale *monad-altc* = *monad* *return* *bind* + *monad-altc-base* *return* *bind* *altc*

for *return* :: ('a, 'm) *return*

and *bind* :: ('a, 'm) *bind*

and *altc* :: ('c, 'm) *altc*

+

assumes *bind-altc1*: $\bigwedge C g f. \text{bind } (\text{altc } C g) f = \text{altc } C (\lambda c. \text{bind } (g c) f)$

and *altc-single*: $\bigwedge x f. \text{altc } (\text{csingle } x) f = f x$

and *altc-cUNION*: $\bigwedge C f g. \text{altc } (\text{cUNION } C f) g = \text{altc } C (\lambda x. \text{altc } (f x) g)$

— We do not assume *altc-const* like for *sample* because the choice set might be empty

and *altc-parametric*: $\bigwedge R. \text{bi-unique } R \implies \text{rel-fun } (\text{rel-cset } R) (\text{rel-fun } (\text{rel-fun } R (=)) (=)) \text{ altc altc}$

begin

lemma *altc-cong*: *cBall* *C* ($\lambda x. f x = g x$) \implies *altc* *C* *f* = *altc* *C* *g*

<proof>

lemma *monad-fail* [*locale-witness*]: *monad-fail* *return* *bind* *fail*

<proof>

end

We can implement *alt* via *altc* only if we know that there are sufficiently many elements in the choice type 'c. For the associativity law, we need at least three elements.

locale *monad-altc3* = *monad-altc* *return* *bind* *altc* + *three* *TYPE*('c)

for *return* :: ('a, 'm) *return*

and *bind* :: ('a, 'm) *bind*

and *altc* :: ('c, 'm) *altc*
begin

definition *alt* :: 'm *alt*
where *alt m1 m2* = *altc (cinsert 1 (csingle 2))* ($\lambda c. \text{if } c = \mathbf{1} \text{ then } m1 \text{ else } m2$)

lemma *monad-alt: monad-alt return bind alt*
 $\langle \text{proof} \rangle$ **including** *cset.lifting* $\langle \text{proof} \rangle$ **including** *cset.lifting* $\langle \text{proof} \rangle$

end

locale *monad-state-altc* =
monad-state return bind get put +
monad-altc return bind altc
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
and *get* :: ('s, 'm) *get*
and *put* :: ('s, 'm) *put*
and *altc* :: ('c, 'm) *altc*
+
assumes *altc-get*: $\bigwedge C f. \text{altc } C (\lambda c. \text{get } (f c)) = \text{get } (\lambda s. \text{altc } C (\lambda c. f c s))$
and *altc-put*: $\bigwedge C f. \text{altc } C (\lambda c. \text{put } s (f c)) = \text{put } s (\text{altc } C f)$

2.8 Writer monad

type-synonym ('w, 'm) *tell* = 'w \Rightarrow 'm \Rightarrow 'm

locale *monad-writer-base* = *monad-base return bind*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
+
fixes *tell* :: ('w, 'm) *tell*

locale *monad-writer* = *monad-writer-base return bind tell + monad return bind*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
and *tell* :: ('w, 'm) *tell*
+
assumes *bind-tell*: $\bigwedge w m f. \text{bind } (\text{tell } w m) f = \text{tell } w (\text{bind } m f)$

2.9 Resumption monad

type-synonym ('o, 'i, 'm) *pause* = 'o \Rightarrow ('i \Rightarrow 'm) \Rightarrow 'm

locale *monad-resumption-base* = *monad-base return bind*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
+
fixes *pause* :: ('o, 'i, 'm) *pause*

locale *monad-resumption* = *monad-resumption-base* *return* *bind* *pause* + *monad*
return *bind*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
and *pause* :: ('o, 'i, 'm) *pause*
+
assumes *bind-pause*: *bind* (*pause* *out* *c*) *f* = *pause* *out* (λi . *bind* (*c* *i*) *f*)

2.10 Commutative monad

locale *monad-commute* = *monad* *return* *bind*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
+
assumes *bind-commute*: *bind* *m* (λx . *bind* *m'* (*f* *x*)) = *bind* *m'* (λy . *bind* *m* (λx .
f *x* *y*))

2.11 Discardable monad

locale *monad-discard* = *monad* *return* *bind*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
+
assumes *bind-const*: *bind* *m* (λ -. *m'*) = *m'*

2.12 Duplicable monad

locale *monad-duplicate* = *monad* *return* *bind*
for *return* :: ('a, 'm) *return*
and *bind* :: ('a, 'm) *bind*
+
assumes *bind-duplicate*: *bind* *m* (λx . *bind* *m* (*f* *x*)) = *bind* *m* (λx . *f* *x* *x*)

3 Monad implementations

3.1 Identity monad

We need a type constructor such that we can overload the monad operations

datatype 'a *id* = *return-id* (*extract*: 'a)

lemmas *return-id-parametric* = *id.ctr-transfer*

lemma *rel-id-unfold*:

rel-id *A* (*return-id* *x*) *m'* \longleftrightarrow ($\exists x'$. *m'* = *return-id* *x'* \wedge *A* *x* *x'*)
rel-id *A* *m* (*return-id* *x'*) \longleftrightarrow ($\exists x$. *m* = *return-id* *x* \wedge *A* *x* *x'*)
<*proof*>

lemma *rel-id-expand*: *M* (*extract* *m*) (*extract* *m'*) \Longrightarrow *rel-id* *M* *m* *m'*

<*proof*>

3.1.1 Plain monad

primrec *bind-id* :: ('a, 'a id) bind
where *bind-id* (return-id x) f = f x

lemma *extract-bind* [*simp*]: *extract* (bind-id x f) = *extract* (f (extract x))
<proof>

lemma *bind-id-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(rel-id A ==> (A ==> rel-id A) ==> rel-id A) bind-id bind-id
<proof>

lemma *monad-id* [*locale-witness*]: *monad* return-id bind-id
<proof>

lemma *monad-commute-id* [*locale-witness*]: *monad-commute* return-id bind-id
<proof>

lemma *monad-discard-id* [*locale-witness*]: *monad-discard* return-id bind-id
<proof>

lemma *monad-duplicate-id* [*locale-witness*]: *monad-duplicate* return-id bind-id
<proof>

3.2 Probability monad

We don't know of a sensible probability monad transformer, so we define the plain probability monad.

type-synonym 'a prob = 'a pmf

lemma *monad-prob* [*locale-witness*]: *monad* return-pmf bind-pmf
<proof>

lemma *monad-prob-prob* [*locale-witness*]: *monad-prob* return-pmf bind-pmf bind-pmf
including *lifting-syntax*
<proof>

lemma *monad-commute-prob* [*locale-witness*]: *monad-commute* return-pmf bind-pmf
<proof>

lemma *monad-discard-prob* [*locale-witness*]: *monad-discard* return-pmf bind-pmf
<proof>

3.3 Resumption

We cannot define a resumption monad transformer because the codatatype recursion would have to go through a type variable. If we plug in something like unbounded non-determinism, then the HOL type does not exist.

codatatype ('o, 'i, 'a) *resumption* = *is-Done*: Done (result: 'a) | *Pause* (output: 'o) (resume: 'i ⇒ ('o, 'i, 'a) *resumption*)

3.3.1 Plain monad

definition *return-resumption* :: 'a ⇒ ('o, 'i, 'a) *resumption*
where *return-resumption* = *Done*

primcorec *bind-resumption* :: ('o, 'i, 'a) *resumption* ⇒ ('a ⇒ ('o, 'i, 'a) *resumption*) ⇒ ('o, 'i, 'a) *resumption*
where *bind-resumption* m f = (if *is-Done* m then f (result m) else *Pause* (output m)) (λi. *bind-resumption* (resume m i) f)

definition *pause-resumption* :: 'o ⇒ ('i ⇒ ('o, 'i, 'a) *resumption*) ⇒ ('o, 'i, 'a) *resumption*
where *pause-resumption* = *Pause*

lemma *is-Done-return-resumption* [*simp*]: *is-Done* (*return-resumption* x)
 ⟨*proof*⟩

lemma *result-return-resumption* [*simp*]: *result* (*return-resumption* x) = x
 ⟨*proof*⟩

lemma *monad-resumption* [*locale-witness*]: *monad* *return-resumption* *bind-resumption*
 ⟨*proof*⟩

lemma *monad-resumption-resumption* [*locale-witness*]:
monad-resumption *return-resumption* *bind-resumption* *pause-resumption*
 ⟨*proof*⟩

3.4 Failure and exception monad transformer

The phantom type variable 'a is needed to avoid hidden polymorphism when overloading the monad operations for the failure monad transformer.

datatype (*plugins del: transfer*) (*phantom-optionT*: 'a, *set-optionT*: 'm) *optionT*
 =
OptionT (*run-option*: 'm)
for *rel*: *rel-optionT'*
map: *map-optionT'*

We define our own relator and mapper such that the phantom variable does not need any relation.

lemma *phantom-optionT* [*simp*]: *phantom-optionT* x = {}
 ⟨*proof*⟩

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

lemma *rel-optionT'-phantom*: *rel-optionT'* A = *rel-optionT'* top

<proof>

lemma *map-optionT'-phantom*: *map-optionT' f = map-optionT' undefined*
<proof>

definition *map-optionT* :: (*'m* ⇒ *'m'*) ⇒ (*'a*, *'m*) *optionT* ⇒ (*'b*, *'m'*) *optionT*
where *map-optionT* = *map-optionT' undefined*

definition *rel-optionT* :: (*'m* ⇒ *'m'* ⇒ *bool*) ⇒ (*'a*, *'m*) *optionT* ⇒ (*'b*, *'m'*)
optionT ⇒ *bool*
where *rel-optionT* = *rel-optionT' top*

lemma *rel-optionTE*:
 assumes *rel-optionT M m m'*
 obtains *x y* **where** *m = OptionT x m' = OptionT y M x y*
<proof>

lemma *rel-optionT-simps [simp]*: *rel-optionT M (OptionT m) (OptionT m')* ↔
M m m'
<proof>

lemma *rel-optionT-eq [relator-eq]*: *rel-optionT (=) = (=)*
<proof>

lemma *rel-optionT-mono [relator-mono]*: *rel-optionT A ≤ rel-optionT B* **if** *A ≤ B*
<proof>

lemma *rel-optionT-distr [relator-distr]*: *rel-optionT A OO rel-optionT B = rel-optionT*
(A OO B)
<proof>

lemma *rel-optionT-Grp*: *rel-optionT (BNF-Def.Grp A f) = BNF-Def.Grp {x.*
set-optionT x ⊆ A} (map-optionT f)
<proof>

lemma *OptionT-parametric [transfer-rule]*: (*M* ==> *rel-optionT M*) *OptionT*
OptionT
<proof>

lemma *run-option-parametric [transfer-rule]*: (*rel-optionT M* ==> *M*) *run-option*
run-option
<proof>

lemma *case-optionT-parametric [transfer-rule]*:
 (*(M ==> X) ==> rel-optionT M ==> X*) *case-optionT case-optionT*
<proof>

lemma *rec-optionT-parametric [transfer-rule]*:
 (*(M ==> X) ==> rel-optionT M ==> X*) *rec-optionT rec-optionT*

<proof>

end

3.4.1 Plain monad, failure, and exceptions

context

fixes *return* :: ('a option, 'm) *return*

and *bind* :: ('a option, 'm) *bind*

begin

definition *return-option* :: ('a, ('a, 'm) optionT) *return*

where *return-option* *x* = OptionT (return (Some *x*))

primrec *bind-option* :: ('a, ('a, 'm) optionT) *bind*

where [*code-unfold, monad-unfold*]:

bind-option (OptionT *x*) *f* =

OptionT (bind *x* (λx . case *x* of None \Rightarrow return (None :: 'a option) | Some *y* \Rightarrow run-option (*f* *y*)))

definition *fail-option* :: ('a, 'm) optionT *fail*

where [*code-unfold, monad-unfold*]: *fail-option* = OptionT (return None)

definition *catch-option* :: ('a, 'm) optionT *catch*

where *catch-option* *m* *h* = OptionT (bind (run-option *m*) (λx . if *x* = None then run-option *h* else return *x*))

lemma *run-bind-option*:

run-option (bind-option *x* *f*) = bind (run-option *x*) (λx . case *x* of None \Rightarrow return None | Some *y* \Rightarrow run-option (*f* *y*))

<proof>

lemma *run-return-option* [*simp*]: *run-option* (return-option *x*) = return (Some *x*)

<proof>

lemma *run-fail-option* [*simp*]: *run-option* fail-option = return None

<proof>

lemma *run-catch-option* [*simp*]:

run-option (catch-option *m1* *m2*) = bind (run-option *m1*) (λx . if *x* = None then run-option *m2* else return *x*)

<proof>

context

assumes *monad*: monad return bind

begin

interpretation *monad* return bind *<proof>*

lemma *monad-optionT* [*locale-witness*]: *monad return-option bind-option* (**is monad** *?return ?bind*)
 ⟨*proof*⟩

lemma *monad-fail-optionT* [*locale-witness*]:
monad-fail return-option bind-option fail-option
 ⟨*proof*⟩

lemma *monad-catch-optionT* [*locale-witness*]:
monad-catch return-option bind-option fail-option catch-option
 ⟨*proof*⟩

end

3.4.2 Reader

context
fixes *ask* :: (*'r*, *'m*) *ask*
begin

definition *ask-option* :: (*'r*, (*'a*, *'m*) *optionT*) *ask*
where [*code-unfold*, *monad-unfold*]: *ask-option f = OptionT (ask (λr. run-option (f r)))*

lemma *run-ask-option* [*simp*]: *run-option (ask-option f) = ask (λr. run-option (f r))*
 ⟨*proof*⟩

lemma *monad-reader-optionT* [*locale-witness*]:
assumes *monad-reader return bind ask*
shows *monad-reader return-option bind-option ask-option*
 ⟨*proof*⟩

end

3.4.3 State

context
fixes *get* :: (*'s*, *'m*) *get*
and *put* :: (*'s*, *'m*) *put*
begin

definition *get-option* :: (*'s*, (*'a*, *'m*) *optionT*) *get*
where *get-option f = OptionT (get (λs. run-option (f s)))*

primrec *put-option* :: (*'s*, (*'a*, *'m*) *optionT*) *put*
where *put-option s (OptionT m) = OptionT (put s m)*

lemma *run-get-option* [*simp*]:
run-option (get-option f) = get (λs. run-option (f s))

<proof>

lemma *run-put-option* [*simp*]:

run-option (put-option s m) = put s (run-option m)

<proof>

context

assumes *state: monad-state return bind get put*

begin

interpretation *monad-state return bind get put* *<proof>*

lemma *monad-state-optionT* [*locale-witness*]:

monad-state return-option bind-option get-option put-option

<proof>

lemma *monad-catch-state-optionT* [*locale-witness*]:

monad-catch-state return-option bind-option fail-option catch-option get-option put-option

<proof>

end

3.4.4 Probability

definition *altc-sample-option* :: (*'x* \Rightarrow (*'b* \Rightarrow *'m*) \Rightarrow *'m*) \Rightarrow *'x* \Rightarrow (*'b* \Rightarrow (*'a*, *'m*) *optionT*) \Rightarrow (*'a*, *'m*) *optionT*

where *altc-sample-option altc-sample p f = OptionT (altc-sample p (λx . run-option (f x)))*

lemma *run-altc-sample-option* [*simp*]: *run-option (altc-sample-option altc-sample p f) = altc-sample p (λx . run-option (f x))*

<proof>

context

fixes *sample* :: (*'p*, *'m*) *sample*

begin

abbreviation *sample-option* :: (*'p*, (*'a*, *'m*) *optionT*) *sample*

where *sample-option* \equiv *altc-sample-option sample*

lemma *monad-prob-optionT* [*locale-witness*]:

assumes *monad-prob return bind sample*

shows *monad-prob return-option bind-option sample-option*

<proof> **including** *lifting-syntax*

<proof>

lemma *monad-state-prob-optionT* [*locale-witness*]:

assumes *monad-state-prob return bind get put sample*

shows *monad-state-prob return-option bind-option get-option put-option sample-option*
 ⟨*proof*⟩

end

3.4.5 Writer

context

fixes *tell* :: ('w, 'm) *tell*

begin

definition *tell-option* :: ('w, ('a, 'm) *optionT*) *tell*

where *tell-option w m* = *OptionT (tell w (run-option m))*

lemma *run-tell-option [simp]*: *run-option (tell-option w m) = tell w (run-option m)*

⟨*proof*⟩

lemma *monad-writer-optionT [locale-witness]*:

assumes *monad-writer return bind tell*

shows *monad-writer return-option bind-option tell-option*

⟨*proof*⟩

end

3.4.6 Binary Non-determinism

context

fixes *alt* :: 'm *alt*

begin

definition *alt-option* :: ('a, 'm) *optionT alt*

where *alt-option m1 m2* = *OptionT (alt (run-option m1) (run-option m2))*

lemma *run-alt-option [simp]*: *run-option (alt-option m1 m2) = alt (run-option m1) (run-option m2)*

⟨*proof*⟩

lemma *monad-alt-optionT [locale-witness]*:

assumes *monad-alt return bind alt*

shows *monad-alt return-option bind-option alt-option*

⟨*proof*⟩

The *fail* of $(-, -)$ *optionT* does not combine with *alt* of the inner monad because $(-, -)$ *optionT* injects failures with *return None* into the inner monad.

lemma *monad-state-alt-optionT [locale-witness]*:

assumes *monad-state-alt return bind get put alt*

shows *monad-state-alt return-option bind-option get-option put-option alt-option*

<proof>

end

3.4.7 Countable Non-determinism

context

fixes $altc :: ('c, 'm) altc$

begin

abbreviation $altc-option :: ('c, ('a, 'm) optionT) altc$

where $altc-option \equiv altc-sample-option altc$

lemma $monad-altc-optionT [locale-witness]:$

assumes $monad-altc return bind altc$

shows $monad-altc return-option bind-option altc-option$

<proof> **including** $lifting-syntax$

<proof>

lemma $monad-altc3-optionT [locale-witness]:$

assumes $monad-altc3 return bind altc$

shows $monad-altc3 return-option bind-option altc-option$

<proof>

lemma $monad-state-altc-optionT [locale-witness]:$

assumes $monad-state-altc return bind get put altc$

shows $monad-state-altc return-option bind-option get-option put-option altc-option$

<proof>

end

end

3.4.8 Resumption

context

fixes $pause :: ('o, 'i, 'm) pause$

begin

definition $pause-option :: ('o, 'i, ('a, 'm) optionT) pause$

where $pause-option out c = OptionT (pause out (\lambda i. run-option (c i)))$

lemma $run-pause-option [simp]: run-option (pause-option out c) = pause out (\lambda i. run-option (c i))$

<proof>

lemma $monad-resumption-optionT [locale-witness]:$

assumes $monad-resumption return bind pause$

shows $monad-resumption return-option bind-option pause-option$

<proof>

end

3.4.9 Commutativity

lemma *monad-commute-optionT* [*locale-witness*]:
 assumes *monad-commute return bind*
 and *monad-discard return bind*
 shows *monad-commute return-option bind-option*
 ⟨*proof*⟩

3.4.10 Duplicability

lemma *monad-duplicate-optionT* [*locale-witness*]:
 assumes *monad-duplicate return bind*
 and *monad-discard return bind*
 shows *monad-duplicate return-option bind-option*
 ⟨*proof*⟩

end

3.4.11 Parametricity

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

lemma *return-option-parametric* [*transfer-rule*]:
 $((rel-option\ A\ ==>\ M)\ ==>\ A\ ==>\ rel-optionT\ M)\ return-option\ re-$
 $turn-option$
 ⟨*proof*⟩

lemma *bind-option-parametric* [*transfer-rule*]:
 $((rel-option\ A\ ==>\ M)\ ==>\ (M\ ==>\ (rel-option\ A\ ==>\ M)\ ==>\ M))$
 $==>\ rel-optionT\ M\ ==>\ (A\ ==>\ rel-optionT\ M)\ ==>\ rel-optionT$
 $M)$
 $bind-option\ bind-option$
 ⟨*proof*⟩

lemma *fail-option-parametric* [*transfer-rule*]:
 $((rel-option\ A\ ==>\ M)\ ==>\ rel-optionT\ M)\ fail-option\ fail-option$
 ⟨*proof*⟩

lemma *catch-option-parametric* [*transfer-rule*]:
 $((rel-option\ A\ ==>\ M)\ ==>\ (M\ ==>\ (rel-option\ A\ ==>\ M)\ ==>\ M))$
 $==>\ rel-optionT\ M\ ==>\ rel-optionT\ M\ ==>\ rel-optionT\ M)$
 $catch-option\ catch-option$
 ⟨*proof*⟩

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

$((R \text{====>} M) \text{====>} M) \text{====>} (R \text{====>} \text{rel-optionT } M) \text{====>} \text{rel-optionT } M$
ask-option ask-option
 <proof>

lemma *get-option-parametric* [transfer-rule]:
 $((S \text{====>} M) \text{====>} M) \text{====>} (S \text{====>} \text{rel-optionT } M) \text{====>} \text{rel-optionT } M$
get-option get-option
 <proof>

lemma *put-option-parametric* [transfer-rule]:
 $(S \text{====>} M \text{====>} M) \text{====>} S \text{====>} \text{rel-optionT } M \text{====>} \text{rel-optionT } M$
put-option put-option
 <proof>

lemma *altc-sample-option-parametric* [transfer-rule]:
 $(A \text{====>} (P \text{====>} M) \text{====>} M) \text{====>} A \text{====>} (P \text{====>} \text{rel-optionT } M) \text{====>} \text{rel-optionT } M$
altc-sample-option altc-sample-option
 <proof>

lemma *alt-option-parametric* [transfer-rule]:
 $(M \text{====>} M \text{====>} M) \text{====>} \text{rel-optionT } M \text{====>} \text{rel-optionT } M \text{====>} \text{rel-optionT } M$
alt-option alt-option
 <proof>

lemma *tell-option-parametric* [transfer-rule]:
 $(W \text{====>} M \text{====>} M) \text{====>} W \text{====>} \text{rel-optionT } M \text{====>} \text{rel-optionT } M$
tell-option tell-option
 <proof>

lemma *pause-option-parametric* [transfer-rule]:
 $(Out \text{====>} (In \text{====>} M) \text{====>} M) \text{====>} Out \text{====>} (In \text{====>} \text{rel-optionT } M) \text{====>} \text{rel-optionT } M$
pause-option pause-option
 <proof>

end

3.5 Reader monad transformer

datatype (*r*, *m*) *envT* = *EnvT* (*run-env*: *r* \Rightarrow *m*)

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

definition *rel-envT* :: (*r* \Rightarrow *r'* \Rightarrow *bool*) \Rightarrow (*m* \Rightarrow *m'* \Rightarrow *bool*) \Rightarrow (*r*, *m*) *envT* \Rightarrow (*r'*, *m'*) *envT* \Rightarrow *bool*

where *rel-envT* *R* *M* = *BNF-Def.vimage2p* *run-env* *run-env* (*R* $\text{====>} M$)

lemma *rel-envTI* [*intro!*]: (*R* $\text{====>} M$) *f g* \Longrightarrow *rel-envT* *R* *M* (*EnvT* *f*) (*EnvT*

g)
 \langle proof \rangle

lemma *rel-envT-simps*: $rel\text{-}envT\ R\ M\ (EnvT\ f)\ (EnvT\ g) \longleftrightarrow (R\ ==\!>\ M)\ f\ g$
 \langle proof \rangle

lemma *rel-envTE* [*cases pred*]:
 assumes $rel\text{-}envT\ R\ M\ m\ m'$
 obtains $f\ g$ **where** $m = EnvT\ f\ m' = EnvT\ g\ (R\ ==\!>\ M)\ f\ g$
 \langle proof \rangle

lemma *rel-envT-eq* [*relator-eq*]: $rel\text{-}envT\ (=)\ (=)\ (=)$
 \langle proof \rangle

lemma *rel-envT-mono* [*relator-mono*]: $\llbracket R \leq R'; M \leq M' \rrbracket \implies rel\text{-}envT\ R'\ M \leq rel\text{-}envT\ R\ M'$
 \langle proof \rangle

lemma *EnvT-parametric* [*transfer-rule*]: $((R\ ==\!>\ M)\ ==\!>\ rel\text{-}envT\ R\ M)\ EnvT\ EnvT$
 \langle proof \rangle

lemma *run-env-parametric* [*transfer-rule*]: $(rel\text{-}envT\ R\ M\ ==\!>\ R\ ==\!>\ M)\ run\text{-}env\ run\text{-}env$
 \langle proof \rangle

lemma *rec-envT-parametric* [*transfer-rule*]:
 $((R\ ==\!>\ M)\ ==\!>\ X)\ ==\!>\ rel\text{-}envT\ R\ M\ ==\!>\ X)\ rec\text{-}envT\ rec\text{-}envT$
 \langle proof \rangle

lemma *case-envT-parametric* [*transfer-rule*]:
 $((R\ ==\!>\ M)\ ==\!>\ X)\ ==\!>\ rel\text{-}envT\ R\ M\ ==\!>\ X)\ case\text{-}envT\ case\text{-}envT$
 \langle proof \rangle

end

3.5.1 Plain monad and ask

context

fixes $return :: ('a, 'm)\ return$

and $bind :: ('a, 'm)\ bind$

begin

definition $return\text{-}env :: ('a, ('r, 'm)\ envT)\ return$

where $return\text{-}env\ x = EnvT\ (\lambda_.\ return\ x)$

primrec $bind\text{-}env :: ('a, ('r, 'm)\ envT)\ bind$

where $bind\text{-}env\ (EnvT\ x)\ f = EnvT\ (\lambda r.\ bind\ (x\ r)\ (\lambda y.\ run\text{-}env\ (f\ y)\ r))$

definition $ask\text{-}env :: ('r, ('r, 'm) envT) ask$
where $ask\text{-}env f = EnvT (\lambda r. run\text{-}env (f r) r)$

lemma $run\text{-}bind\text{-}env [simp]: run\text{-}env (bind\text{-}env x f) r = bind (run\text{-}env x r) (\lambda y. run\text{-}env (f y) r)$
 $\langle proof \rangle$

lemma $run\text{-}return\text{-}env [simp]: run\text{-}env (return\text{-}env x) r = return x$
 $\langle proof \rangle$

lemma $run\text{-}ask\text{-}env [simp]: run\text{-}env (ask\text{-}env f) r = run\text{-}env (f r) r$
 $\langle proof \rangle$

context

assumes $monad: monad\ return\ bind$

begin

interpretation $monad\ return\ bind :: ('a, 'm) bind \langle proof \rangle$

lemma $monad\text{-}envT [locale\text{-}witness]: monad\ return\text{-}env\ bind\text{-}env$
 $\langle proof \rangle$

lemma $monad\text{-}reader\text{-}envT [locale\text{-}witness]:$
 $monad\text{-}reader\ return\text{-}env\ bind\text{-}env\ ask\text{-}env$
 $\langle proof \rangle$

end

3.5.2 Failure

context

fixes $fail :: 'm fail$

begin

definition $fail\text{-}env :: ('r, 'm) envT fail$
where $fail\text{-}env = EnvT (\lambda r. fail)$

lemma $run\text{-}fail\text{-}env [simp]: run\text{-}env fail\text{-}env r = fail$
 $\langle proof \rangle$

lemma $monad\text{-}fail\text{-}envT [locale\text{-}witness]:$
assumes $monad\text{-}fail\ return\ bind\ fail$
shows $monad\text{-}fail\ return\text{-}env\ bind\text{-}env\ fail\text{-}env$
 $\langle proof \rangle$

context

fixes $catch :: 'm catch$

begin

definition *catch-env* :: ('r, 'm) envT catch
where *catch-env m1 m2* = EnvT ($\lambda r. \text{catch } (\text{run-env } m1 \ r) \ (\text{run-env } m2 \ r)$)

lemma *run-catch-env [simp]*: $\text{run-env } (\text{catch-env } m1 \ m2) \ r = \text{catch } (\text{run-env } m1 \ r) \ (\text{run-env } m2 \ r)$
⟨proof⟩

lemma *monad-catch-envT [locale-witness]*:
assumes *monad-catch return bind fail catch*
shows *monad-catch return-env bind-env fail-env catch-env*
⟨proof⟩

end

end

3.5.3 State

context
fixes *get* :: ('s, 'm) get
and *put* :: ('s, 'm) put
begin

definition *get-env* :: ('s, ('r, 'm) envT) get
where *get-env f* = EnvT ($\lambda r. \text{get } (\lambda s. \text{run-env } (f \ s) \ r)$)

definition *put-env* :: ('s, ('r, 'm) envT) put
where *put-env s m* = EnvT ($\lambda r. \text{put } s \ (\text{run-env } m \ r)$)

lemma *run-get-env [simp]*: $\text{run-env } (\text{get-env } f) \ r = \text{get } (\lambda s. \text{run-env } (f \ s) \ r)$
⟨proof⟩

lemma *run-put-env [simp]*: $\text{run-env } (\text{put-env } s \ m) \ r = \text{put } s \ (\text{run-env } m \ r)$
⟨proof⟩

lemma *monad-state-envT [locale-witness]*:
assumes *monad-state return bind get put*
shows *monad-state return-env bind-env get-env put-env*
⟨proof⟩

3.5.4 Probability

context
fixes *sample* :: ('p, 'm) sample
begin

definition *sample-env* :: ('p, ('r, 'm) envT) sample
where *sample-env p f* = EnvT ($\lambda r. \text{sample } p \ (\lambda x. \text{run-env } (f \ x) \ r)$)

lemma *run-sample-env* [*simp*]: *run-env (sample-env p f) r = sample p (λx. run-env (f x) r)*
 ⟨*proof*⟩

lemma *monad-prob-envT* [*locale-witness*]:
assumes *monad-prob return bind sample*
shows *monad-prob return-env bind-env sample-env*
 ⟨*proof*⟩ **including** *lifting-syntax*
 ⟨*proof*⟩

lemma *monad-state-prob-envT* [*locale-witness*]:
assumes *monad-state-prob return bind get put sample*
shows *monad-state-prob return-env bind-env get-env put-env sample-env*
 ⟨*proof*⟩

end

3.5.5 Binary Non-determinism

context
fixes *alt* :: 'm *alt*
begin

definition *alt-env* :: ('r, 'm) *envT alt*
where *alt-env m1 m2 = EnvT (λr. alt (run-env m1 r) (run-env m2 r))*

lemma *run-alt-env* [*simp*]: *run-env (alt-env m1 m2) r = alt (run-env m1 r) (run-env m2 r)*
 ⟨*proof*⟩

lemma *monad-alt-envT* [*locale-witness*]:
assumes *monad-alt return bind alt*
shows *monad-alt return-env bind-env alt-env*
 ⟨*proof*⟩

lemma *monad-fail-alt-envT* [*locale-witness*]:
fixes *fail*
assumes *monad-fail-alt return bind fail alt*
shows *monad-fail-alt return-env bind-env (fail-env fail) alt-env*
 ⟨*proof*⟩

lemma *monad-state-alt-envT* [*locale-witness*]:
assumes *monad-state-alt return bind get put alt*
shows *monad-state-alt return-env bind-env get-env put-env alt-env*
 ⟨*proof*⟩

end

3.5.6 Countable Non-determinism

context

fixes $altc :: ('c, 'm) altc$

begin

definition $altc-env :: ('c, ('r, 'm) envT) altc$

where $altc-env C f = EnvT (\lambda r. altc C (\lambda c. run-env (f c) r))$

lemma $run-altc-env [simp]: run-env (altc-env C f) r = altc C (\lambda c. run-env (f c) r)$

$\langle proof \rangle$

lemma $monad-altc-envT [locale-witness]:$

assumes $monad-altc return bind altc$

shows $monad-altc return-env bind-env altc-env$

$\langle proof \rangle$ **including** $lifting-syntax$

$\langle proof \rangle$

lemma $monad-altc3-envT [locale-witness]:$

assumes $monad-altc3 return bind altc$

shows $monad-altc3 return-env bind-env altc-env$

$\langle proof \rangle$

lemma $monad-state-altc-envT [locale-witness]:$

assumes $monad-state-altc return bind get put altc$

shows $monad-state-altc return-env bind-env get-env put-env altc-env$

$\langle proof \rangle$

end

end

3.5.7 Resumption

context

fixes $pause :: ('o, 'i, 'm) pause$

begin

definition $pause-env :: ('o, 'i, ('r, 'm) envT) pause$

where $pause-env out c = EnvT (\lambda r. pause out (\lambda i. run-env (c i) r))$

lemma $run-pause-env [simp]:$

$run-env (pause-env out c) r = pause out (\lambda i. run-env (c i) r)$

$\langle proof \rangle$

lemma $monad-resumption-envT [locale-witness]:$

assumes $monad-resumption return bind pause$

shows $monad-resumption return-env bind-env pause-env$

$\langle proof \rangle$

end

3.5.8 Writer

context

fixes $tell :: ('w, 'm) \rightarrow tell$

begin

definition $tell\text{-}env :: ('w, ('r, 'm) \rightarrow envT) \rightarrow tell$

where $tell\text{-}env\ w\ m = EnvT\ (\lambda r. tell\ w\ (run\text{-}env\ m\ r))$

lemma $run\text{-}tell\text{-}env\ [simp]: run\text{-}env\ (tell\text{-}env\ w\ m)\ r = tell\ w\ (run\text{-}env\ m\ r)$

$\langle proof \rangle$

lemma $monad\text{-}writer\text{-}envT\ [locale\text{-}witness]:$

assumes $monad\text{-}writer\ return\ bind\ tell$

shows $monad\text{-}writer\ return\text{-}env\ bind\text{-}env\ tell\text{-}env$

$\langle proof \rangle$

end

3.5.9 Commutativity

lemma $monad\text{-}commute\text{-}envT\ [locale\text{-}witness]:$

assumes $monad\text{-}commute\ return\ bind$

shows $monad\text{-}commute\ return\text{-}env\ bind\text{-}env$

$\langle proof \rangle$

3.5.10 Discardability

lemma $monad\text{-}discard\text{-}envT\ [locale\text{-}witness]:$

assumes $monad\text{-}discard\ return\ bind$

shows $monad\text{-}discard\ return\text{-}env\ bind\text{-}env$

$\langle proof \rangle$

3.5.11 Duplicability

lemma $monad\text{-}duplicate\text{-}envT\ [locale\text{-}witness]:$

assumes $monad\text{-}duplicate\ return\ bind$

shows $monad\text{-}duplicate\ return\text{-}env\ bind\text{-}env$

$\langle proof \rangle$

end

3.5.12 Parametricity

context includes $lifting\text{-}syntax$ begin

lemma $return\text{-}env\text{-}parametric\ [transfer\text{-}rule]:$

$((A \text{====>} M) \text{====>} A \text{====>} \text{rel-envT } R \ M) \text{ return-env return-env}$
 $\langle \text{proof} \rangle$

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

$((M \text{====>} (A \text{====>} M) \text{====>} M) \text{====>} \text{rel-envT } R \ M \text{====>} (A \text{====>} \text{rel-envT } R \ M) \text{====>} \text{rel-envT } R \ M)$
bind-env bind-env
 $\langle \text{proof} \rangle$

lemma *ask-env-parametric* [*transfer-rule*]: $((R \text{====>} \text{rel-envT } R \ M) \text{====>} \text{rel-envT } R \ M)$ *ask-env ask-env*
 $\langle \text{proof} \rangle$

lemma *fail-env-parametric* [*transfer-rule*]: $(M \text{====>} \text{rel-envT } R \ M)$ *fail-env fail-env*
 $\langle \text{proof} \rangle$

lemma *catch-env-parametric* [*transfer-rule*]:

$((M \text{====>} M \text{====>} M) \text{====>} \text{rel-envT } R \ M \text{====>} \text{rel-envT } R \ M \text{====>} \text{rel-envT } R \ M)$ *catch-env catch-env*
 $\langle \text{proof} \rangle$

lemma *get-env-parametric* [*transfer-rule*]:

$((S \text{====>} M) \text{====>} M) \text{====>} (S \text{====>} \text{rel-envT } R \ M) \text{====>} \text{rel-envT } R \ M)$ *get-env get-env*
 $\langle \text{proof} \rangle$

lemma *put-env-parametric* [*transfer-rule*]:

$(S \text{====>} M \text{====>} M) \text{====>} S \text{====>} \text{rel-envT } R \ M \text{====>} \text{rel-envT } R \ M)$ *put-env put-env*
 $\langle \text{proof} \rangle$

lemma *sample-env-parametric* [*transfer-rule*]:

$((\text{rel-pmf } P \text{====>} (P \text{====>} M) \text{====>} M) \text{====>} \text{rel-pmf } P \text{====>} (P \text{====>} \text{rel-envT } R \ M) \text{====>} \text{rel-envT } R \ M)$
sample-env sample-env
 $\langle \text{proof} \rangle$

lemma *alt-env-parametric* [*transfer-rule*]:

$(M \text{====>} M \text{====>} M) \text{====>} \text{rel-envT } R \ M \text{====>} \text{rel-envT } R \ M \text{====>} \text{rel-envT } R \ M)$ *alt-env alt-env*
 $\langle \text{proof} \rangle$

lemma *altc-env-parametric* [*transfer-rule*]:

$((\text{rel-cset } C \text{====>} (C \text{====>} M) \text{====>} M) \text{====>} \text{rel-cset } C \text{====>} (C \text{====>} \text{rel-envT } R \ M) \text{====>} \text{rel-envT } R \ M)$
altc-env altc-env
 $\langle \text{proof} \rangle$

lemma *pause-env-parametric* [*transfer-rule*]:

$((Out \text{====>} (In \text{====>} M) \text{====>} M) \text{====>} Out \text{====>} (In \text{====>} rel\text{-env}T R M) \text{====>} rel\text{-env}T R M)$
pause-env pause-env
 <proof>

lemma *tell-env-parametric* [*transfer-rule*]:

$((W \text{====>} M \text{====>} M) \text{====>} W \text{====>} rel\text{-env}T R M \text{====>} rel\text{-env}T R M)$
tell-env tell-env
 <proof>

end

3.6 Unbounded non-determinism

abbreviation (*input*) *return-set* :: ('a, 'a set) **return where** *return-set* $x \equiv \{x\}$

abbreviation (*input*) *bind-set* :: ('a, 'a set) **bind where** *bind-set* $\equiv \lambda A f. \bigcup (f \text{ ` } A)$

abbreviation (*input*) *fail-set* :: 'a set **fail where** *fail-set* $\equiv \{\}$

abbreviation (*input*) *alt-set* :: 'a set **alt where** *alt-set* $\equiv (\cup)$

abbreviation (*input*) *altc-set* :: ('c, 'a set) **altc where** *altc-set* $C \equiv \lambda f. \bigcup (f \text{ ` } rcset C)$

lemma *monad-set* [*locale-witness*]: *monad return-set bind-set*
 <proof>

lemma *monad-fail-set* [*locale-witness*]: *monad-fail return-set bind-set fail-set*
 <proof>

lemma *monad-lift-set* [*simp*]: *monad-base.lift return-set bind-set = image*
 <proof>

lemma *monad-alt-set* [*locale-witness*]: *monad-alt return-set bind-set alt-set*
 <proof>

lemma *monad-altc-set* [*locale-witness*]: *monad-altc return-set bind-set altc-set*
including *cset.lifting lifting-syntax*
 <proof>

lemma *monad-altc3-set* [*locale-witness*]:
monad-altc3 return-set bind-set (altc-set :: ('c, 'a set) altc)
if [*locale-witness*]: *three TYPE('c)*
 <proof>

3.7 Non-determinism transformer

datatype (*plugins del: transfer*) (*phantom-nondetT: 'a, set-nondetT: 'm*) *nondetT*
 $= NondetT$ (*run-nondet: 'm*)
for *map: map-nondetT'*
rel: rel-nondetT'

We define our own relator and mapper such that the phantom variable does not need any relation.

lemma *phantom-nondetT* [*simp*]: *phantom-nondetT* $x = \{\}$
 ⟨*proof*⟩

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

lemma *rel-nondetT'-phantom*: *rel-nondetT'* $A = \text{rel-nondetT}' \text{ top}$
 ⟨*proof*⟩

lemma *map-nondetT'-phantom*: *map-nondetT'* $f = \text{map-nondetT}' \text{ undefined}$
 ⟨*proof*⟩

definition *map-nondetT* :: $('m \Rightarrow 'm') \Rightarrow ('a, 'm) \text{ nondetT} \Rightarrow ('b, 'm') \text{ nondetT}$
where *map-nondetT* = *map-nondetT'* *undefined*

definition *rel-nondetT* :: $('m \Rightarrow 'm' \Rightarrow \text{bool}) \Rightarrow ('a, 'm) \text{ nondetT} \Rightarrow ('b, 'm') \text{ nondetT} \Rightarrow \text{bool}$
where *rel-nondetT* = *rel-nondetT'* *top*

lemma *rel-nondetTE*:
assumes *rel-nondetT* $M m m'$
obtains $x y$ **where** $m = \text{NondetT } x m' = \text{NondetT } y M x y$
 ⟨*proof*⟩

lemma *rel-nondetT-simps* [*simp*]: *rel-nondetT* $M (\text{NondetT } m) (\text{NondetT } m') \longleftrightarrow M m m'$
 ⟨*proof*⟩

lemma *rel-nondetT-unfold*:
 $\bigwedge m m'. \text{rel-nondetT } M (\text{NondetT } m) m' \longleftrightarrow (\exists m''. m' = \text{NondetT } m'' \wedge M m m'')$
 $\bigwedge m m'. \text{rel-nondetT } M m (\text{NondetT } m') \longleftrightarrow (\exists m''. m = \text{NondetT } m'' \wedge M m'' m')$
 ⟨*proof*⟩

lemma *rel-nondetT-expand*: $M (\text{run-nondet } m) (\text{run-nondet } m') \Longrightarrow \text{rel-nondetT } M m m'$
 ⟨*proof*⟩

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

lemma *rel-nondetT-mono* [*relator-mono*]: *rel-nondetT* $A \leq \text{rel-nondetT } B$ **if** $A \leq B$
 ⟨*proof*⟩

lemma *rel-nondetT-distr* [*relator-distr*]: *rel-nondetT* $A \text{ OO } \text{rel-nondetT } B = \text{rel-nondetT } (A \text{ OO } B)$

<proof>

lemma *rel-nondetT-Grp*: *rel-nondetT (BNF-Def.Grp A f) = BNF-Def.Grp {x. set-nondetT x \subseteq A} (map-nondetT f)*
<proof>

lemma *NondetT-parametric [transfer-rule]*: *(M \implies rel-nondetT M) NondetT NondetT*
<proof>

lemma *run-nondet-parametric [transfer-rule]*: *(rel-nondetT M \implies M) run-nondet run-nondet*
<proof>

lemma *case-nondetT-parametric [transfer-rule]*:
((M \implies X) \implies rel-nondetT M \implies X) case-nondetT case-nondetT
<proof>

lemma *rec-nondetT-parametric [transfer-rule]*:
((M \implies X) \implies rel-nondetT M \implies X) rec-nondetT rec-nondetT
<proof>

end

3.7.1 Generic implementation

type-synonym *('a, 'm, 's) merge = 's \Rightarrow ('a \Rightarrow 'm) \Rightarrow 'm*

locale *nondetM-base = monad-base return bind*

for *return :: ('s, 'm) return*
and *bind :: ('s, 'm) bind*
and *merge :: ('a, 'm, 's) merge*
and *empty :: 's*
and *single :: 'a \Rightarrow 's*
and *union :: 's \Rightarrow 's \Rightarrow 's (infixl \cup 65)*

begin

definition *return-nondet :: ('a, ('a, 'm) nondetT) return*
where *return-nondet x = NondetT (return (single x))*

definition *bind-nondet :: ('a, ('a, 'm) nondetT) bind*
where *bind-nondet m f = NondetT (bind (run-nondet m) (λA . merge A (run-nondet \circ f)))*

definition *fail-nondet :: ('a, 'm) nondetT fail*
where *fail-nondet = NondetT (return empty)*

definition *alt-nondet :: ('a, 'm) nondetT alt*
where *alt-nondet m1 m2 = NondetT (bind (run-nondet m1) (λA . bind (run-nondet*

$m2) (\lambda B. \text{return } (A \cup B)))$

definition $\text{get-nondet} :: ('state, 'm) \text{get} \Rightarrow ('state, ('a, 'm) \text{nondetT}) \text{get}$
where $\text{get-nondet get } f = \text{NondetT } (\text{get } (\lambda s. \text{run-nondet } (f s)))$ **for** get

definition $\text{put-nondet} :: ('state, 'm) \text{put} \Rightarrow ('state, ('a, 'm) \text{nondetT}) \text{put}$
where $\text{put-nondet put } s m = \text{NondetT } (\text{put } s (\text{run-nondet } m))$ **for** put

definition $\text{ask-nondet} :: ('r, 'm) \text{ask} \Rightarrow ('r, ('a, 'm) \text{nondetT}) \text{ask}$
where $\text{ask-nondet ask } f = \text{NondetT } (\text{ask } (\lambda r. \text{run-nondet } (f r)))$

The canonical lift of sampling into $(-, -) \text{nondetT}$ does not satisfy *monad-prob*, because sampling does not distribute over *bind* backwards. Intuitively, if we sample first, then the same sample is used in all non-deterministic choices. But if we sample later, each non-deterministic choice may sample a different value.

lemma $\text{run-return-nondet [simp]: run-nondet (return-nondet } x) = \text{return (single } x)$
 $\langle \text{proof} \rangle$

lemma $\text{run-bind-nondet [simp]: run-nondet (bind-nondet } m f) = \text{bind (run-nondet } m) (\lambda A. \text{merge } A (\text{run-nondet } \circ f))$
 $\langle \text{proof} \rangle$

lemma $\text{run-fail-nondet [simp]: run-nondet fail-nondet} = \text{return empty}$
 $\langle \text{proof} \rangle$

lemma $\text{run-alt-nondet [simp]: run-nondet (alt-nondet } m1 m2) = \text{bind (run-nondet } m1) (\lambda A. \text{bind (run-nondet } m2) (\lambda B. \text{return } (A \cup B)))$
 $\langle \text{proof} \rangle$

lemma $\text{run-get-nondet [simp]: run-nondet (get-nondet get } f) = \text{get } (\lambda s. \text{run-nondet } (f s))$ **for** get
 $\langle \text{proof} \rangle$

lemma $\text{run-put-nondet [simp]: run-nondet (put-nondet put } s m) = \text{put } s (\text{run-nondet } m)$ **for** put
 $\langle \text{proof} \rangle$

lemma $\text{run-ask-nondet [simp]: run-nondet (ask-nondet ask } f) = \text{ask } (\lambda r. \text{run-nondet } (f r))$ **for** ask
 $\langle \text{proof} \rangle$

end

lemma $\text{bind-nondet-cong [cong]: nondetM-base.bind-nondet bind merge} = \text{nondetM-base.bind-nondet bind merge}$

for *bind merge* \langle *proof* \rangle

lemmas [*code*] =
 nondetM-base.return-nondet-def
 nondetM-base.bind-nondet-def
 nondetM-base.fail-nondet-def
 nondetM-base.alt-nondet-def
 nondetM-base.get-nondet-def
 nondetM-base.put-nondet-def
 nondetM-base.ask-nondet-def

locale *nondetM* = *nondetM-base return bind merge empty single union*

+
 monad-commute return bind
 for *return* :: ('s, 'm) *return*
 and *bind* :: ('s, 'm) *bind*
 and *merge* :: ('a, 'm, 's) *merge*
 and *empty* :: 's
 and *single* :: 'a \Rightarrow 's
 and *union* :: 's \Rightarrow 's \Rightarrow 's (**infixl** \cup 65)
+
 assumes *bind-merge-merge*:
 $\bigwedge y f g. \text{bind } (\text{merge } y f) (\lambda A. \text{merge } A g) = \text{merge } y (\lambda x. \text{bind } (f x) (\lambda A. \text{merge } A g))$
 and *merge-empty*: $\bigwedge f. \text{merge } \text{empty } f = \text{return } \text{empty}$
 and *merge-single*: $\bigwedge x f. \text{merge } (\text{single } x) f = f x$
 and *merge-single2*: $\bigwedge A. \text{merge } A (\lambda x. \text{return } (\text{single } x)) = \text{return } A$
 and *merge-union*: $\bigwedge A B f. \text{merge } (A \cup B) f = \text{bind } (\text{merge } A f) (\lambda A'. \text{bind } (\text{merge } B f) (\lambda B'. \text{return } (A' \cup B')))$
 and *union-assoc*: $\bigwedge A B C. (A \cup B) \cup C = A \cup (B \cup C)$
 and *empty-union*: $\bigwedge A. \text{empty} \cup A = A$
 and *union-empty*: $\bigwedge A. A \cup \text{empty} = A$
begin

lemma *monad-nondetT* [*locale-witness*]: *monad return-nondet bind-nondet*
 \langle *proof* \rangle

lemma *monad-fail-nondetT* [*locale-witness*]: *monad-fail return-nondet bind-nondet fail-nondet*
 \langle *proof* \rangle

lemma *monad-alt-nondetT* [*locale-witness*]: *monad-alt return-nondet bind-nondet alt-nondet*
 \langle *proof* \rangle

lemma *monad-fail-alt-nondetT* [*locale-witness*]:
 monad-fail-alt return-nondet bind-nondet fail-nondet alt-nondet
 \langle *proof* \rangle

lemma *monad-state-nondetT* [*locale-witness*]:

— It's not really sensible to assume a commutative state monad, but let's prove it anyway ...

fixes *get put*
assumes *monad-state return bind get put*
shows *monad-state return-nondet bind-nondet (get-nondet get) (put-nondet put)*
<proof>

lemma *monad-state-alt-nondetT* [*locale-witness*]:

fixes *get put*
assumes *monad-state return bind get put*
shows *monad-state-alt return-nondet bind-nondet (get-nondet get) (put-nondet put) alt-nondet*
<proof>

end

lemmas *nondetM-lemmas* =

nondetM.monad-nondetT
nondetM.monad-fail-nondetT
nondetM.monad-alt-nondetT
nondetM.monad-fail-alt-nondetT
nondetM.monad-state-nondetT

locale *nondetM-ask* = *nondetM return bind merge empty single union*

for *return* :: ('s, 'm) *return*
and *bind* :: ('s, 'm) *bind*
and *ask* :: ('r, 'm) *ask*
and *merge* :: ('a, 'm, 's) *merge*
and *empty* :: 's
and *single* :: 'a \Rightarrow 's
and *union* :: 's \Rightarrow 's \Rightarrow 's (**infixl** \cup 65)

+

assumes *monad-reader: monad-reader return bind ask*

assumes *merge-ask:*

$\bigwedge A (f :: 'a \Rightarrow 'r \Rightarrow ('a, 'm) \text{ nondetT}). \text{merge } A (\lambda x. \text{ask } (\lambda r. \text{run-nondet } (f x r))) =$
 $\text{ask } (\lambda r. \text{merge } A (\lambda x. \text{run-nondet } (f x r)))$

begin

interpretation *monad-reader return bind ask* *<proof>*

lemma *monad-reader-nondetT: monad-reader return-nondet bind-nondet (ask-nondet ask)*

<proof>

end

lemmas *nondetM-ask-lemmas* =

nondetM-ask.monad-reader-nondetT

3.7.2 Parametricity

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

lemma *return-nondet-parametric* [*transfer-rule*]:

$((S \text{====>} M) \text{====>} (A \text{====>} S) \text{====>} A \text{====>} \text{rel-nondetT } M)$
nondetM-base.return-nondet nondetM-base.return-nondet
<proof>

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

$((M \text{====>} (S \text{====>} M) \text{====>} M) \text{====>} (S \text{====>} (A \text{====>} M) \text{====>} M) \text{====>} \text{rel-nondetT } M \text{====>} (A \text{====>} \text{rel-nondetT } M) \text{====>} \text{rel-nondetT } M)$
nondetM-base.bind-nondet nondetM-base.bind-nondet
<proof>

lemma *fail-nondet-parametric* [*transfer-rule*]:

$((S \text{====>} M) \text{====>} S \text{====>} \text{rel-nondetT } M) \text{====>} \text{rel-nondetT } M$
nondetM-base.fail-nondet nondetM-base.fail-nondet
<proof>

lemma *alt-nondet-parametric* [*transfer-rule*]:

$((S \text{====>} M) \text{====>} (M \text{====>} (S \text{====>} M) \text{====>} M) \text{====>} (S \text{====>} S) \text{====>} \text{rel-nondetT } M \text{====>} \text{rel-nondetT } M)$
nondetM-base.alt-nondet nondetM-base.alt-nondet
<proof>

lemma *get-nondet-parametric* [*transfer-rule*]:

$((S \text{====>} M) \text{====>} M) \text{====>} (S \text{====>} \text{rel-nondetT } M) \text{====>} \text{rel-nondetT } M$
nondetM-base.get-nondet nondetM-base.get-nondet
<proof>

lemma *put-nondet-parametric* [*transfer-rule*]:

$((S \text{====>} M \text{====>} M) \text{====>} S \text{====>} \text{rel-nondetT } M \text{====>} \text{rel-nondetT } M)$
nondetM-base.put-nondet nondetM-base.put-nondet
<proof>

lemma *ask-nondet-parametric* [*transfer-rule*]:

$((R \text{====>} M) \text{====>} M) \text{====>} (R \text{====>} \text{rel-nondetT } M) \text{====>} \text{rel-nondetT } M$
nondetM-base.ask-nondet nondetM-base.ask-nondet
<proof>

end

3.7.3 Implementation using lists

context

fixes $return :: ('a\ list, 'm)\ return$
and $bind :: ('a\ list, 'm)\ bind$
and $lunionM\ lUnionM$
defines $lunionM\ m1\ m2 \equiv bind\ m1\ (\lambda A.\ bind\ m2\ (\lambda B.\ return\ (A\ @\ B)))$
and $lUnionM\ ms \equiv foldr\ lunionM\ ms\ (return\ [])$

begin

definition $lmerge :: 'a\ list \Rightarrow ('a \Rightarrow 'm) \Rightarrow 'm$ **where**

$lmerge\ A\ f = lUnionM\ (map\ f\ A)$

context

assumes $monad-commute\ return\ bind$

begin

interpretation $monad-commute\ return\ bind$ $\langle proof \rangle$

interpretation $nondetM-base\ return\ bind\ lmerge\ []\ \lambda x.\ [x]\ (@)\ \langle proof \rangle$

lemma $lUnionM-empty\ [simp]: lUnionM\ [] = return\ []$ $\langle proof \rangle$

lemma $lUnionM-Cons\ [simp]: lUnionM\ (x\ \#\ M) = lunionM\ x\ (lUnionM\ M)$ **for**
 $x\ M$

$\langle proof \rangle$

lemma $lunionM-return-empty1\ [simp]: lunionM\ (return\ [])\ x = x$ **for** x

$\langle proof \rangle$

lemma $lunionM-return-empty2\ [simp]: lunionM\ x\ (return\ []) = x$ **for** x

$\langle proof \rangle$

lemma $lunionM-return-return\ [simp]: lunionM\ (return\ A)\ (return\ B) = return\ (A\ @\ B)$ **for** $A\ B$

$\langle proof \rangle$

lemma $lunionM-assoc: lunionM\ (lunionM\ x\ y)\ z = lunionM\ x\ (lunionM\ y\ z)$ **for**
 $x\ y\ z$

$\langle proof \rangle$

lemma $lunionM-lUnionM1: lunionM\ (lUnionM\ A)\ x = foldr\ lunionM\ A\ x$ **for** A
 x

$\langle proof \rangle$

lemma $lUnionM-append\ [simp]: lUnionM\ (A\ @\ B) = lunionM\ (lUnionM\ A)\ (lUnionM\ B)$ **for** $A\ B$

$\langle proof \rangle$

lemma $lUnionM-return\ [simp]: lUnionM\ (map\ (\lambda x.\ return\ [x])\ A) = return\ A$ **for**
 A

$\langle proof \rangle$

lemma $bind-lunionM: bind\ (lunionM\ m\ m')\ f = lunionM\ (bind\ m\ f)\ (bind\ m'\ f)$
if $\bigwedge A\ B.\ f\ (A\ @\ B) = bind\ (f\ A)\ (\lambda x.\ bind\ (f\ B)\ (\lambda y.\ return\ (x\ @\ y)))$ **for** m
 $m'\ f$

$\langle proof \rangle$

lemma $list-nondetM: nondetM\ return\ bind\ lmerge\ []\ (\lambda x.\ [x])\ (@)$

$\langle proof \rangle$

lemma *list-nondetM-ask*:
notes *list-nondetM* [*locale-witness*]
assumes [*locale-witness*]: *monad-reader return bind ask*
shows *nondetM-ask return bind ask lmerge [] (λx. [x]) (@)*
 ⟨*proof*⟩

lemmas *list-nondetMs* [*locale-witness*] =
nondetM-lemmas [*OF list-nondetM*]
nondetM-ask-lemmas [*OF list-nondetM-ask*]

end

end

lemma *lmerge-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 ((*list-all2 A ==> M*) ==> (*M ==> (list-all2 A ==> M)*) ==> *M*)
 ==> *list-all2 A ==> (A ==> M)* ==> *M*)
lmerge lmerge
 ⟨*proof*⟩

3.7.4 Implementation using multisets

context

fixes *return* :: ('a multiset, 'm) *return*
and *bind* :: ('a multiset, 'm) *bind*
and *mUnionM* *mUnionM*
defines *munionM* *m1 m2* ≡ *bind m1 (λA. bind m2 (λB. return (A + B)))*
and *mUnionM* ≡ *fold-mset munionM (return {#})*

begin

definition *mmerge* :: 'a multiset ⇒ ('a ⇒ 'm) ⇒ 'm
where *mmerge* *A f* = *mUnionM (image-mset f A)*

context

assumes *monad-commute return bind*

begin

interpretation *monad-commute return bind* ⟨*proof*⟩

interpretation *nondetM-base return bind mmerge {#} λx. {#x#} (+)* ⟨*proof*⟩

lemma *munionM-comp-fun-commute*: *comp-fun-commute munionM*
 ⟨*proof*⟩

interpretation *comp-fun-commute munionM* ⟨*proof*⟩

lemma *mUnionM-empty* [*simp*]: *mUnionM {#} = return {#}* ⟨*proof*⟩

lemma *mUnionM-add-mset* [*simp*]: *mUnionM (add-mset x M) = munionM x*
 (*mUnionM M*) **for** *x M*

$\langle \text{proof} \rangle$
lemma *munionM-return-empty1* [simp]: *munionM* (return {#}) *x* = *x* **for** *x*
 $\langle \text{proof} \rangle$
lemma *munionM-return-empty2* [simp]: *munionM* *x* (return {#}) = *x* **for** *x*
 $\langle \text{proof} \rangle$
lemma *munionM-return-return* [simp]: *munionM* (return *A*) (return *B*) = return
(*A* + *B*) **for** *A B*
 $\langle \text{proof} \rangle$
lemma *munionM-assoc*: *munionM* (*munionM* *x y*) *z* = *munionM* *x* (*munionM* *y*
z) **for** *x y z*
 $\langle \text{proof} \rangle$
lemma *munionM-commute*: *munionM* *x y* = *munionM* *y x* **for** *x y*
 $\langle \text{proof} \rangle$
lemma *munionM-mUnionM1*: *munionM* (*mUnionM* *A*) *x* = *fold-mset munionM*
x A **for** *A x*
 $\langle \text{proof} \rangle$
lemma *munionM-mUnionM2*: *munionM* *x* (*mUnionM* *A*) = *fold-mset munionM*
x A **for** *x A*
 $\langle \text{proof} \rangle$
lemma *mUnionM-add* [simp]: *mUnionM* (*A* + *B*) = *munionM* (*mUnionM* *A*)
(*mUnionM* *B*) **for** *A B*
 $\langle \text{proof} \rangle$
lemma *mUnionM-return* [simp]: *mUnionM* (*image-mset* ($\lambda x. \text{return } \{\#x\# \}$) *A*)
= *return A* **for** *A*
 $\langle \text{proof} \rangle$
lemma *bind-munionM*: *bind* (*munionM* *m m'*) *f* = *munionM* (*bind* *m f*) (*bind* *m'*
f)
if $\bigwedge A B. f (A + B) = \text{bind} (f A) (\lambda x. \text{bind} (f B) (\lambda y. \text{return } (x + y)))$ **for** *m*
m' f
 $\langle \text{proof} \rangle$

lemma *mset-nondetM*: *nondetM* return *bind* *mmerge* {#} ($\lambda x. \{\#x\# \}$) (+)
 $\langle \text{proof} \rangle$

lemma *mset-nondetM-ask*:
notes *mset-nondetM*[*locale-witness*]
assumes [*locale-witness*]: *monad-reader* return *bind* *ask*
shows *nondetM-ask* return *bind* *ask* *mmerge* {#} ($\lambda x. \{\#x\# \}$) (+)
 $\langle \text{proof} \rangle$

lemmas *mset-nondetMs* [*locale-witness*] =
nondetM-lemmas[*OF mset-nondetM*]
nondetM-ask-lemmas[*OF mset-nondetM-ask*]

end

end

lemma *mmerge-parametric*:

includes *lifting-syntax*
assumes *return* [*transfer-rule*]: (*rel-mset* *A* \implies *M*) *return1* *return2*
and *bind* [*transfer-rule*]: (*M* \implies (*rel-mset* *A* \implies *M*) \implies *M*) *bind1*
bind2
and *comm1*: *monad-commute* *return1* *bind1*
and *comm2*: *monad-commute* *return2* *bind2*
shows (*rel-mset* *A* \implies (*A* \implies *M*) \implies *M*) (*mmerge* *return1* *bind1*)
(*mmerge* *return2* *bind2*)
 \langle *proof* \rangle

3.7.5 Implementation using finite sets

context

fixes *return* :: ('a *fset*, 'm) *return*
and *bind* :: ('a *fset*, 'm) *bind*
and *funionM* *fUnionM*
defines *funionM* *m1* *m2* \equiv *bind* *m1* (λ A. *bind* *m2* (λ B. *return* (*A* \cup *B*)))
and *fUnionM* \equiv *ffold* *funionM* (*return* $\{\}$)
begin

definition *fmerge* :: 'a *fset* \Rightarrow ('a \Rightarrow 'm) \Rightarrow 'm
where *fmerge* *A* *f* = *fUnionM* (*fimage* *f* *A*)

context

assumes *monad-commute* *return* *bind*
and *monad-duplicate* *return* *bind*
begin

interpretation *monad-commute* *return* *bind* \langle *proof* \rangle

interpretation *monad-duplicate* *return* *bind* \langle *proof* \rangle

interpretation *nondetM-base* *return* *bind* *fmerge* $\{\}$ λ x. $\{|x|\}$ ($|\cup|$) \langle *proof* \rangle

lemma *funionM-comp-fun-commute*: *comp-fun-commute* *funionM*
 \langle *proof* \rangle

interpretation *comp-fun-commute* *funionM* \langle *proof* \rangle

lemma *funionM-comp-fun-idem*: *comp-fun-idem* *funionM*
 \langle *proof* \rangle

interpretation *comp-fun-idem* *funionM* \langle *proof* \rangle

lemma *fUnionM-empty* [*simp*]: *fUnionM* $\{\}$ = *return* $\{\}$ \langle *proof* \rangle

lemma *fUnionM-finset* [*simp*]: *fUnionM* (*finsert* *x* *M*) = *funionM* *x* (*fUnionM* *M*)
for *x* *M*

\langle *proof* \rangle

lemma *funionM-return-empty1* [*simp*]: *funionM* (*return* $\{\}$) *x* = *x* **for** *x*
 \langle *proof* \rangle

lemma *funionM-return-empty2* [*simp*]: *funionM* *x* (*return* $\{\}$) = *x* **for** *x*

$\langle \text{proof} \rangle$
lemma *funionM-return-return* [simp]: *funionM* (return *A*) (return *B*) = return (*A* | \cup | *B*) **for** *A B*
 $\langle \text{proof} \rangle$
lemma *funionM-assoc*: *funionM* (*funionM* *x y*) *z* = *funionM* *x* (*funionM* *y z*) **for** *x y z*
 $\langle \text{proof} \rangle$
lemma *funionM-commute*: *funionM* *x y* = *funionM* *y x* **for** *x y*
 $\langle \text{proof} \rangle$
lemma *funionM-fUnionM1*: *funionM* (*fUnionM* *A*) *x* = *ffold funionM* *x A* **for** *A x*
 $\langle \text{proof} \rangle$
lemma *funionM-fUnionM2*: *funionM* *x* (*fUnionM* *A*) = *ffold funionM* *x A* **for** *x A*
 $\langle \text{proof} \rangle$
lemma *fUnionM-funion* [simp]: *fUnionM* (*A* | \cup | *B*) = *funionM* (*fUnionM* *A*) (*fUnionM* *B*) **for** *A B*
 $\langle \text{proof} \rangle$
lemma *fUnionM-return* [simp]: *fUnionM* (*fimage* ($\lambda x.$ return $\{|x|\}$) *A*) = return *A* **for** *A*
 $\langle \text{proof} \rangle$
lemma *bind-funionM*: *bind* (*funionM* *m m'*) *f* = *funionM* (*bind* *m f*) (*bind* *m' f*)
if $\bigwedge A B. f$ (*A* | \cup | *B*) = *bind* (*f A*) ($\lambda x.$ *bind* (*f B*) ($\lambda y.$ return (*x* | \cup | *y*))) **for** *m m' f*
 $\langle \text{proof} \rangle$
lemma *fUnionM-return-fempty* [simp]: *fUnionM* (*fimage* ($\lambda x.$ return $\{|\}$) *A*) = return $\{|\}$ **for** *A*
 $\langle \text{proof} \rangle$
lemma *funionM-bind*: *funionM* (*bind* *m f*) (*bind* *m g*) = *bind* *m* ($\lambda x.$ *funionM* (*f x*) (*g x*)) **for** *m f g*
 $\langle \text{proof} \rangle$
lemma *fUnionM-funionM*:
fUnionM (($\lambda y.$ *funionM* (*f y*) (*g y*)) | \dagger | *A*) = *funionM* (*fUnionM* (*f* | \dagger | *A*)) (*fUnionM* (*g* | \dagger | *A*)) **for** *f g A*
 $\langle \text{proof} \rangle$

lemma *fset-nondetM*: *nondetM* return *bind fmerge* $\{|\}$ ($\lambda x.$ $\{|x|\}$) (| \cup |)
 $\langle \text{proof} \rangle$

lemma *fset-nondetM-ask*:
notes *fset-nondetM*[*locale-witness*]
assumes [*locale-witness*]: *monad-reader* return *bind ask*
shows *nondetM-ask* return *bind ask fmerge* $\{|\}$ ($\lambda x.$ $\{|x|\}$) (| \cup |)
 $\langle \text{proof} \rangle$

lemmas *fset-nondetMs* [*locale-witness*] =
nondetM-lemmas[*OF fset-nondetM*]
nondetM-ask-lemmas[*OF fset-nondetM-ask*]

context

assumes *monad-discard return bind*
begin

interpretation *monad-discard return bind* \langle *proof* \rangle

lemma *fmerge-bind*:

$fmerge\ A\ (\lambda x. bind\ m'\ (\lambda A'. fmerge\ A'\ (f\ x))) = bind\ m'\ (\lambda A'. fmerge\ A\ (\lambda x. fmerge\ A'\ (f\ x)))$
 \langle *proof* \rangle

lemma *fmerge-commute*: $fmerge\ A\ (\lambda x. fmerge\ B\ (f\ x)) = fmerge\ B\ (\lambda y. fmerge\ A\ (\lambda x. f\ x\ y))$
 \langle *proof* \rangle

lemma *monad-commute-nondetT-fset* [*locale-witness*]:
monad-commute return-nondet bind-nondet
 \langle *proof* \rangle

end

end

end

lemma *fmerge-parametric*:

includes *lifting-syntax*
assumes *return* [*transfer-rule*]: $(rel-fset\ A\ ==>\ M)\ return1\ return2$
and *bind* [*transfer-rule*]: $(M\ ==>\ (rel-fset\ A\ ==>\ M)) ==>\ M)\ bind1\ bind2$
and *comm1*: *monad-commute return1 bind1 monad-duplicate return1 bind1*
and *comm2*: *monad-commute return2 bind2 monad-duplicate return2 bind2*
shows $(rel-fset\ A\ ==>\ (A\ ==>\ M)) ==>\ M)\ (fmerge\ return1\ bind1)\ (fmerge\ return2\ bind2)$
 \langle *proof* \rangle

3.7.6 Implementation using countable sets

For non-finite choices, we cannot generically construct the merge operation. So we formalize in a locale what can be proven generically and then prove instances of the locale for concrete locale implementations.

We need two separate merge parameters because we must merge effects over choices (type *'c*) and effects over the non-deterministic results (type *'a*) of computations.

locale *cset-nondetM-base* =
nondetM-base return bind merge cempty csingle cUn
for *return* :: $(\ 'a\ cset, \ 'm)\ return$

and *bind* :: ('a cset, 'm) *bind*
and *merge* :: ('a, 'm, 'a cset) *merge*
and *mergex* :: ('c, 'm, 'c cset) *mergex*
begin

definition *altc-nondet* :: ('c, ('a, 'm) *nondetT*) *altc* **where**
altc-nondet A f = *NondetT* (*mergex* A (*run-nondet* ◦ f))

lemma *run-altc-nondet* [*simp*]: *run-nondet* (*altc-nondet* A f) = *mergex* A (*run-nondet* ◦ f)
 ⟨*proof*⟩

end

locale *cset-nondetM* =

cset-nondetM-base *return* *bind* *merge* *mergex*

+

monad-commute *return* *bind*

+

monad-duplicate *return* *bind*

for *return* :: ('a cset, 'm) *return*

and *bind* :: ('a cset, 'm) *bind*

and *merge* :: ('a, 'm, 'a cset) *merge*

and *mergex* :: ('c, 'm, 'c cset) *mergex*

+

assumes *bind-merge-merge*:

$\bigwedge y f g. \text{bind } (\text{merge } y f) (\lambda A. \text{merge } A g) = \text{merge } y (\lambda x. \text{bind } (f x) (\lambda A. \text{merge } A g))$

and *merge-empty*: $\bigwedge f. \text{merge } \text{empty } f = \text{return } \text{empty}$

and *merge-single*: $\bigwedge x f. \text{merge } (\text{csingle } x) f = f x$

and *merge-single2*: $\bigwedge A. \text{merge } A (\lambda x. \text{return } (\text{csingle } x)) = \text{return } A$

and *merge-union*: $\bigwedge A B f. \text{merge } (\text{cUn } A B) f = \text{bind } (\text{merge } A f) (\lambda A'. \text{bind } (\text{merge } B f) (\lambda B'. \text{return } (\text{cUn } A' B')))$

and *bind-mergex-merge*:

$\bigwedge y f g. \text{bind } (\text{mergex } y f) (\lambda A. \text{merge } A g) = \text{mergex } y (\lambda x. \text{bind } (f x) (\lambda A. \text{merge } A g))$

and *mergex-single*: $\bigwedge x f. \text{mergex } (\text{csingle } x) f = f x$

and *mergex-UNION*: $\bigwedge C f g. \text{mergex } (\text{cUNION } C f) g = \text{mergex } C (\lambda x. \text{mergex } (f x) g)$

and *mergex-parametric* [*transfer-rule*]:

$\bigwedge R. \text{bi-unique } R \implies \text{rel-fun } (\text{rel-cset } R) (\text{rel-fun } (\text{rel-fun } R (=)) (=)) \text{mergex } \text{mergex}$

begin

interpretation *nondetM* *return* *bind* *merge* *empty* *csingle* *cUn*

⟨*proof*⟩

sublocale *nondet*: *monad-altc* *return-nondet* *bind-nondet* *altc-nondet*

including *lifting-syntax*

<proof>

end

locale *cset-nondetM3* =
 cset-nondetM *return* *bind* *merge* *mergec*
 +
 three *TYPE*('c)
 for *return* :: ('a *cset*, 'm) *return*
 and *bind* :: ('a *cset*, 'm) *bind*
 and *merge* :: ('a, 'm, 'a *cset*) *merge*
 and *mergec* :: ('c, 'm, 'c *cset*) *merge*
begin

interpretation *nondet*: *monad-altc3* *return-nondet* *bind-nondet* *altc-nondet* *<proof>*

end

Identity monad definition *merge-id* :: ('c, 'a *cset* *id*, 'c *cset*) *merge* **where**
 merge-id *A* *f* = *return-id* (*cUNION* *A* (*extract* \circ *f*))

lemma *extract-merge-id* [*simp*]: *extract* (*merge-id* *A* *f*) = *cUNION* *A* (*extract* \circ *f*)
<proof>

lemma *merge-id-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 (*rel-cset* *A* \implies (*A* \implies *rel-id* (*rel-cset* *A*)) \implies *rel-id* (*rel-cset* *A*))
merge-id *merge-id*
<proof>

lemma *cset-nondetM-id* [*locale-witness*]: *cset-nondetM* *return-id* *bind-id* *merge-id*
merge-id
 including *lifting-syntax*
<proof>

Reader monad transformer definition *merge-env* :: ('c, 'm, 'c *cset*) *merge*
 \Rightarrow ('c, ('r, 'm) *envT*, 'c *cset*) *merge* **where**
 merge-env *merge* *A* *f* = *EnvT* (λr . *merge* *A* (λa . *run-env* (*f* *a*) *r*)) **for** *merge*

lemma *run-merge-env* [*simp*]: *run-env* (*merge-env* *merge* *A* *f*) *r* = *merge* *A* (λa .
run-env (*f* *a*) *r*) **for** *merge*
<proof>

lemma *merge-env-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
 ((*rel-cset* *C* \implies (*C* \implies *M*) \implies *M*) \implies *rel-cset* *C* \implies (*C*
 \implies *rel-envT* *R* *M*) \implies *rel-envT* *R* *M*)
 merge-env *merge-env*
<proof>

lemma *cset-nondetM-envT* [*locale-witness*]:

```

fixes return :: ('a cset, 'm) return
and bind :: ('a cset, 'm) bind
and merge :: ('a, 'm, 'a cset) merge
and mergec :: ('c, 'm, 'c cset) merge
assumes cset-nondetM return bind merge mergec
shows cset-nondetM (return-env return) (bind-env bind) (merge-env merge)
(merge-env mergec)
<proof> including lifting-syntax
<proof>

```

3.8 State transformer

```

datatype ('s, 'm) stateT = StateT (run-state: 's ⇒ 'm)
for rel: rel-stateT'

```

We define a more general relator for $(-, -)$ *stateT* than the one generated by the datatype package such that we can also show parametricity in the state.

```

context includes lifting-syntax begin

```

```

definition rel-stateT :: ('s ⇒ 's' ⇒ bool) ⇒ ('m ⇒ 'm' ⇒ bool) ⇒ ('s, 'm) stateT
⇒ ('s', 'm') stateT ⇒ bool
where rel-stateT S M m m' ←→ (S ===> M) (run-state m) (run-state m')

```

```

lemma rel-stateT-eq [relator-eq]: rel-stateT (=) (=) = (=)
<proof>

```

```

lemma rel-stateT-mono [relator-mono]: [[ S' ≤ S; M ≤ M' ]] ⇒ rel-stateT S M
≤ rel-stateT S' M'
<proof>

```

```

lemma StateT-parametric [transfer-rule]: ((S ===> M) ===> rel-stateT S M)
StateT StateT
<proof>

```

```

lemma run-state-parametric [transfer-rule]: (rel-stateT S M ===> S ===> M)
run-state run-state
<proof>

```

```

lemma case-stateT-parametric [transfer-rule]:
(((S ===> M) ===> A) ===> rel-stateT S M ===> A) case-stateT case-stateT
<proof>

```

```

lemma rec-stateT-parametric [transfer-rule]:
(((S ===> M) ===> A) ===> rel-stateT S M ===> A) rec-stateT rec-stateT
<proof>

```

```

lemma rel-stateT-Grp: rel-stateT (=) (BNF-Def.Grp UNIV f) = BNF-Def.Grp
UNIV (map-stateT f)
<proof>

```

end

3.8.1 Plain monad, get, and put

context

fixes $return :: ('a \times 's, 'm) \rightarrow 'a$

and $bind :: ('a \times 's, 'm) \rightarrow 'a \rightarrow 'a \times 's$

begin

primrec $bind\text{-}state :: ('a, ('s, 'm) \text{state}T) \rightarrow 'a \rightarrow 'a \times 's$

where $bind\text{-}state (StateT\ x)\ f = StateT\ (\lambda s. bind\ (x\ s)\ (\lambda(a, s'). run\text{-}state\ (f\ a)\ s'))$

definition $return\text{-}state :: ('a, ('s, 'm) \text{state}T) \rightarrow 'a \times 's$

where $return\text{-}state\ x = StateT\ (\lambda s. return\ (x, s))$

definition $get\text{-}state :: ('s, ('s, 'm) \text{state}T) \rightarrow 's$

where $get\text{-}state\ f = StateT\ (\lambda s. run\text{-}state\ (f\ s)\ s)$

primrec $put\text{-}state :: ('s, ('s, 'm) \text{state}T) \rightarrow 's \rightarrow 's$

where $put\text{-}state\ s (StateT\ f) = StateT\ (\lambda-. f\ s)$

lemma $run\text{-}put\text{-}state$ [simp]: $run\text{-}state\ (put\text{-}state\ s\ m)\ s' = run\text{-}state\ m\ s$
(proof)

lemma $run\text{-}get\text{-}state$ [simp]: $run\text{-}state\ (get\text{-}state\ f)\ s = run\text{-}state\ (f\ s)\ s$
(proof)

lemma $run\text{-}bind\text{-}state$ [simp]:

$run\text{-}state\ (bind\text{-}state\ x\ f)\ s = bind\ (run\text{-}state\ x\ s)\ (\lambda(a, s'). run\text{-}state\ (f\ a)\ s')$
(proof)

lemma $run\text{-}return\text{-}state$ [simp]:

$run\text{-}state\ (return\text{-}state\ x)\ s = return\ (x, s)$
(proof)

context

assumes $monad: monad\ return\ bind$

begin

interpretation $monad\ return\ bind$ (proof)

lemma $monad\text{-}stateT$ [locale-witness]: $monad\ return\text{-}state\ bind\text{-}state$ (is $monad$
 $?return\ ?bind$)

(proof)

lemma $monad\text{-}state\text{-}stateT$ [locale-witness]:

$monad\text{-}state\ return\text{-}state\ bind\text{-}state\ get\text{-}state\ put\text{-}state$

<proof>

end

We cannot define a generic lifting operation for state like in Haskell. If we separate the monad type variable from the element type variable, then *lift* should have type $'a \ 'm \Rightarrow (('a \times \ 's) \ 'm) \ stateT$, but this means that the type of results must change, which does not work for monomorphic monads. Instead, we must lift all operations individually. *lift-definition* does not work because the monad transformer type is typically larger than the base type, but *lift-definition* only works if the lifted type is no bigger.

3.8.2 Failure

context

fixes *fail* :: *'m fail*

begin

definition *fail-state* :: (*'s, 'm*) *stateT fail*

where *fail-state* = *StateT* ($\lambda s. \ fail$)

lemma *run-fail-state* [*simp*]: *run-state fail-state s = fail*

<proof>

lemma *monad-fail-stateT* [*locale-witness*]:

assumes *monad-fail return bind fail*

shows *monad-fail return-state bind-state fail-state (is monad-fail ?return ?bind ?fail)*

<proof>

notepad begin

catch cannot be lifted through the state monad according to *monad-catch-state* because there is now way to communicate the state updates to the handler.

<proof>

end

end

3.8.3 Reader

context

fixes *ask* :: (*'r, 'm*) *ask*

begin

definition *ask-state* :: (*'r, ('s, 'm) stateT*) *ask*

where *ask-state f* = *StateT* ($\lambda s. \ ask \ (\lambda r. \ run-state \ (f \ r) \ s)$)

lemma *run-ask-state* [*simp*]:
 $run\text{-}state\ (ask\text{-}state\ f)\ s = ask\ (\lambda r.\ run\text{-}state\ (f\ r)\ s)$
 ⟨*proof*⟩

lemma *monad-reader-stateT* [*locale-witness*]:
assumes *monad-reader return bind ask*
shows *monad-reader return-state bind-state ask-state*
 ⟨*proof*⟩

lemma *monad-reader-state-stateT* [*locale-witness*]:
assumes *monad-reader return bind ask*
shows *monad-reader-state return-state bind-state ask-state get-state put-state*
 ⟨*proof*⟩

end

3.8.4 Probability

definition *altc-sample-state* :: $('x \Rightarrow ('b \Rightarrow 'm) \Rightarrow 'm) \Rightarrow 'x \Rightarrow ('b \Rightarrow ('s, 'm) stateT) \Rightarrow ('s, 'm) stateT$
where *altc-sample-state altc-sample p f* = *StateT* $(\lambda s.\ altc\text{-}sample\ p\ (\lambda x.\ run\text{-}state\ (f\ x)\ s))$

lemma *run-altc-sample-state* [*simp*]:
 $run\text{-}state\ (altc\text{-}sample\text{-}state\ altc\text{-}sample\ p\ f)\ s = altc\text{-}sample\ p\ (\lambda x.\ run\text{-}state\ (f\ x)\ s)$
 ⟨*proof*⟩

context
fixes *sample* :: $('p, 'm)\ sample$
begin

abbreviation *sample-state* :: $('p, ('s, 'm)\ stateT)\ sample$ **where**
sample-state $\equiv altc\text{-}sample\text{-}state\ sample$

context
assumes *monad-prob return bind sample*
begin

interpretation *monad-prob return bind sample* ⟨*proof*⟩

lemma *monad-prob-stateT* [*locale-witness*]: *monad-prob return-state bind-state sample-state*
including *lifting-syntax*
 ⟨*proof*⟩

lemma *monad-state-prob-stateT* [*locale-witness*]:
monad-state-prob return-state bind-state get-state put-state sample-state
 ⟨*proof*⟩

end

end

3.8.5 Writer

context

fixes $tell :: ('w, 'm) \text{ tell}$

begin

definition $tell\text{-state} :: ('w, ('s, 'm) \text{ stateT}) \text{ tell}$

where $tell\text{-state } w \ m = \text{StateT } (\lambda s. \text{tell } w \ (\text{run-state } m \ s))$

lemma $run\text{-tell-state}$ [*simp*]: $run\text{-state } (tell\text{-state } w \ m) \ s = \text{tell } w \ (\text{run-state } m \ s)$
(*proof*)

lemma $monad\text{-writer-stateT}$ [*locale-witness*]:

assumes $monad\text{-writer return bind tell}$

shows $monad\text{-writer return-state bind-state tell-state}$

(*proof*)

end

3.8.6 Binary Non-determinism

context

fixes $alt :: 'm \text{ alt}$

begin

definition $alt\text{-state} :: ('s, 'm) \text{ stateT } alt$

where $alt\text{-state } m1 \ m2 = \text{StateT } (\lambda s. \text{alt } (\text{run-state } m1 \ s) \ (\text{run-state } m2 \ s))$

lemma $run\text{-alt-state}$ [*simp*]: $run\text{-state } (alt\text{-state } m1 \ m2) \ s = \text{alt } (\text{run-state } m1 \ s) \ (\text{run-state } m2 \ s)$
(*proof*)

context **assumes** $monad\text{-alt return bind alt}$ **begin**

interpretation $monad\text{-alt return bind alt}$ (*proof*)

lemma $monad\text{-alt-stateT}$ [*locale-witness*]: $monad\text{-alt return-state bind-state alt-state}$
(*proof*)

lemma $monad\text{-state-alt-stateT}$ [*locale-witness*]:

$monad\text{-state-alt return-state bind-state get-state put-state alt-state}$

(*proof*)

end

lemma *monad-fail-alt-stateT* [*locale-witness*]:
 fixes *fail*
 assumes *monad-fail-alt return bind fail alt*
 shows *monad-fail-alt return-state bind-state (fail-state fail) alt-state*
 ⟨*proof*⟩

end

3.8.7 Countable Non-determinism

context
 fixes *altc* :: (*'c*, *'m*) *altc*
begin

abbreviation *altc-state* :: (*'c*, (*'s*, *'m*) *stateT*) *altc*
where *altc-state* \equiv *altc-sample-state altc*

context
 includes *lifting-syntax*
 assumes *monad-altc return bind altc*
begin

interpretation *monad-altc return bind altc* ⟨*proof*⟩

lemma *monad-altc-stateT* [*locale-witness*]: *monad-altc return-state bind-state altc-state*
 ⟨*proof*⟩

lemma *monad-state-altc-stateT* [*locale-witness*]:
monad-state-altc return-state bind-state get-state put-state altc-state
 ⟨*proof*⟩

end

lemma *monad-altc3-stateT* [*locale-witness*]:
 assumes *monad-altc3 return bind altc*
 shows *monad-altc3 return-state bind-state altc-state*
 ⟨*proof*⟩

end

3.8.8 Resumption

context
 fixes *pause* :: (*'o*, *'i*, *'m*) *pause*
begin

definition *pause-state* :: (*'o*, *'i*, (*'s*, *'m*) *stateT*) *pause*
where *pause-state out c* = *StateT* ($\lambda s.$ *pause out* ($\lambda i.$ *run-state* (*c i*) *s*))

lemma *run-pause-state* [*simp*]:

run-state (pause-state out c) s = pause out (λi. run-state (c i) s)
⟨proof⟩

lemma *monad-resumption-stateT [locale-witness]:*
 assumes *monad-resumption return bind pause*
 shows *monad-resumption return-state bind-state pause-state*
⟨proof⟩

end

end

3.8.9 Parametricity

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

lemma *return-state-parametric [transfer-rule]:*
 $((rel\text{-}prod\ A\ S\ ==>\ M)\ ==>\ A\ ==>\ rel\text{-}stateT\ S\ M)\ return\text{-}state\ re\text{-}turn\text{-}state$
⟨proof⟩

lemma *bind-state-parametric [transfer-rule]:*
 $((M\ ==>\ (rel\text{-}prod\ A\ S\ ==>\ M)\ ==>\ M)\ ==>\ rel\text{-}stateT\ S\ M\ ==>\ (A\ ==>\ rel\text{-}stateT\ S\ M)\ ==>\ rel\text{-}stateT\ S\ M)$
 bind-state bind-state
⟨proof⟩

lemma *get-state-parametric [transfer-rule]:*
 $((S\ ==>\ rel\text{-}stateT\ S\ M)\ ==>\ rel\text{-}stateT\ S\ M)\ get\text{-}state\ get\text{-}state$
⟨proof⟩

lemma *put-state-parametric [transfer-rule]:*
 $(S\ ==>\ rel\text{-}stateT\ S\ M\ ==>\ rel\text{-}stateT\ S\ M)\ put\text{-}state\ put\text{-}state$
⟨proof⟩

lemma *fail-state-parametric [transfer-rule]:* $(M\ ==>\ rel\text{-}stateT\ S\ M)\ fail\text{-}state\ fail\text{-}state$
⟨proof⟩

lemma *ask-state-parametric [transfer-rule]:*
 $((R\ ==>\ M)\ ==>\ M)\ ==>\ (R\ ==>\ rel\text{-}stateT\ S\ M)\ ==>\ rel\text{-}stateT\ S\ M)\ ask\text{-}state\ ask\text{-}state$
⟨proof⟩

lemma *altc-sample-state-parametric [transfer-rule]:*
 $((X\ ==>\ (P\ ==>\ M)\ ==>\ M)\ ==>\ X\ ==>\ (P\ ==>\ rel\text{-}stateT\ S\ M)\ ==>\ rel\text{-}stateT\ S\ M)$
 altc-sample-state altc-sample-state
⟨proof⟩

lemma *tell-state-parametric* [*transfer-rule*]:
 $((W \text{====>} M \text{====>} M) \text{====>} W \text{====>} \text{rel-stateT } S \ M \text{====>} \text{rel-stateT } S \ M)$
tell-state tell-state
 <proof>

lemma *alt-state-parametric* [*transfer-rule*]:
 $((M \text{====>} M \text{====>} M) \text{====>} \text{rel-stateT } S \ M \text{====>} \text{rel-stateT } S \ M \text{====>} \text{rel-stateT } S \ M)$
alt-state alt-state
 <proof>

lemma *pause-state-parametric* [*transfer-rule*]:
 $((Out \text{====>} (In \text{====>} M) \text{====>} M) \text{====>} Out \text{====>} (In \text{====>} \text{rel-stateT } S \ M) \text{====>} \text{rel-stateT } S \ M)$
pause-state pause-state
 <proof>

end

3.9 Writer monad transformer

We implement a simple writer monad which collects all the output in a list. It would also have been possible to use a monoid instead. The phantom type variables *'a* and *'w* are needed to avoid hidden polymorphism when overloading the monad operations for the writer monad transformer.

datatype (*'w*, *'a*, *'m*) *writerT* = *WriterT* (*run-writer*: *'m*)

context

fixes *return* :: (*'a* × *'w list*, *'m*) *return*

and *bind* :: (*'a* × *'w list*, *'m*) *bind*

begin

definition *return-writer* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *return*

where *return-writer* *x* = *WriterT* (*return* (*x*, []))

definition *bind-writer* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *bind*

where *bind-writer* *m f* = *WriterT* (*bind* (*run-writer* *m*) ($\lambda(a, ws). \text{bind} (\text{run-writer } (f \ a)) (\lambda(b, ws'). \text{return } (b, ws \ @ \ ws'))$))

definition *tell-writer* :: (*'w*, (*'w*, *'a*, *'m*) *writerT*) *tell*

where *tell-writer* *w m* = *WriterT* (*bind* (*run-writer* *m*) ($\lambda(a, ws). \text{return } (a, w \ # \ ws)$))

lemma *run-return-writer* [*simp*]: *run-writer* (*return-writer* *x*) = *return* (*x*, [])

<proof>

lemma *run-bind-writer* [*simp*]:
 $run\text{-}writer\ (bind\text{-}writer\ m\ f) = bind\ (run\text{-}writer\ m)\ (\lambda(a, ws). bind\ (run\text{-}writer\ (f\ a))\ (\lambda(b, ws'). return\ (b, ws\ @\ ws')))$
<proof>

lemma *run-tell-writer* [*simp*]:
 $run\text{-}writer\ (tell\text{-}writer\ w\ m) = bind\ (run\text{-}writer\ m)\ (\lambda(a, ws). return\ (a, w\ \#\ ws))$
<proof>

context
assumes *monad return bind*
begin

interpretation *monad return bind* *<proof>*

lemma *monad-writerT* [*locale-witness*]: *monad return-writer bind-writer*
<proof>

lemma *monad-writer-writerT* [*locale-witness*]: *monad-writer return-writer bind-writer tell-writer*
<proof>

end

3.9.1 Failure

context
fixes *fail* :: *'m fail*
begin

definition *fail-writer* :: (*'w, 'a, 'm*) *writerT fail*
where *fail-writer* = *WriterT fail*

lemma *run-fail-writer* [*simp*]: *run-writer fail-writer = fail*
<proof>

lemma *monad-fail-writerT* [*locale-witness*]:
assumes *monad-fail return bind fail*
shows *monad-fail return-writer bind-writer fail-writer*
<proof>

Just like for the state monad, we cannot lift *catch* because the output before the failure would be lost.

3.9.2 State

context
fixes *get* :: (*'s, 'm*) *get*

and $put :: ('s, 'm) put$
begin

definition $get-writer :: ('s, ('w, 'a, 'm) writerT) get$
where $get-writer f = WriterT (get (\lambda s. run-writer (f s)))$

definition $put-writer :: ('s, ('w, 'a, 'm) writerT) put$
where $put-writer s m = WriterT (put s (run-writer m))$

lemma $run-get-writer [simp]: run-writer (get-writer f) = get (\lambda s. run-writer (f s))$
 $\langle proof \rangle$

lemma $run-put-writer [simp]: run-writer (put-writer s m) = put s (run-writer m)$
 $\langle proof \rangle$

lemma $monad-state-writerT [locale-witness]:$
assumes $monad-state return bind get put$
shows $monad-state return-writer bind-writer get-writer put-writer$
 $\langle proof \rangle$

3.9.3 Probability

definition $altc-sample-writer :: ('x \Rightarrow ('b \Rightarrow 'm) \Rightarrow 'm) \Rightarrow 'x \Rightarrow ('b \Rightarrow ('w, 'a, 'm) writerT) \Rightarrow ('w, 'a, 'm) writerT$
where $altc-sample-writer altc-sample p f = WriterT (altc-sample p (\lambda p. run-writer (f p)))$

lemma $run-altc-sample-writer [simp]:$
 $run-writer (altc-sample-writer altc-sample p f) = altc-sample p (\lambda p. run-writer (f p))$
 $\langle proof \rangle$

context
fixes $sample :: ('p, 'm) sample$
begin

abbreviation $sample-writer :: ('p, ('w, 'a, 'm) writerT) sample$
where $sample-writer \equiv altc-sample-writer sample$

lemma $monad-prob-writerT [locale-witness]:$
assumes $monad-prob return bind sample$
shows $monad-prob return-writer bind-writer sample-writer$
 $\langle proof \rangle$ **including** $lifting-syntax$
 $\langle proof \rangle$

lemma $monad-state-prob-writerT [locale-witness]:$
assumes $monad-state-prob return bind get put sample$

shows *monad-state-prob return-writer bind-writer get-writer put-writer sample-writer*
<proof>

end

3.9.4 Reader

context

fixes *ask* :: ('r, 'm) *ask*

begin

definition *ask-writer* :: ('r, ('w, 'a, 'm) *writerT*) *ask*

where *ask-writer* *f* = *WriterT* (*ask* (λr . *run-writer* (*f* *r*)))

lemma *run-ask-writer* [*simp*]: *run-writer* (*ask-writer* *f*) = *ask* (λr . *run-writer* (*f* *r*))

<proof>

lemma *monad-reader-writerT* [*locale-witness*]:

assumes *monad-reader return bind ask*

shows *monad-reader return-writer bind-writer ask-writer*

<proof>

lemma *monad-reader-state-writerT* [*locale-witness*]:

assumes *monad-reader-state return bind ask get put*

shows *monad-reader-state return-writer bind-writer ask-writer get-writer put-writer*

<proof>

end

3.9.5 Resumption

context

fixes *pause* :: ('o, 'i, 'm) *pause*

begin

definition *pause-writer* :: ('o, 'i, ('w, 'a, 'm) *writerT*) *pause*

where *pause-writer* *out* *c* = *WriterT* (*pause* *out* ($\lambda input$. *run-writer* (*c* *input*)))

lemma *run-pause-writer* [*simp*]:

run-writer (*pause-writer* *out* *c*) = *pause* *out* ($\lambda input$. *run-writer* (*c* *input*))

<proof>

lemma *monad-resumption-writerT* [*locale-witness*]:

assumes *monad-resumption return bind pause*

shows *monad-resumption return-writer bind-writer pause-writer*

<proof>

end

3.9.6 Binary Non-determinism

context

fixes $alt :: 'm \text{ alt}$

begin

definition $alt\text{-}writer :: ('w, 'a, 'm) \text{ writer}T \text{ alt}$

where $alt\text{-}writer \ m \ m' = \text{Writer}T \ (alt \ (run\text{-}writer \ m) \ (run\text{-}writer \ m'))$

lemma $run\text{-}alt\text{-}writer \ [simp]: \ run\text{-}writer \ (alt\text{-}writer \ m \ m') = alt \ (run\text{-}writer \ m) \ (run\text{-}writer \ m')$

$\langle proof \rangle$

lemma $monad\text{-}alt\text{-}writerT \ [locale\text{-}witness]:$

assumes $monad\text{-}alt \ return \ bind \ alt$

shows $monad\text{-}alt \ return\text{-}writer \ bind\text{-}writer \ alt\text{-}writer$

$\langle proof \rangle$

lemma $monad\text{-}fail\text{-}alt\text{-}writerT \ [locale\text{-}witness]:$

assumes $monad\text{-}fail\text{-}alt \ return \ bind \ fail \ alt$

shows $monad\text{-}fail\text{-}alt \ return\text{-}writer \ bind\text{-}writer \ fail\text{-}writer \ alt\text{-}writer$

$\langle proof \rangle$

lemma $monad\text{-}state\text{-}alt\text{-}writerT \ [locale\text{-}witness]:$

assumes $monad\text{-}state\text{-}alt \ return \ bind \ get \ put \ alt$

shows $monad\text{-}state\text{-}alt \ return\text{-}writer \ bind\text{-}writer \ get\text{-}writer \ put\text{-}writer \ alt\text{-}writer$

$\langle proof \rangle$

end

3.9.7 Countable Non-determinism

context

fixes $altc :: ('c, 'm) \text{ altc}$

begin

abbreviation $altc\text{-}writer :: ('c, ('w, 'a, 'm) \text{ writer}T) \text{ altc}$

where $altc\text{-}writer \equiv altc\text{-}sample\text{-}writer \ altc$

lemma $monad\text{-}altc\text{-}writerT \ [locale\text{-}witness]:$

assumes $monad\text{-}altc \ return \ bind \ altc$

shows $monad\text{-}altc \ return\text{-}writer \ bind\text{-}writer \ altc\text{-}writer$

$\langle proof \rangle$ **including** $lifting\text{-}syntax$

$\langle proof \rangle$

lemma $monad\text{-}altc3\text{-}writerT \ [locale\text{-}witness]:$

assumes $monad\text{-}altc3 \ return \ bind \ altc$

shows $monad\text{-}altc3 \ return\text{-}writer \ bind\text{-}writer \ altc\text{-}writer$

$\langle proof \rangle$

lemma *monad-state-altc-writerT* [*locale-witness*]:
assumes *monad-state-altc return bind get put altc*
shows *monad-state-altc return-writer bind-writer get-writer put-writer altc-writer*
<proof>

end

end

end

end

3.9.8 Parametricity

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

lemma *return-writer-parametric* [*transfer-rule*]:
 $((rel\text{-}prod\ A\ (list\text{-}all2\ W) ==> M) ==> A ==> rel\text{-}writerT\ W\ A\ M)$
return-writer return-writer
<proof>

lemma *bind-writer-parametric* [*transfer-rule*]:
 $((rel\text{-}prod\ A\ (list\text{-}all2\ W) ==> M) ==> (M ==> (rel\text{-}prod\ A\ (list\text{-}all2\ W) ==> M) ==> M) ==> rel\text{-}writerT\ W\ A\ M ==> (A ==> rel\text{-}writerT\ W\ A\ M) ==> rel\text{-}writerT\ W\ A\ M)$
bind-writer bind-writer
<proof>

lemma *tell-writer-parametric* [*transfer-rule*]:
 $((rel\text{-}prod\ A\ (list\text{-}all2\ W) ==> M) ==> (M ==> (rel\text{-}prod\ A\ (list\text{-}all2\ W) ==> M) ==> M) ==> W ==> rel\text{-}writerT\ W\ A\ M ==> rel\text{-}writerT\ W\ A\ M)$
tell-writer tell-writer
<proof>

lemma *ask-writer-parametric* [*transfer-rule*]:
 $((R ==> M) ==> M) ==> (R ==> rel\text{-}writerT\ W\ A\ M) ==> rel\text{-}writerT\ W\ A\ M)$
ask-writer ask-writer
<proof>

lemma *fail-writer-parametric* [*transfer-rule*]:
 $(M ==> rel\text{-}writerT\ W\ A\ M)$
fail-writer fail-writer
<proof>

lemma *get-writer-parametric* [*transfer-rule*]:
 $((S ==> M) ==> M) ==> (S ==> rel\text{-}writerT\ W\ A\ M) ==>$

rel-writerT W A M) get-writer get-writer
 ⟨proof⟩

lemma *put-writer-parametric [transfer-rule]*:
 ((*S* ==> *M* ==> *M*) ==> *S* ==> *rel-writerT W A M* ==> *rel-writerT W A M*) *put-writer put-writer*
 ⟨proof⟩

lemma *altc-sample-writer-parametric [transfer-rule]*:
 ((*X* ==> (*P* ==> *M*) ==> *M*) ==> *X* ==> (*P* ==> *rel-writerT W A M*) ==> *rel-writerT W A M*)
altc-sample-writer altc-sample-writer
 ⟨proof⟩

lemma *alt-writer-parametric [transfer-rule]*:
 ((*M* ==> *M* ==> *M*) ==> *rel-writerT W A M* ==> *rel-writerT W A M*)
alt-writer alt-writer
 ⟨proof⟩

lemma *pause-writer-parametric [transfer-rule]*:
 ((*Out* ==> (*In* ==> *M*) ==> *M*) ==> *Out* ==> (*In* ==> *rel-writerT W A M*) ==> *rel-writerT W A M*)
pause-writer pause-writer
 ⟨proof⟩

end

3.10 Continuation monad transformer

datatype (*'a*, *'m*) *contT* = *ContT* (*run-cont*: (*'a* ⇒ *'m*) ⇒ *'m*)

3.10.1 CallCC

type-synonym (*'a*, *'m*) *calcc* = ((*'a* ⇒ *'m*) ⇒ *'m*) ⇒ *'m*

definition *calcc-cont* :: (*'a*, (*'a*, *'m*) *contT*) *calcc*
where *calcc-cont* *f* = *ContT* ($\lambda k. \text{run-cont } (f (\lambda x. \text{ContT } (\lambda -. k x))) k$)

lemma *run-calcc-cont [simp]*: *run-cont* (*calcc-cont* *f*) *k* = *run-cont* (*f* ($\lambda x. \text{ContT } (\lambda -. k x)$)) *k*
 ⟨proof⟩

3.10.2 Plain monad

definition *return-cont* :: (*'a*, (*'a*, *'m*) *contT*) *return*
where *return-cont* *x* = *ContT* ($\lambda k. k x$)

definition *bind-cont* :: (*'a*, (*'a*, *'m*) *contT*) *bind*
where *bind-cont* *m f* = *ContT* ($\lambda k. \text{run-cont } m (\lambda x. \text{run-cont } (f x) k)$)

lemma *run-return-cont* [*simp*]: *run-cont* (*return-cont* *x*) *k* = *k* *x*
⟨*proof*⟩

lemma *run-bind-cont* [*simp*]: *run-cont* (*bind-cont* *m* *f*) *k* = *run-cont* *m* ($\lambda x.$ *run-cont* (*f* *x*) *k*)
⟨*proof*⟩

lemma *monad-contT* [*locale-witness*]: *monad* *return-cont* *bind-cont* (**is monad** ?*return* ?*bind*)
⟨*proof*⟩

3.10.3 Failure

context

fixes *fail* :: 'm *fail*

begin

definition *fail-cont* :: ('a, 'm) *contT* *fail*
where *fail-cont* = *ContT* ($\lambda.$ *fail*)

lemma *run-fail-cont* [*simp*]: *run-cont* *fail-cont* *k* = *fail*
⟨*proof*⟩

lemma *monad-fail-contT* [*locale-witness*]: *monad-fail* *return-cont* *bind-cont* *fail-cont*
⟨*proof*⟩

end

3.10.4 State

context

fixes *get* :: ('s, 'm) *get*

and *put* :: ('s, 'm) *put*

begin

definition *get-cont* :: ('s, ('a, 'm) *contT*) *get*
where *get-cont* *f* = *ContT* ($\lambda k.$ *get* ($\lambda s.$ *run-cont* (*f* *s*) *k*))

definition *put-cont* :: ('s, ('a, 'm) *contT*) *put*
where *put-cont* *s* *m* = *ContT* ($\lambda k.$ *put* *s* (*run-cont* *m* *k*))

lemma *run-get-cont* [*simp*]: *run-cont* (*get-cont* *f*) *k* = *get* ($\lambda s.$ *run-cont* (*f* *s*) *k*)
⟨*proof*⟩

lemma *run-put-cont* [*simp*]: *run-cont* (*put-cont* *s* *m*) *k* = *put* *s* (*run-cont* *m* *k*)
⟨*proof*⟩

lemma *monad-state-contT* [*locale-witness*]:

assumes *monad-state return' bind' get put* — We don't need the plain monad operations for lifting.

shows *monad-state return-cont bind-cont get-cont (put-cont :: ('s, ('a, 'm) contT) put)*

(**is** *monad-state ?return ?bind ?get ?put*)

<proof>

end

4 Locales for monad homomorphisms

locale *monad-hom = m1: monad return1 bind1 +*

m2: monad return2 bind2

for *return1 :: ('a, 'm1) return*

and *bind1 :: ('a, 'm1) bind*

and *return2 :: ('a, 'm2) return*

and *bind2 :: ('a, 'm2) bind*

and *h :: 'm1 \Rightarrow 'm2*

+

assumes *hom-return: $\bigwedge x. h (return1 x) = return2 x$*

and *hom-bind: $\bigwedge x f. h (bind1 x f) = bind2 (h x) (h \circ f)$*

begin

lemma *hom-lift [simp]: $h (m1.lift f m) = m2.lift f (h m)$*

<proof>

end

locale *monad-state-hom = m1: monad-state return1 bind1 get1 put1 +*

m2: monad-state return2 bind2 get2 put2 +

monad-hom return1 bind1 return2 bind2 h

for *return1 :: ('a, 'm1) return*

and *bind1 :: ('a, 'm1) bind*

and *get1 :: ('s, 'm1) get*

and *put1 :: ('s, 'm1) put*

and *return2 :: ('a, 'm2) return*

and *bind2 :: ('a, 'm2) bind*

and *get2 :: ('s, 'm2) get*

and *put2 :: ('s, 'm2) put*

and *h :: 'm1 \Rightarrow 'm2*

+

assumes *hom-get [simp]: $h (get1 f) = get2 (h \circ f)$*

and *hom-put [simp]: $h (put1 s m) = put2 s (h m)$*

locale *monad-fail-hom = m1: monad-fail return1 bind1 fail1 +*

m2: monad-fail return2 bind2 fail2 +

monad-hom return1 bind1 return2 bind2 h

for *return1 :: ('a, 'm1) return*

and *bind1 :: ('a, 'm1) bind*

```

and fail1 :: 'm1 fail
and return2 :: ('a, 'm2) return
and bind2 :: ('a, 'm2) bind
and fail2 :: 'm2 fail
and h :: 'm1 ⇒ 'm2
+
assumes hom-fail [simp]: h fail1 = fail2

locale monad-catch-hom = m1: monad-catch return1 bind1 fail1 catch1 +
  m2: monad-catch return2 bind2 fail2 catch2 +
  monad-fail-hom return1 bind1 fail1 return2 bind2 fail2 h
for return1 :: ('a, 'm1) return
and bind1 :: ('a, 'm1) bind
and fail1 :: 'm1 fail
and catch1 :: 'm1 catch
and return2 :: ('a, 'm2) return
and bind2 :: ('a, 'm2) bind
and fail2 :: 'm2 fail
and catch2 :: 'm2 catch
and h :: 'm1 ⇒ 'm2
+
assumes hom-catch [simp]: h (catch1 m1 m2) = catch2 (h m1) (h m2)

locale monad-reader-hom = m1: monad-reader return1 bind1 ask1 +
  m2: monad-reader return2 bind2 ask2 +
  monad-hom return1 bind1 return2 bind2 h
for return1 :: ('a, 'm1) return
and bind1 :: ('a, 'm1) bind
and ask1 :: ('r, 'm1) ask
and return2 :: ('a, 'm2) return
and bind2 :: ('a, 'm2) bind
and ask2 :: ('r, 'm2) ask
and h :: 'm1 ⇒ 'm2
+
assumes hom-ask [simp]: h (ask1 f) = ask2 (h ∘ f)

locale monad-prob-hom = m1: monad-prob return1 bind1 sample1 +
  m2: monad-prob return2 bind2 sample2 +
  monad-hom return1 bind1 return2 bind2 h
for return1 :: ('a, 'm1) return
and bind1 :: ('a, 'm1) bind
and sample1 :: ('p, 'm1) sample
and return2 :: ('a, 'm2) return
and bind2 :: ('a, 'm2) bind
and sample2 :: ('p, 'm2) sample
and h :: 'm1 ⇒ 'm2
+
assumes hom-sample [simp]: h (sample1 p f) = sample2 p (h ∘ f)

```

locale *monad-alt-hom* = *m1*: *monad-alt* *return1* *bind1* *alt1* +
m2: *monad-alt* *return2* *bind2* *alt2* +
monad-hom *return1* *bind1* *return2* *bind2* *h*
for *return1* :: ('a, 'm1) *return*
and *bind1* :: ('a, 'm1) *bind*
and *alt1* :: 'm1 *alt*
and *return2* :: ('a, 'm2) *return*
and *bind2* :: ('a, 'm2) *bind*
and *alt2* :: 'm2 *alt*
and *h* :: 'm1 \Rightarrow 'm2
+
assumes *hom-alt* [*simp*]: $h (alt1\ m\ m') = alt2 (h\ m) (h\ m')$

locale *monad-altc-hom* = *m1*: *monad-altc* *return1* *bind1* *altc1* +
m2: *monad-altc* *return2* *bind2* *altc2* +
monad-hom *return1* *bind1* *return2* *bind2* *h*
for *return1* :: ('a, 'm1) *return*
and *bind1* :: ('a, 'm1) *bind*
and *altc1* :: ('c, 'm1) *altc*
and *return2* :: ('a, 'm2) *return*
and *bind2* :: ('a, 'm2) *bind*
and *altc2* :: ('c, 'm2) *altc*
and *h* :: 'm1 \Rightarrow 'm2
+
assumes *hom-altc* [*simp*]: $h (altc1\ C\ f) = altc2\ C (h\ o\ f)$

locale *monad-writer-hom* = *m1*: *monad-writer* *return1* *bind1* *tell1* +
m2: *monad-writer* *return2* *bind2* *tell2* +
monad-hom *return1* *bind1* *return2* *bind2* *h*
for *return1* :: ('a, 'm1) *return*
and *bind1* :: ('a, 'm1) *bind*
and *tell1* :: ('w, 'm1) *tell*
and *return2* :: ('a, 'm2) *return*
and *bind2* :: ('a, 'm2) *bind*
and *tell2* :: ('w, 'm2) *tell*
and *h* :: 'm1 \Rightarrow 'm2
+
assumes *hom-tell* [*simp*]: $h (tell1\ w\ m) = tell2\ w (h\ m)$

locale *monad-resumption-hom* = *m1*: *monad-resumption* *return1* *bind1* *pause1* +
m2: *monad-resumption* *return2* *bind2* *pause2* +
monad-hom *return1* *bind1* *return2* *bind2* *h*
for *return1* :: ('a, 'm1) *return*
and *bind1* :: ('a, 'm1) *bind*
and *pause1* :: ('o, 'i, 'm1) *pause*
and *return2* :: ('a, 'm2) *return*
and *bind2* :: ('a, 'm2) *bind*
and *pause2* :: ('o, 'i, 'm2) *pause*
and *h* :: 'm1 \Rightarrow 'm2

+
assumes *hom-pause* [*simp*]: $h (\text{pause1 out } c) = \text{pause2 out } (h \circ c)$

5 Switching between monads

Homomorphisms are functional relations between monads. In general, it is more convenient to use arbitrary relations as embeddings because arbitrary relations allow us to change the type of values in a monad. As different monad transformers change the value type in different ways, the embeddings must also support such changes in values.

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

5.1 Embedding Identity into Probability

named-theorems *cr-id-prob-transfer*

definition *prob-of-id* :: 'a id \Rightarrow 'a prob **where**
prob-of-id m = *return-pmf* (*extract* m)

lemma *monad-id-prob-hom* [*locale-witness*]:
monad-hom *return-id* *bind-id* *return-pmf* *bind-pmf* *prob-of-id*
 ⟨*proof*⟩

inductive *cr-id-prob* :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a id \Rightarrow 'b prob \Rightarrow bool **for** A
where A x y \Longrightarrow *cr-id-prob* A (*return-id* x) (*return-pmf* y)

inductive-simps *cr-id-prob-simps* [*simp*]: *cr-id-prob* A (*return-id* x) (*return-pmf* y)

lemma *cr-id-prob-return* [*cr-id-prob-transfer*]: (A \Longrightarrow *cr-id-prob* A) *return-id* *return-pmf*
 ⟨*proof*⟩

lemma *cr-id-prob-bind* [*cr-id-prob-transfer*]:
 (*cr-id-prob* A \Longrightarrow (A \Longrightarrow *cr-id-prob* B) \Longrightarrow *cr-id-prob* B) *bind-id* *bind-pmf*
 ⟨*proof*⟩

lemma *cr-id-prob-Grp*: *cr-id-prob* (BNF-Def.Grp A f) = BNF-Def.Grp {x. *set-id* x \subseteq A} (*return-pmf* \circ f \circ *extract*)
 ⟨*proof*⟩

5.2 State and Reader

When no state updates are needed, the operation *get* can be replaced by *ask*.

named-theorems *cr-envT-stateT-transfer*

definition *cr-prod1* :: 'c ⇒ ('a ⇒ 'b ⇒ bool) ⇒ 'a ⇒ 'b × 'c ⇒ bool
where *cr-prod1 c' A* = (λa (b, c). A a b ∧ c' = c)

lemma *cr-prod1-simps* [*simp*]: *cr-prod1 c' A a (b, c) ⟷ A a b ∧ c' = c*
⟨*proof*⟩

lemma *cr-prod1I*: *A a b ⟹ cr-prod1 c' A a (b, c')* ⟨*proof*⟩

lemma *cr-prod1-Pair-transfer* [*cr-envT-stateT-transfer*]: (*A ⟹ eq-onp ((=) c) ⟹ cr-prod1 c A*) (λa -. a) *Pair*
⟨*proof*⟩

lemma *cr-prod1-fst-transfer* [*cr-envT-stateT-transfer*]: (*cr-prod1 c A ⟹ A*)
(λa. a) *fst*
⟨*proof*⟩

lemma *cr-prod1-case-prod-transfer* [*cr-envT-stateT-transfer*]:
(*A ⟹ eq-onp ((=) c) ⟹ C*) ⟹ *cr-prod1 c A ⟹ C*) (λf a. f a
case-prod
⟨*proof*⟩

lemma *cr-prod1-Grp*: *cr-prod1 c (BNF-Def.Grp A f) = BNF-Def.Grp A* (λb. (f
b, c))
⟨*proof*⟩

definition *cr-envT-stateT* :: 's ⇒ ('m1 ⇒ 'm2 ⇒ bool) ⇒ ('s, 'm1) *envT* ⇒ ('s,
'm2) *stateT* ⇒ bool
where *cr-envT-stateT s M m1 m2* = (*eq-onp ((=) s) ⟹ M*) (*run-env m1*)
(*run-state m2*)

lemma *cr-envT-stateT-simps* [*simp*]:
cr-envT-stateT s M (EnvT f) (StateT g) ⟷ M (f s) (g s)
⟨*proof*⟩

lemma *cr-envT-stateTE*:
assumes *cr-envT-stateT s M m1 m2*
obtains *f g* **where** *m1 = EnvT f m2 = StateT g (eq-onp ((=) s) ⟹ M) f g*
⟨*proof*⟩

lemma *cr-envT-stateTD*: *cr-envT-stateT s M m1 m2 ⟹ M (run-env m1 s)*
(*run-state m2 s*)
⟨*proof*⟩

lemma *cr-envT-stateT-run* [*cr-envT-stateT-transfer*]:
(*cr-envT-stateT s M ⟹ eq-onp ((=) s) ⟹ M*) *run-env run-state*
⟨*proof*⟩

lemma *cr-envT-stateT-StateT-EnvT* [*cr-envT-stateT-transfer*]:
 $((eq-onp ((=) s) ==> M) ==> cr-envT-stateT s M) EnvT StateT$
 <proof>

lemma *cr-envT-stateT-rec* [*cr-envT-stateT-transfer*]:
 $((eq-onp ((=) s) ==> M) ==> C) ==> cr-envT-stateT s M ==> C$
rec-envT rec-stateT
 <proof>

lemma *cr-envT-stateT-return* [*cr-envT-stateT-transfer*]:
notes [*transfer-rule*] = *cr-envT-stateT-transfer* **shows**
 $((cr-prod1 s A ==> M) ==> A ==> cr-envT-stateT s M) return-env$
return-state
 <proof>

lemma *cr-envT-stateT-bind* [*cr-envT-stateT-transfer*]:
 $((M ==> (cr-prod1 s A ==> M) ==> M) ==> cr-envT-stateT s M$
 $==> (A ==> cr-envT-stateT s M) ==> cr-envT-stateT s M)$
bind-env bind-state
 <proof>

lemma *cr-envT-stateT-ask-get* [*cr-envT-stateT-transfer*]:
 $((eq-onp ((=) s) ==> cr-envT-stateT s M) ==> cr-envT-stateT s M) ask-env$
get-state
 <proof>

lemma *cr-envT-stateT-fail* [*cr-envT-stateT-transfer*]:
notes [*transfer-rule*] = *cr-envT-stateT-transfer* **shows**
 $(M ==> cr-envT-stateT s M) fail-env fail-state$
 <proof>

5.3 - *spmf* and $(-, - prob) optionT$

This section defines the mapping between the *- spmf* monad and the monad obtained by composing transforming *- prob* with $(-, -) optionT$.

definition *cr-spmf-prob-optionT* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a, 'a option prob) optionT$
 $\Rightarrow 'b spmf \Rightarrow bool$

where *cr-spmf-prob-optionT* $A p q \iff rel-spmf A (run-option p) q$

lemma *cr-spmf-prob-optionTI*: $rel-spmf A (run-option p) q \implies cr-spmf-prob-optionT$
 $A p q$
 <proof>

lemma *cr-spmf-prob-optionTD*: $cr-spmf-prob-optionT A p q \implies rel-spmf A (run-option$
 $p) q$
 <proof>

lemma *cr-spmf-prob-optionT-return-transfer*:

— Cannot be used as a transfer rule in *transfer-prover* because *return-spmf* is not a constant.

($A \text{ ===> } cr\text{-spmof-prob-optionT } A$) (*return-option return-pmf*) *return-spmf*
 $\langle proof \rangle$

lemma *cr-spmof-prob-optionT-bind-transfer*:

($cr\text{-spmof-prob-optionT } A \text{ ===> } (A \text{ ===> } cr\text{-spmof-prob-optionT } A) \text{ ===> } cr\text{-spmof-prob-optionT } A$)
 (*bind-option return-pmf bind-pmf*) *bind-spmf*
 $\langle proof \rangle$

lemma *cr-spmof-prob-optionT-fail-transfer*:

$cr\text{-spmof-prob-optionT } A$ (*fail-option return-pmf*) (*return-pmf None*)
 $\langle proof \rangle$

abbreviation (*input*) *spmof-of-prob-optionT* :: ('a, 'a option prob) optionT \Rightarrow 'a *spmof*

where *spmof-of-prob-optionT* \equiv *run-option*

abbreviation (*input*) *prob-optionT-of-spmof* :: 'a *spmof* \Rightarrow ('a, 'a option prob) optionT

where *prob-optionT-of-spmof* \equiv *OptionT*

lemma *spmof-of-prob-optionT-transfer*: ($cr\text{-spmof-prob-optionT } A \text{ ===> } rel\text{-spmof } A$) *spmof-of-prob-optionT* ($\lambda x. x$)
 $\langle proof \rangle$

lemma *prob-optionT-of-spmof-transfer*: ($rel\text{-spmof } A \text{ ===> } cr\text{-spmof-prob-optionT } A$) *prob-optionT-of-spmof* ($\lambda x. x$)
 $\langle proof \rangle$

5.4 Probabilities and countable non-determinism

named-theorems *cr-prob-ndi-transfer*

context includes *cset.lifting* **begin**

interpretation *cset-nondetM* *return-id* *bind-id* *merge-id* *merge-id* $\langle proof \rangle$

lift-definition *cset-pmf* :: 'a *pmf* \Rightarrow 'a *cset* **is** *set-pmf* $\langle proof \rangle$

inductive *cr-pmf-cset* :: 'a *pmf* \Rightarrow 'a *cset* \Rightarrow bool **for** *p* **where**
cr-pmf-cset *p* (*cset-pmf* *p*)

lemma *cr-pmf-cset-Grp*: $cr\text{-pmf-cset} = BNF\text{-Def.Grp } UNIV$ *cset-pmf*
 $\langle proof \rangle$

lemma *cr-pmf-cset-return-pmf* [*cr-prob-ndi-transfer*]:
 ((=) ===> *cr-pmf-cset*) *return-pmf* *csingle*

<proof>

inductive *cr-prob-ndi* :: ('a ⇒ 'b ⇒ bool) ⇒ 'a prob ⇒ ('b, 'b cset id) nondetT ⇒ bool

for *A p B* **where**

cr-prob-ndi A p B **if** *rel-set A (set-pmf p) (rcset (extract (run-nondet B)))*

lemma *cr-prob-ndi-Grp*: *cr-prob-ndi (BNF-Def.Grp UNIV f) = BNF-Def.Grp UNIV (NondetT ∘ return-id ∘ cimage f ∘ cset-pmf)*

<proof>

lemma *cr-ndi-prob-return* [*cr-prob-ndi-transfer*]:

(A ==> cr-prob-ndi A) return-pmf return-nondet

<proof>

lemma *cr-ndi-prob-bind* [*cr-prob-ndi-transfer*]:

(cr-prob-ndi A ==> (A ==> cr-prob-ndi A) ==> cr-prob-ndi A) bind-pmf bind-nondet

<proof>

lemma *cr-ndi-prob-sample* [*cr-prob-ndi-transfer*]:

(cr-pmf-cset ==> ((=) ==> cr-prob-ndi A) ==> cr-prob-ndi A) bind-pmf altc-nondet

<proof>

end

end

end

6 Overloaded monad operations

theory *Monad-Overloading* **imports** *Monomorphic-Monad* **begin**

consts *return* :: ('a, 'm) return

consts *bind* :: ('a, 'm) bind

consts *get* :: ('s, 'm) get

consts *put* :: ('s, 'm) put

consts *fail* :: 'm fail

consts *catch* :: 'm catch

consts *ask* :: ('r, 'm) ask

consts *sample* :: ('p, 'm) sample

consts *pause* :: ('o, 'i, 'm) pause

consts *tell* :: ('w, 'm) tell

consts *alt* :: 'm alt

consts *altc* :: ('c, 'm) altc

6.1 Identity monad

overloading

$bind-id' \equiv bind :: ('a, 'a id) bind$

$return-id \equiv return :: ('a, 'a id) return$

begin

definition $bind-id' :: ('a, 'a id) bind$

where $[code-unfold, monad-unfold]: bind-id' = bind-id$

definition $return-id :: ('a, 'a id) return$

where $[code-unfold, monad-unfold]: return-id = id.return-id$

end

lemma $extract-bind' [simp]: extract (bind x f) = extract (f (extract x))$

$\langle proof \rangle$

lemma $extract-return [simp]: extract (return x) = x$

$\langle proof \rangle$

lemma $monad-id' [locale-witness]: monad return (bind :: ('a, 'a id) bind)$

$\langle proof \rangle$

lemma $monad-commute-id' [locale-witness]: monad-commute return (bind :: ('a, 'a id) bind)$

$\langle proof \rangle$

6.2 Probability monad

overloading

$return-prob \equiv return :: ('a, 'a prob) return$

$bind-prob \equiv bind :: ('a, 'a prob) bind$

$sample-prob \equiv sample :: ('p, 'a prob) sample$

begin

definition $return-prob :: ('a, 'a pmf) return$

where $[code-unfold, monad-unfold]: return-prob = return-pmf$

definition $bind-prob :: ('a, 'a prob) bind$

where $[code-unfold, monad-unfold]: bind-prob = bind-pmf$

definition $sample-prob :: ('p, 'a pmf) sample$

where $[code-unfold, monad-unfold]: sample-prob = bind-pmf$

end

lemma $monad-prob' [locale-witness]: monad return (bind :: ('a, 'a prob) bind)$

$\langle proof \rangle$

lemma *monad-commute-prob'* [*locale-witness*]: *monad-commute return (bind :: ('a, 'a prob) bind)*
 ⟨*proof*⟩

lemma *monad-prob-prob'* [*locale-witness*]: *monad-prob return (bind :: ('a, 'a prob) bind) (sample :: ('p, 'a prob) sample)*
 ⟨*proof*⟩

6.3 Nondeterminism monad transformer

As the collection type is not determined from the type of the return operation, we can only provide definitions for one collection type implementation. We choose multisets. Accordingly, *altc* is not available.

consts

munionMT :: 'a itself ⇒ 'm ⇒ 'm ⇒ 'm
mUnionMT :: 'a itself ⇒ 'm multiset ⇒ 'm

overloading

return-nondetT ≡ return :: ('a, ('a, 'm) nondetT) return (unchecked)
bind-nondetT ≡ bind :: ('a, ('a, 'm) nondetT) bind (unchecked)
fail-nondetT ≡ fail :: ('a, 'm) nondetT fail (unchecked)
ask-nondetT ≡ ask :: ('r, ('a, 'm) nondetT) ask
get-nondetT ≡ get :: ('s, ('a, 'm) nondetT) get
put-nondetT ≡ put :: ('s, ('a, 'm) nondetT) put
alt-nondetT ≡ alt :: ('a, 'm) nondetT alt (unchecked)
munionMT ≡ munionMT :: 'a itself ⇒ 'm ⇒ 'm ⇒ 'm (unchecked)
mUnionMT ≡ mUnionMT :: 'a itself ⇒ 'm multiset ⇒ 'm (unchecked)

begin

interpretation *nondetM-base return bind mmerge return bind {#} λx. {#x#}*
 (+) ⟨*proof*⟩

definition *return-nondetT :: ('a, ('a, 'm) nondetT) return*
where [*code-unfold, monad-unfold*]: *return-nondetT = return-nondet*

definition *bind-nondetT :: ('a, ('a, 'm) nondetT) bind*
where [*code-unfold, monad-unfold*]: *bind-nondetT = bind-nondet*

definition *fail-nondetT :: ('a, 'm) nondetT fail*
where [*code-unfold, monad-unfold*]: *fail-nondetT = fail-nondet*

definition *ask-nondetT :: ('r, ('a, 'm) nondetT) ask*
where [*code-unfold, monad-unfold*]: *ask-nondetT = ask-nondet ask*

definition *get-nondetT :: ('s, ('a, 'm) nondetT) get*
where [*code-unfold, monad-unfold*]: *get-nondetT = get-nondet get*

definition *put-nondetT :: ('s, ('a, 'm) nondetT) put*
where [*code-unfold, monad-unfold*]: *put-nondetT = put-nondet put*

definition *alt-nondetT* :: ('a, 'm) nondetT alt
where [*code-unfold, monad-unfold*]: *alt-nondetT* = *alt-nondet*

definition *munionMT* :: 'a itself \Rightarrow 'm \Rightarrow 'm \Rightarrow 'm
where *munionMT* - *m1 m2* = *bind m1* ($\lambda A.$ *bind m2* ($\lambda B.$ *return* (*A + B* :: 'a multiset)))

definition *mUnionMT* :: 'a itself \Rightarrow 'm multiset \Rightarrow 'm
where *mUnionMT* - = *fold-mset* (*munionMT* TYPE('a)) (*return* ({#} :: 'a multiset))

end

context begin

interpretation *nondetM-base* *return bind mmerge return bind* {#} $\lambda x.$ {#x#}
 (+) *<proof>*

lemma *run-bind-nondetT*:

fixes *f* :: 'a \Rightarrow ('a, 'm) nondetT **shows**
run-nondet (*bind m f*) = *bind* (*run-nondet m*) ($\lambda A.$ *mUnionMT* TYPE('a)
 (*image-mset* (*run-nondet* \circ *f*) *A*))
<proof>

lemma *run-return-nondetT* [*simp*]: *run-nondet* (*return x* :: ('a, 'm) nondetT) =
return {#x#} **for** *x* :: 'a
<proof>

lemma *run-fail-nondetT* [*simp*]: *run-nondet* (*fail* :: ('a, 'm) nondetT) = *return*
 ({#} :: 'a multiset)
<proof>

lemma *run-ask-nondetT* [*simp*]: *run-nondet* (*ask f*) = *ask* ($\lambda r.$ *run-nondet* (*f r*))
<proof>

lemma *run-get-nondetT* [*simp*]: *run-nondet* (*get f*) = *get* ($\lambda s.$ *run-nondet* (*f s*))
<proof>

lemma *run-put-nondetT* [*simp*]: *run-nondet* (*put s m*) = *put s* (*run-nondet m*)
<proof>

lemma *run-alt-nondetT* [*simp*]:

run-nondet (*alt m m'* :: ('a, 'm) nondetT) =
bind (*run-nondet m*) ($\lambda A :: 'a$ multiset. *bind* (*run-nondet m'*) ($\lambda B.$ *return* (*A + B*)))
<proof>

end

lemma *monad-nondetT'* [*locale-witness*]:
monad-commute return (bind :: ('a multiset, 'm) bind)
 \implies *monad return (bind :: ('a, ('a, 'm) nondetT) bind)*
<proof>

lemma *monad-fail-nondetT'* [*locale-witness*]:
monad-commute return (bind :: ('a multiset, 'm) bind)
 \implies *monad-fail return (bind :: ('a, ('a, 'm) nondetT) bind) fail*
<proof>

lemma *monad-alt-nondetT'* [*locale-witness*]:
monad-commute return (bind :: ('a multiset, 'm) bind)
 \implies *monad-alt return (bind :: ('a, ('a, 'm) nondetT) bind) alt*
<proof>

lemma *monad-fail-alt-nondetT'* [*locale-witness*]:
monad-commute return (bind :: ('a multiset, 'm) bind)
 \implies *monad-fail-alt return (bind :: ('a, ('a, 'm) nondetT) bind) fail alt*
<proof>

lemma *monad-reader-nondetT'* [*locale-witness*]:
 \llbracket *monad-commute return (bind :: ('a multiset, 'm) bind);*
monad-reader return (bind :: ('a multiset, 'm) bind) (ask :: ('r, 'm) ask) \rrbracket
 \implies *monad-reader return (bind :: ('a, ('a, 'm) nondetT) bind) (ask :: ('r, ('a, 'm)*
nondetT) ask)
<proof>

6.4 State monad transformer

overloading

get-stateT \equiv *get :: ('s, ('s, 'm) stateT) get*
put-stateT \equiv *put :: ('s, ('s, 'm) stateT) put*
bind-stateT \equiv *bind :: ('a, ('s, 'm) stateT) bind (unchecked)*
return-stateT \equiv *return :: ('a, ('s, 'm) stateT) return (unchecked)*
fail-stateT \equiv *fail :: ('s, 'm) stateT fail*
ask-stateT \equiv *ask :: ('r, ('s, 'm) stateT) ask*
sample-stateT \equiv *sample :: ('p, ('s, 'm) stateT) sample*
tell-stateT \equiv *tell :: ('w, ('s, 'm) stateT) tell*
alt-stateT \equiv *alt :: ('s, 'm) stateT alt*
altc-stateT \equiv *altc :: ('c, ('s, 'm) stateT) altc*
pause-stateT \equiv *pause :: ('o, 'i, ('s, 'm) stateT) pause*

begin

definition *get-stateT* :: ('s, ('s, 'm) stateT) get
where [*code-unfold, monad-unfold*]: *get-stateT* = *get-state*

definition *put-stateT* :: ('s, ('s, 'm) stateT) put
where [*code-unfold, monad-unfold*]: *put-stateT* = *put-state*

definition $bind\text{-}stateT :: ('a, ('s, 'm) stateT) bind$
where $[code\text{-}unfold, monad\text{-}unfold]: bind\text{-}stateT = bind\text{-}state bind$

definition $return\text{-}stateT :: ('a, ('s, 'm) stateT) return$
where $[code\text{-}unfold, monad\text{-}unfold]: return\text{-}stateT = return\text{-}state return$

definition $fail\text{-}stateT :: ('s, 'm) stateT fail$
where $[code\text{-}unfold, monad\text{-}unfold]: fail\text{-}stateT = fail\text{-}state fail$

definition $ask\text{-}stateT :: ('r, ('s, 'm) stateT) ask$
where $[code\text{-}unfold, monad\text{-}unfold]: ask\text{-}stateT = ask\text{-}state ask$

definition $sample\text{-}stateT :: ('p, ('s, 'm) stateT) sample$
where $[code\text{-}unfold, monad\text{-}unfold]: sample\text{-}stateT = sample\text{-}state sample$

definition $tell\text{-}stateT :: ('w, ('s, 'm) stateT) tell$
where $[code\text{-}unfold, monad\text{-}unfold]: tell\text{-}stateT = tell\text{-}state tell$

definition $alt\text{-}stateT :: ('s, 'm) stateT alt$
where $[code\text{-}unfold, monad\text{-}unfold]: alt\text{-}stateT = alt\text{-}state alt$

definition $altc\text{-}stateT :: ('c, ('s, 'm) stateT) altc$
where $[code\text{-}unfold, monad\text{-}unfold]: altc\text{-}stateT = altc\text{-}state altc$

definition $pause\text{-}stateT :: ('o, 'i, ('s, 'm) stateT) pause$
where $[code\text{-}unfold, monad\text{-}unfold]: pause\text{-}stateT = pause\text{-}state pause$

end

lemma $run\text{-}bind\text{-}stateT [simp]:$
 $run\text{-}state (bind x f) s = bind (run\text{-}state x s) (\lambda(a, s'). run\text{-}state (f a) s')$
 $\langle proof \rangle$

lemma $run\text{-}return\text{-}stateT [simp]: run\text{-}state (return x) s = return (x, s)$
 $\langle proof \rangle$

lemma $run\text{-}put\text{-}stateT [simp]: run\text{-}state (put s m) s' = run\text{-}state m s$
 $\langle proof \rangle$

lemma $run\text{-}get\text{-}state [simp]: run\text{-}state (get f) s = run\text{-}state (f s) s$
 $\langle proof \rangle$

lemma $run\text{-}fail\text{-}stateT [simp]: run\text{-}state fail s = fail$
 $\langle proof \rangle$

lemma $run\text{-}ask\text{-}stateT [simp]: run\text{-}state (ask f) s = ask (\lambda r. run\text{-}state (f r) s)$
 $\langle proof \rangle$

lemma $run\text{-}sample\text{-}stateT [simp]: run\text{-}state (sample p f) s = sample p (\lambda x. run\text{-}state$

(f x) s
<proof>

lemma *run-tell-stateT [simp]: run-state (tell w m) s = tell w (run-state m s)*
<proof>

lemma *run-alt-stateT [simp]: run-state (alt m m') s = alt (run-state m s) (run-state m' s)*
<proof>

lemma *run-altc-stateT [simp]: run-state (altc C f) s = altc C (λx. run-state (f x) s)*
<proof>

lemma *run-pause-stateT [simp]: run-state (pause out c) s = pause out (λinput. run-state (c input) s)*
<proof>

lemma *monad-stateT' [locale-witness]:*
monad return (bind :: ('a × 's, 'm) bind) ⇒ monad return (bind :: ('a, ('s, 'm) stateT) bind)
<proof>

lemma *monad-state-stateT' [locale-witness]:*
monad return (bind :: ('a × 's, 'm) bind)
⇒ monad-state return (bind :: ('a, ('s, 'm) stateT) bind) get (put :: ('s, ('s, 'm) stateT) put)
<proof>

lemma *monad-fail-stateT' [locale-witness]:*
monad-fail return (bind :: ('a × 's, 'm) bind) fail
⇒ monad-fail return (bind :: ('a, ('s, 'm) stateT) bind) fail
<proof>

lemma *monad-reader-stateT' [locale-witness]:*
monad-reader return (bind :: ('a × 's, 'm) bind) (ask :: ('r, 'm) ask)
⇒ monad-reader return (bind :: ('a, ('s, 'm) stateT) bind) (ask :: ('r, ('s, 'm) stateT) ask)
<proof>

lemma *monad-reader-state-stateT' [locale-witness]:*
monad-reader return (bind :: ('a × 's, 'm) bind) (ask :: ('r, 'm) ask)
⇒ monad-reader-state return (bind :: ('a, ('s, 'm) stateT) bind) (ask :: ('r, ('s, 'm) stateT) ask) get-state put-state
<proof>

lemma *monad-prob-stateT' [locale-witness]:*
monad-prob return (bind :: ('a × 's, 'm) bind) (sample :: ('p, 'm) sample)
⇒ monad-prob return (bind :: ('a, ('s, 'm) stateT) bind) (sample :: ('p, ('s, 'm)

stateT) *sample*)
<proof>

lemma *monad-state-prob-stateT'* [*locale-witness*]:

monad-prob return (bind :: ('a × 's, 'm) bind) (sample :: ('p, 'm) sample)
⇒ *monad-state-prob return (bind :: ('a, ('s, 'm) stateT) bind) get (put :: ('s, ('s, 'm) stateT) put) (sample :: ('p, ('s, 'm) stateT) sample)*
<proof>

lemma *monad-writer-stateT'* [*locale-witness*]:

monad-writer return (bind :: ('a × 's, 'm) bind) (tell :: ('w, 'm) tell)
⇒ *monad-writer return (bind :: ('a, ('s, 'm) stateT) bind) (tell :: ('w, ('s, 'm) stateT) tell)*
<proof>

lemma *monad-alt-stateT'* [*locale-witness*]:

monad-alt return (bind :: ('a × 's, 'm) bind) alt
⇒ *monad-alt return (bind :: ('a, ('s, 'm) stateT) bind) alt*
<proof>

lemma *monad-state-alt-stateT'* [*locale-witness*]:

monad-alt return (bind :: ('a × 's, 'm) bind) alt
⇒ *monad-state-alt return (bind :: ('a, ('s, 'm) stateT) bind) (get :: ('s, ('s, 'm) stateT) get) put alt*
<proof>

lemma *monad-fail-alt-stateT'* [*locale-witness*]:

monad-fail-alt return (bind :: ('a × 's, 'm) bind) fail alt
⇒ *monad-fail-alt return (bind :: ('a, ('s, 'm) stateT) bind) fail alt*
<proof>

lemma *monad-altc-stateT'* [*locale-witness*]:

monad-altc return (bind :: ('a × 's, 'm) bind) (altc :: ('c, 'm) altc)
⇒ *monad-altc return (bind :: ('a, ('s, 'm) stateT) bind) (altc :: ('c, ('s, 'm) stateT) altc)*
<proof>

lemma *monad-state-altc-stateT'* [*locale-witness*]:

monad-altc return (bind :: ('a × 's, 'm) bind) (altc :: ('c, 'm) altc)
⇒ *monad-state-altc return (bind :: ('a, ('s, 'm) stateT) bind) (get :: ('s, ('s, 'm) stateT) get) put (altc :: ('c, ('s, 'm) stateT) altc)*
<proof>

lemma *monad-resumption-stateT'* [*locale-witness*]:

monad-resumption return (bind :: ('a × 's, 'm) bind) (pause :: ('o, 'i, 'm) pause)
⇒ *monad-resumption return (bind :: ('a, ('s, 'm) stateT) bind) (pause :: ('o, 'i, ('s, 'm) stateT) pause)*
<proof>

6.5 Failure and Exception monad transformer

overloading

return-optionT \equiv *return* :: ('a, ('a, 'm) optionT) return (**unchecked**)

bind-optionT \equiv *bind* :: ('a, ('a, 'm) optionT) bind (**unchecked**)

fail-optionT \equiv *fail* :: ('a, 'm) optionT fail (**unchecked**)

catch-optionT \equiv *catch* :: ('a, 'm) optionT catch (**unchecked**)

ask-optionT \equiv *ask* :: ('r, ('a, 'm) optionT) ask

get-optionT \equiv *get* :: ('s, ('a, 'm) optionT) get

put-optionT \equiv *put* :: ('s, ('a, 'm) optionT) put

sample-optionT \equiv *sample* :: ('p, ('a, 'm) optionT) sample

tell-optionT \equiv *tell* :: ('w, ('a, 'm) optionT) tell

alt-optionT \equiv *alt* :: ('a, 'm) optionT alt

altc-optionT \equiv *altc* :: ('c, ('a, 'm) optionT) altc

pause-optionT \equiv *pause* :: ('o, 'i, ('a, 'm) optionT) pause

begin

definition *return-optionT* :: ('a, ('a, 'm) optionT) return

where [*code-unfold*, *monad-unfold*]: *return-optionT* = *return-option return*

definition *bind-optionT* :: ('a, ('a, 'm) optionT) bind

where [*code-unfold*, *monad-unfold*]: *bind-optionT* = *bind-option return bind*

definition *fail-optionT* :: ('a, 'm) optionT fail

where [*code-unfold*, *monad-unfold*]: *fail-optionT* = *fail-option return*

definition *catch-optionT* :: ('a, 'm) optionT catch

where [*code-unfold*, *monad-unfold*]: *catch-optionT* = *catch-option return bind*

definition *ask-optionT* :: ('r, ('a, 'm) optionT) ask

where [*code-unfold*, *monad-unfold*]: *ask-optionT* = *ask-option ask*

definition *get-optionT* :: ('s, ('a, 'm) optionT) get

where [*code-unfold*, *monad-unfold*]: *get-optionT* = *get-option get*

definition *put-optionT* :: ('s, ('a, 'm) optionT) put

where [*code-unfold*, *monad-unfold*]: *put-optionT* = *put-option put*

definition *sample-optionT* :: ('p, ('a, 'm) optionT) sample

where [*code-unfold*, *monad-unfold*]: *sample-optionT* = *sample-option sample*

definition *tell-optionT* :: ('w, ('a, 'm) optionT) tell

where [*code-unfold*, *monad-unfold*]: *tell-optionT* = *tell-option tell*

definition *alt-optionT* :: ('a, 'm) optionT alt

where [*code-unfold*, *monad-unfold*]: *alt-optionT* = *alt-option alt*

definition *altc-optionT* :: ('c, ('a, 'm) optionT) altc

where [*code-unfold*, *monad-unfold*]: *altc-optionT* = *altc-option altc*

definition *pause-optionT* :: ('o, 'i, ('a, 'm) optionT) pause
where [*code-unfold, monad-unfold*]: *pause-optionT* = *pause-option pause*

end

lemma *run-bind-optionT*:

fixes $f :: 'a \Rightarrow ('a, 'm) \text{ optionT}$ **shows**
 $\text{run-option (bind } x \text{ f)} = \text{bind (run-option } x) (\lambda x. \text{ case } x \text{ of None} \Rightarrow \text{return (None} \\
:: 'a \text{ option)} \mid \text{Some } y \Rightarrow \text{run-option (f } y))$
<proof>

lemma *run-return-optionT* [*simp*]: $\text{run-option (return } x :: ('a, 'm) \text{ optionT)} = \\
\text{return (Some } x)$ **for** $x :: 'a$
<proof>

lemma *run-fail-optionT* [*simp*]: $\text{run-option (fail :: ('a, 'm) \text{ optionT fail)} = \text{return} \\
(\text{None} :: 'a \text{ option})$
<proof>

lemma *run-catch-optionT* [*simp*]:

$\text{run-option (catch } m \text{ h :: ('a, 'm) \text{ optionT)} = \\
\text{bind (run-option } m) (\lambda x :: 'a \text{ option. if } x = \text{None then run-option } h \text{ else return} \\
x)$
<proof>

lemma *run-ask-optionT* [*simp*]: $\text{run-option (ask } f) = \text{ask } (\lambda r. \text{run-option (f } r))$
<proof>

lemma *run-get-optionT* [*simp*]: $\text{run-option (get } f) = \text{get } (\lambda s. \text{run-option (f } s))$
<proof>

lemma *run-put-optionT* [*simp*]: $\text{run-option (put } s \text{ m)} = \text{put } s (\text{run-option } m)$
<proof>

lemma *run-sample-optionT* [*simp*]: $\text{run-option (sample } p \text{ f)} = \text{sample } p (\lambda x. \text{run-option} \\
(\text{f } x))$
<proof>

lemma *run-tell-optionT* [*simp*]: $\text{run-option (tell } w \text{ m)} = \text{tell } w (\text{run-option } m)$
<proof>

lemma *run-alt-optionT* [*simp*]: $\text{run-option (alt } m \text{ m')} = \text{alt (run-option } m) (\text{run-option} \\
m')$
<proof>

lemma *run-altc-optionT* [*simp*]: $\text{run-option (altc } C \text{ f)} = \text{altc } C (\text{run-option } \circ \text{ f})$
<proof>

lemma *run-pause-optionT* [*simp*]: $\text{run-option (pause out } c) = \text{pause out } (\lambda \text{input.}$

run-option (c input)
<proof>

lemma *monad-optionT' [locale-witness]:*
 monad return (bind :: ('a option, 'm) bind)
 \implies *monad return (bind :: ('a, ('a, 'm) optionT) bind)*
<proof>

lemma *monad-fail-optionT' [locale-witness]:*
 monad return (bind :: ('a option, 'm) bind)
 \implies *monad-fail return (bind :: ('a, ('a, 'm) optionT) bind) fail*
<proof>

lemma *monad-catch-optionT' [locale-witness]:*
 monad return (bind :: ('a option, 'm) bind)
 \implies *monad-catch return (bind :: ('a, ('a, 'm) optionT) bind) fail catch*
<proof>

lemma *monad-reader-optionT' [locale-witness]:*
 monad-reader return (bind :: ('a option, 'm) bind) (ask :: ('r, 'm) ask)
 \implies *monad-reader return (bind :: ('a, ('a, 'm) optionT) bind) (ask :: ('r, ('a, 'm) optionT) ask)*
<proof>

lemma *monad-state-optionT' [locale-witness]:*
 monad-state return (bind :: ('a option, 'm) bind) (get :: ('s, 'm) get) put
 \implies *monad-state return (bind :: ('a, ('a, 'm) optionT) bind) (get :: ('s, ('a, 'm) optionT) get) put*
<proof>

lemma *monad-catch-state-optionT' [locale-witness]:*
 monad-state return (bind :: ('a option, 'm) bind) (get :: ('s, 'm) get) put
 \implies *monad-catch-state return (bind :: ('a, ('a, 'm) optionT) bind) fail catch (get :: ('s, ('a, 'm) optionT) get) put*
<proof>

lemma *monad-prob-optionT' [locale-witness]:*
 monad-prob return (bind :: ('a option, 'm) bind) (sample :: ('p, 'm) sample)
 \implies *monad-prob return (bind :: ('a, ('a, 'm) optionT) bind) (sample :: ('p, ('a, 'm) optionT) sample)*
<proof>

lemma *monad-state-prob-optionT' [locale-witness]:*
 monad-state-prob return (bind :: ('a option, 'm) bind) (get :: ('s, 'm) get) put (sample :: ('p, 'm) sample)
 \implies *monad-state-prob return (bind :: ('a, ('a, 'm) optionT) bind) (get :: ('s, ('a, 'm) optionT) get) put (sample :: ('p, ('a, 'm) optionT) sample)*
<proof>

lemma *monad-writer-optionT'* [locale-witness]:

monad-writer return (bind :: ('a option, 'm) bind) (tell :: ('w, 'm) tell)
 \implies *monad-writer return (bind :: ('a, ('a, 'm) optionT) bind) (tell :: ('w, ('a, 'm) optionT) tell)*
<proof>

lemma *monad-alt-optionT'* [locale-witness]:

monad-alt return (bind :: ('a option, 'm) bind) alt
 \implies *monad-alt return (bind :: ('a, ('a, 'm) optionT) bind) alt*
<proof>

lemma *monad-state-alt-optionT'* [locale-witness]:

monad-state-alt return (bind :: ('a option, 'm) bind) (get :: ('s, 'm) get) put alt
 \implies *monad-state-alt return (bind :: ('a, ('a, 'm) optionT) bind) (get :: ('s, ('a, 'm) optionT) get) put alt*
<proof>

lemma *monad-altc-optionT'* [locale-witness]:

monad-altc return (bind :: ('a option, 'm) bind) (altc :: ('c, 'm) altc)
 \implies *monad-altc return (bind :: ('a, ('a, 'm) optionT) bind) (altc :: ('c, ('a, 'm) optionT) altc)*
<proof>

lemma *monad-state-altc-optionT'* [locale-witness]:

monad-state-altc return (bind :: ('a option, 'm) bind) (get :: ('s, 'm) get) put (altc :: ('c, 'm) altc)
 \implies *monad-state-altc return (bind :: ('a, ('a, 'm) optionT) bind) (get :: ('s, ('a, 'm) optionT) get) put (altc :: ('c, ('a, 'm) optionT) altc)*
<proof>

lemma *monad-resumption-optionT'* [locale-witness]:

monad-resumption return (bind :: ('a option, 'm) bind) (pause :: ('o, 'i, 'm) pause)
 \implies *monad-resumption return (bind :: ('a, ('a, 'm) optionT) bind) (pause :: ('o, 'i, ('a, 'm) optionT) pause)*
<proof>

lemma *monad-commute-optionT'* [locale-witness]:

\llbracket *monad-commute return (bind :: ('a option, 'm) bind); monad-discard return (bind :: ('a option, 'm) bind)* \rrbracket
 \implies *monad-commute return (bind :: ('a, ('a, 'm) optionT) bind)*
<proof>

6.6 Reader monad transformer

overloading

return-envT \equiv *return :: ('a, ('r, 'm) envT) return*
bind-envT \equiv *bind :: ('a, ('r, 'm) envT) bind*
fail-envT \equiv *fail :: ('r, 'm) envT fail*
get-envT \equiv *get :: ('s, ('r, 'm) envT) get*

$put-envT \equiv put :: ('s, ('r, 'm) envT) put$
 $sample-envT \equiv sample :: ('p, ('r, 'm) envT) sample$
 $ask-envT \equiv ask :: ('r, ('r, 'm) envT) ask$
 $catch-envT \equiv catch :: ('r, 'm) envT catch$
 $alt-envT \equiv alt :: ('r, 'm) envT alt$
 $altc-envT \equiv altc :: ('c, ('r, 'm) envT) altc$
 $pause-envT \equiv pause :: ('o, 'i, ('r, 'm) envT) pause$
 $tell-envT \equiv tell :: ('w, ('r, 'm) envT) tell$

begin

definition $return-envT :: ('a, ('r, 'm) envT) return$
where $[code-unfold, monad-unfold]: return-envT = return-env return$

definition $bind-envT :: ('a, ('r, 'm) envT) bind$
where $[code-unfold, monad-unfold]: bind-envT = bind-env bind$

definition $ask-envT :: ('r, ('r, 'm) envT) ask$
where $[code-unfold, monad-unfold]: ask-envT = ask-env$

definition $fail-envT :: ('r, 'm) envT fail$
where $[code-unfold, monad-unfold]: fail-envT = fail-env fail$

definition $get-envT :: ('s, ('r, 'm) envT) get$
where $[code-unfold, monad-unfold]: get-envT = get-env get$

definition $put-envT :: ('s, ('r, 'm) envT) put$
where $[code-unfold, monad-unfold]: put-envT = put-env put$

definition $sample-envT :: ('p, ('r, 'm) envT) sample$
where $[code-unfold, monad-unfold]: sample-envT = sample-env sample$

definition $catch-envT :: ('r, 'm) envT catch$
where $[code-unfold, monad-unfold]: catch-envT = catch-env catch$

definition $alt-envT :: ('r, 'm) envT alt$
where $[code-unfold, monad-unfold]: alt-envT = alt-env alt$

definition $altc-envT :: ('c, ('r, 'm) envT) altc$
where $[code-unfold, monad-unfold]: altc-envT = altc-env altc$

definition $pause-envT :: ('o, 'i, ('r, 'm) envT) pause$
where $[code-unfold, monad-unfold]: pause-envT = pause-env pause$

definition $tell-envT :: ('w, ('r, 'm) envT) tell$
where $[code-unfold, monad-unfold]: tell-envT = tell-env tell$

end

lemma $run-bind-envT [simp]: run-env (bind x f) r = bind (run-env x r) (\lambda y.$

run-env (*f y*) *r*
⟨*proof*⟩

lemma *run-return-envT* [*simp*]: *run-env* (*return x*) *r* = *return x*
⟨*proof*⟩

lemma *run-ask-envT* [*simp*]: *run-env* (*ask f*) *r* = *run-env* (*f r*) *r*
⟨*proof*⟩

lemma *run-fail-envT* [*simp*]: *run-env fail* *r* = *fail*
⟨*proof*⟩

lemma *run-get-envT* [*simp*]: *run-env* (*get f*) *r* = *get* ($\lambda s.$ *run-env* (*f s*) *r*)
⟨*proof*⟩

lemma *run-put-envT* [*simp*]: *run-env* (*put s m*) *r* = *put s* (*run-env m r*)
⟨*proof*⟩

lemma *run-sample-envT* [*simp*]: *run-env* (*sample p f*) *r* = *sample p* ($\lambda x.$ *run-env* (*f x*) *r*)
⟨*proof*⟩

lemma *run-catch-envT* [*simp*]: *run-env* (*catch m h*) *r* = *catch* (*run-env m r*)
(*run-env h r*)
⟨*proof*⟩

lemma *run-alt-envT* [*simp*]: *run-env* (*alt m m'*) *r* = *alt* (*run-env m r*) (*run-env*
m' r)
⟨*proof*⟩

lemma *run-altc-envT* [*simp*]: *run-env* (*altc C f*) *r* = *altc C* ($\lambda x.$ *run-env* (*f x*) *r*)
⟨*proof*⟩

lemma *run-pause-envT* [*simp*]: *run-env* (*pause out c*) *r* = *pause out* ($\lambda input.$
run-env (*c input*) *r*)
⟨*proof*⟩

lemma *run-tell-envT* [*simp*]: *run-env* (*tell s m*) *r* = *tell s* (*run-env m r*)
⟨*proof*⟩

lemma *monad-envT'* [*locale-witness*]:
 monad return (*bind* :: (*'a*, *'m*) *bind*)
 \implies *monad return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*)
⟨*proof*⟩

lemma *monad-reader-envT'* [*locale-witness*]:
 monad return (*bind* :: (*'a*, *'m*) *bind*)
 \implies *monad-reader return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*) (*ask* :: (*'r*, (*'r*, *'m*)
envT) *ask*)

<proof>

lemma *monad-fail-envT'* [locale-witness]:

monad-fail return (bind :: ('a, 'm) bind) fail
 \implies *monad-fail return (bind :: ('a, ('r, 'm) envT) bind) fail*

<proof>

lemma *monad-catch-envT'* [locale-witness]:

monad-catch return (bind :: ('a, 'm) bind) fail catch
 \implies *monad-catch return (bind :: ('a, ('r, 'm) envT) bind) fail catch*

<proof>

lemma *monad-state-envT'* [locale-witness]:

monad-state return (bind :: ('a, 'm) bind) (get :: ('s, 'm) get) put
 \implies *monad-state return (bind :: ('a, ('r, 'm) envT) bind) (get :: ('s, ('r, 'm) envT) get) put*

<proof>

lemma *monad-prob-envT'* [locale-witness]:

monad-prob return (bind :: ('a, 'm) bind) (sample :: ('p, 'm) sample)
 \implies *monad-prob return (bind :: ('a, ('r, 'm) envT) bind) (sample :: ('p, ('r, 'm) envT) sample)*

<proof>

lemma *monad-state-prob-envT'* [locale-witness]:

monad-state-prob return (bind :: ('a, 'm) bind) (get :: ('s, 'm) get) put (sample :: ('p, 'm) sample)
 \implies *monad-state-prob return (bind :: ('a, ('r, 'm) envT) bind) (get :: ('s, ('r, 'm) envT) get) put (sample :: ('p, ('r, 'm) envT) sample)*

<proof>

lemma *monad-alt-envT'* [locale-witness]:

monad-alt return (bind :: ('a, 'm) bind) alt
 \implies *monad-alt return (bind :: ('a, ('r, 'm) envT) bind) alt*

<proof>

lemma *monad-fail-alt-envT'* [locale-witness]:

monad-fail-alt return (bind :: ('a, 'm) bind) fail alt
 \implies *monad-fail-alt return (bind :: ('a, ('r, 'm) envT) bind) fail alt*

<proof>

lemma *monad-state-alt-envT'* [locale-witness]:

monad-state-alt return (bind :: ('a, 'm) bind) (get :: ('s, 'm) get) put alt
 \implies *monad-state-alt return (bind :: ('a, ('r, 'm) envT) bind) (get :: ('s, ('r, 'm) envT) get) put alt*

<proof>

lemma *monad-altc-envT'* [locale-witness]:

monad-altc return (bind :: ('a, 'm) bind) (altc :: ('c, 'm) altc)

\implies *monad-altc return* (*bind* :: ('a, ('r, 'm) envT) bind) (*altc* :: ('c, ('r, 'm) envT) altc)
 <proof>

lemma *monad-state-altc-envT'* [*locale-witness*]:

monad-state-altc return (*bind* :: ('a, 'm) bind) (*get* :: ('s, 'm) get) *put* (*altc* :: ('c, 'm) altc)

\implies *monad-state-altc return* (*bind* :: ('a, ('r, 'm) envT) bind) (*get* :: ('s, ('r, 'm) envT) get) *put* (*altc* :: ('c, ('r, 'm) envT) altc)

<proof>

lemma *monad-resumption-envT'* [*locale-witness*]:

monad-resumption return (*bind* :: ('a, 'm) bind) (*pause* :: ('o, 'i, 'm) pause)

\implies *monad-resumption return* (*bind* :: ('a, ('r, 'm) envT) bind) (*pause* :: ('o, 'i, ('r, 'm) envT) pause)

<proof>

lemma *monad-writer-readerT'* [*locale-witness*]:

monad-writer return (*bind* :: ('a, 'm) bind) (*tell* :: ('w, 'm) tell)

\implies *monad-writer return* (*bind* :: ('a, ('r, 'm) envT) bind) (*tell* :: ('w, ('r, 'm) envT) tell)

<proof>

lemma *monad-commute-envT'* [*locale-witness*]:

monad-commute return (*bind* :: ('a, 'm) bind)

\implies *monad-commute return* (*bind* :: ('a, ('r, 'm) envT) bind)

<proof>

lemma *monad-discard-envT'* [*locale-witness*]:

monad-discard return (*bind* :: ('a, 'm) bind)

\implies *monad-discard return* (*bind* :: ('a, ('r, 'm) envT) bind)

<proof>

6.7 Writer monad transformer

overloading

return-writerT \equiv *return* :: ('a, ('w, 'a, 'm) writerT) return (**unchecked**)

bind-writerT \equiv *bind* :: ('a, ('w, 'a, 'm) writerT) bind (**unchecked**)

fail-writerT \equiv *fail* :: ('w, 'a, 'm) writerT fail

get-writerT \equiv *get* :: ('s, ('w, 'a, 'm) writerT) get

put-writerT \equiv *put* :: ('s, ('w, 'a, 'm) writerT) put

sample-writerT \equiv *sample* :: ('p, ('w, 'a, 'm) writerT) sample

ask-writerT \equiv *ask* :: ('r, ('w, 'a, 'm) writerT) ask

alt-writerT \equiv *alt* :: ('w, 'a, 'm) writerT alt

altc-writerT \equiv *altc* :: ('c, ('w, 'a, 'm) writerT) altc

pause-writerT \equiv *pause* :: ('o, 'i, ('w, 'a, 'm) writerT) pause

tell-writerT \equiv *tell* :: ('w, ('w, 'a, 'm) writerT) tell (**unchecked**)

begin

definition *return-writerT* :: ('a, ('w, 'a, 'm) writerT) return
where [code-unfold, monad-unfold]: *return-writerT* = *return-writer return*

definition *bind-writerT* :: ('a, ('w, 'a, 'm) writerT) bind
where [code-unfold, monad-unfold]: *bind-writerT* = *bind-writer return bind*

definition *ask-writerT* :: ('r, ('w, 'a, 'm) writerT) ask
where [code-unfold, monad-unfold]: *ask-writerT* = *ask-writer ask*

definition *fail-writerT* :: ('w, 'a, 'm) writerT fail
where [code-unfold, monad-unfold]: *fail-writerT* = *fail-writer fail*

definition *get-writerT* :: ('s, ('w, 'a, 'm) writerT) get
where [code-unfold, monad-unfold]: *get-writerT* = *get-writer get*

definition *put-writerT* :: ('s, ('w, 'a, 'm) writerT) put
where [code-unfold, monad-unfold]: *put-writerT* = *put-writer put*

definition *sample-writerT* :: ('p, ('w, 'a, 'm) writerT) sample
where [code-unfold, monad-unfold]: *sample-writerT* = *sample-writer sample*

definition *alt-writerT* :: ('w, 'a, 'm) writerT alt
where [code-unfold, monad-unfold]: *alt-writerT* = *alt-writer alt*

definition *altc-writerT* :: ('c, ('w, 'a, 'm) writerT) altc
where [code-unfold, monad-unfold]: *altc-writerT* = *altc-writer altc*

definition *pause-writerT* :: ('o, 'i, ('w, 'a, 'm) writerT) pause
where [code-unfold, monad-unfold]: *pause-writerT* = *pause-writer pause*

definition *tell-writerT* :: ('w, ('w, 'a, 'm) writerT) tell
where [code-unfold, monad-unfold]: *tell-writerT* = *tell-writer return bind*

end

lemma *run-bind-writerT* [simp]:
run-writer (bind m f :: ('w, 'a, 'm) writerT) = bind (run-writer m) (\(a :: 'a, ws :: 'w list). bind (run-writer (f a)) (\(b :: 'a, ws' :: 'w list). return (b, ws @ ws')))
 ⟨proof⟩

lemma *run-return-writerT* [simp]: *run-writer (return x :: ('w, 'a, 'm) writerT) = return (x :: 'a, [] :: 'w list)*
 ⟨proof⟩

lemma *run-ask-writerT* [simp]: *run-writer (ask f) = ask (\r. run-writer (f r))*
 ⟨proof⟩

lemma *run-fail-writerT* [simp]: *run-writer fail = fail*
 ⟨proof⟩

lemma *run-get-writerT* [simp]: *run-writer* (get f) = get (λs. *run-writer* (f s))
 ⟨proof⟩

lemma *run-put-writerT* [simp]: *run-writer* (put s m) = put s (*run-writer* m)
 ⟨proof⟩

lemma *run-sample-writerT* [simp]: *run-writer* (sample p f) = sample p (λx. *run-writer* (f x))
 ⟨proof⟩

lemma *run-alt-writerT* [simp]: *run-writer* (alt m m') = alt (*run-writer* m) (*run-writer* m')
 ⟨proof⟩

lemma *run-altc-writerT* [simp]: *run-writer* (altc C f) = altc C (*run-writer* ∘ f)
 ⟨proof⟩

lemma *run-pause-writerT* [simp]: *run-writer* (pause out c) = pause out (λinput. *run-writer* (c input))
 ⟨proof⟩

lemma *run-tell-writerT* [simp]:
run-writer (tell (w :: 'w) m :: ('w, 'a, 'm) writerT) =
 bind (*run-writer* m) (λ(a :: 'a, ws :: 'w list). return (a, w # ws))
 ⟨proof⟩

lemma *monad-writerT'* [locale-witness]:
 monad return (bind :: ('a × 'w list, 'm) bind)
 ⇒ monad return (bind :: ('a, ('w, 'a, 'm) writerT) bind)
 ⟨proof⟩

lemma *monad-writer-writerT'* [locale-witness]:
 monad return (bind :: ('a × 'w list, 'm) bind)
 ⇒ monad-writer return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (tell :: ('w, ('w, 'a, 'm) writerT) tell)
 ⟨proof⟩

lemma *monad-fail-writerT'* [locale-witness]:
 monad-fail return (bind :: ('a × 'w list, 'm) bind) fail
 ⇒ monad-fail return (bind :: ('a, ('w, 'a, 'm) writerT) bind) fail
 ⟨proof⟩

lemma *monad-state-writerT'* [locale-witness]:
 monad-state return (bind :: ('a × 'w list, 'm) bind) (get :: ('s, 'm) get) put
 ⇒ monad-state return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (get :: ('s, ('w, 'a, 'm) writerT) get) put
 ⟨proof⟩

lemma *monad-prob-writerT'* [locale-witness]:

monad-prob return (bind :: ('a × 'w list, 'm) bind) (sample :: ('p, 'm) sample)
⇒ *monad-prob return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (sample :: ('p, ('w, 'a, 'm) writerT) sample)*
{proof}

lemma *monad-state-prob-writerT'* [locale-witness]:

monad-state-prob return (bind :: ('a × 'w list, 'm) bind) (get :: ('s, 'm) get) put
(sample :: ('p, 'm) sample)
⇒ *monad-state-prob return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (get :: ('s, ('w, 'a, 'm) writerT) get) put (sample :: ('p, ('w, 'a, 'm) writerT) sample)*
{proof}

lemma *monad-reader-writerT'* [locale-witness]:

monad-reader return (bind :: ('a × 'w list, 'm) bind) (ask :: ('r, 'm) ask)
⇒ *monad-reader return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (ask :: ('r, ('w, 'a, 'm) writerT) ask)*
{proof}

lemma *monad-reader-state-writerT'* [locale-witness]:

monad-reader-state return (bind :: ('a × 'w list, 'm) bind) (ask :: ('r, 'm) ask)
(get :: ('s, 'm) get) put
⇒ *monad-reader-state return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (ask :: ('r, ('w, 'a, 'm) writerT) ask) (get :: ('s, ('w, 'a, 'm) writerT) get) put*
{proof}

lemma *monad-resumption-writerT'* [locale-witness]:

monad-resumption return (bind :: ('a × 'w list, 'm) bind) (pause :: ('o, 'i, 'm) pause)
⇒ *monad-resumption return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (pause :: ('o, 'i, ('w, 'a, 'm) writerT) pause)*
{proof}

lemma *monad-alt-writerT'* [locale-witness]:

monad-alt return (bind :: ('a × 'w list, 'm) bind) alt
⇒ *monad-alt return (bind :: ('a, ('w, 'a, 'm) writerT) bind) alt*
{proof}

lemma *monad-fail-alt-writerT'* [locale-witness]:

monad-fail-alt return (bind :: ('a × 'w list, 'm) bind) fail alt
⇒ *monad-fail-alt return (bind :: ('a, ('w, 'a, 'm) writerT) bind) fail alt*
{proof}

lemma *monad-state-alt-writerT'* [locale-witness]:

monad-state-alt return (bind :: ('a × 'w list, 'm) bind) (get :: ('s, 'm) get) put alt
⇒ *monad-state-alt return (bind :: ('a, ('w, 'a, 'm) writerT) bind) (get :: ('s, ('w, 'a, 'm) writerT) get) put alt*
{proof}

lemma *monad-altc-writerT'* [*locale-witness*]:
 $monad-altc\ return\ (bind :: ('a \times 'w\ list, 'm)\ bind)\ (altc :: ('c, 'm)\ altc)$
 $\implies monad-altc\ return\ (bind :: ('a, ('w, 'a, 'm)\ writerT)\ bind)\ (altc :: ('c, ('w,$
 $'a, 'm)\ writerT)\ altc)$
<proof>

lemma *monad-state-altc-writerT'* [*locale-witness*]:
 $monad-state-altc\ return\ (bind :: ('a \times 'w\ list, 'm)\ bind)\ (get :: ('s, 'm)\ get)\ put$
 $(altc :: ('c, 'm)\ altc)$
 $\implies monad-state-altc\ return\ (bind :: ('a, ('w, 'a, 'm)\ writerT)\ bind)\ (get :: ('s,$
 $('w, 'a, 'm)\ writerT)\ get)\ put\ (altc :: ('c, ('w, 'a, 'm)\ writerT)\ altc)$
<proof>

6.8 Continuation monad transformer

overloading

$return-contT \equiv return :: ('a, ('a, 'm)\ contT)\ return$
 $bind-contT \equiv bind :: ('a, ('a, 'm)\ contT)\ bind$
 $fail-contT \equiv fail :: ('a, 'm)\ contT\ fail$
 $get-contT \equiv get :: ('s, ('a, 'm)\ contT)\ get$
 $put-contT \equiv put :: ('s, ('a, 'm)\ contT)\ put$

begin

definition *return-contT* :: ('a, ('a, 'm) contT) return
where [*code-unfold, monad-unfold*]: *return-contT* = *return-cont*

definition *bind-contT* :: ('a, ('a, 'm) contT) bind
where [*code-unfold, monad-unfold*]: *bind-contT* = *bind-cont*

definition *fail-contT* :: ('a, 'm) contT fail
where [*code-unfold, monad-unfold*]: *fail-contT* = *fail-cont fail*

definition *get-contT* :: ('s, ('a, 'm) contT) get
where [*code-unfold, monad-unfold*]: *get-contT* = *get-cont get*

definition *put-contT* :: ('s, ('a, 'm) contT) put
where [*code-unfold, monad-unfold*]: *put-contT* = *put-cont put*

end

lemma *monad-contT'* [*locale-witness*]: *monad return (bind :: ('a, ('a, 'm) contT)*
bind)
<proof>

lemma *monad-fail-contT'* [*locale-witness*]: *monad-fail return (bind :: ('a, ('a, 'm)*
contT) bind) fail
<proof>

lemma *monad-state-contT'* [*locale-witness*]:

```

    monad-state return (bind :: ('a, 'm) bind) (get :: ('s, 'm) get) put
    ==> monad-state return (bind :: ('a, ('a, 'm) contT) bind) (get :: ('s, ('a, 'm)
contT) get) put
<proof>

```

end

7 Examples

7.1 Monadic interpreter

theory *Interpreter* **imports** *Monomorphic-Monad* **begin**

declare [[*show-variants*]]

definition *apply* :: ('a => 'b) => 'a => 'b **where** *apply* f x = f x

lemma *apply-eq-onp*: **includes** *lifting-syntax* **shows** (eq-onp P ==>> (=) ==>> (=)) *apply apply*
<proof>

7.1.1 Basic interpreter

datatype (vars: 'v) *exp* = *Var* 'v | *Const* int | *Plus* 'v *exp* 'v *exp* | *Div* 'v *exp* 'v *exp*

lemma *rel-exp-simps* [*simp*]:

```

rel-exp V (Var x) e' <-> (∃ y. e' = Var y ∧ V x y)
rel-exp V (Const n) e' <-> e' = Const n
rel-exp V (Plus e1 e2) e' <-> (∃ e1' e2'. e' = Plus e1' e2' ∧ rel-exp V e1 e1' ∧
rel-exp V e2 e2')
rel-exp V (Div e1 e2) e' <-> (∃ e1' e2'. e' = Div e1' e2' ∧ rel-exp V e1 e1' ∧
rel-exp V e2 e2')
rel-exp V e (Var y) <-> (∃ x. e = Var x ∧ V x y)
rel-exp V e (Const n) <-> e = Const n
rel-exp V e (Plus e1' e2') <-> (∃ e1 e2. e = Plus e1 e2 ∧ rel-exp V e1 e1' ∧
rel-exp V e2 e2')
rel-exp V e (Div e1' e2') <-> (∃ e1 e2. e = Div e1 e2 ∧ rel-exp V e1 e1' ∧
rel-exp V e2 e2')
<proof>

```

lemma *finite-vars* [*simp*]: *finite* (vars e)
<proof>

locale *exp-base* = *monad-fail-base* **return** *bind* *fail*
for *return* :: (int, 'm) *return*
and *bind* :: (int, 'm) *bind*
and *fail* :: 'm *fail*
begin

```

context fixes  $E :: 'v \Rightarrow 'm$  begin
primrec  $eval :: 'v \exp \Rightarrow 'm$ 
where
   $eval (Var\ x) = E\ x$ 
|  $eval (Const\ i) = return\ i$ 
|  $eval (Plus\ e1\ e2) = bind\ (eval\ e1)\ (\lambda i. bind\ (eval\ e2)\ (\lambda j. return\ (i + j)))$ 
|  $eval (Div\ e1\ e2) = bind\ (eval\ e1)\ (\lambda i. bind\ (eval\ e2)\ (\lambda j. if\ j = 0\ then\ fail\ else\ return\ (i\ div\ j)))$ 

```

end

```

context fixes  $\sigma :: 'v \Rightarrow 'w \exp$  begin
primrec  $subst :: 'v \exp \Rightarrow 'w \exp$ 
where

```

```

   $subst (Const\ n) = Const\ n$ 
|  $subst (Var\ x) = \sigma\ x$ 
|  $subst (Plus\ e1\ e2) = Plus\ (subst\ e1)\ (subst\ e2)$ 
|  $subst (Div\ e1\ e2) = Div\ (subst\ e1)\ (subst\ e2)$ 

```

end

lemma *compositional*: $eval\ E\ (subst\ \sigma\ e) = eval\ (eval\ E\ \circ\ \sigma)\ e$
<proof>

end

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

```

includes lifting-syntax shows
   $(( (= ) \implies M ) \implies ( M \implies (( = ) \implies M ) \implies M ) \implies M$ 
 $\implies ( V \implies M ) \implies rel\text{-exp}\ V \implies M)$ 
  exp-base.eval exp-base.eval
<proof>

```

declare *exp-base.eval.simps* [code]

context *exp-base* **begin**

lemma *eval-cong*:

```

  assumes  $\bigwedge x. x \in vars\ e \implies E\ x = E'\ x$ 
  shows  $eval\ E\ e = eval\ E'\ e$ 
  including lifting-syntax
<proof>

```

end

7.1.2 Memoisation

lemma *case-option-apply*: $case\ option\ none\ some\ x\ y = case\ option\ (none\ y)\ (\lambda a. some\ a\ y)\ x$

<proof>

lemma (in monad-base) *bind-if2*:

bind m (λx. if b then t x else e x) = (if b then bind m t else bind m e)

<proof>

lemma (in monad-base) *bind-case-option2*:

*bind m (λx. case-option (none x) (some x) y) = case-option (bind m none) (λa.
bind m (λx. some x a)) y*

<proof>

locale *memoization-base* = *monad-state-base* return bind get put

for *return* :: ('a, 'm) return

and *bind* :: ('a, 'm) bind

and *get* :: ('k → 'a, 'm) get

and *put* :: ('k → 'a, 'm) put

begin

definition *memo* :: ('k ⇒ 'm) ⇒ 'k ⇒ 'm

where

memo f x =

get (λtable.

case table x of Some y ⇒ return y

| None ⇒ bind (f x) (λy. update (λm. m(x ↦ y)) (return y)))

lemma *memo-cong* [*cong, fundef-cong*]: $\llbracket x = y; f y = g y \rrbracket \implies memo f x =$

memo g y

<proof>

end

declare *memoization-base.memo-def* [*code*]

locale *memoization* = *memoization-base* return bind get put + *monad-state* return

bind get put

for *return* :: ('a, 'm) return

and *bind* :: ('a, 'm) bind

and *get* :: ('k → 'a, 'm) get

and *put* :: ('k → 'a, 'm) put

begin

lemma *memo-idem*: *memo (memo f) x = memo f x*

<proof>

lemma *memo-same*:

bind (memo f x) (λa. bind (memo f x) (g a)) = bind (memo f x) (λa. g a a)

<proof>

lemma *memo-commute*:

assumes *f-bind*: $\bigwedge m x g. \text{bind } m (\lambda a. \text{bind } (f x) (g a)) = \text{bind } (f x) (\lambda b. \text{bind } m (\lambda a. g a b))$
and *f-get*: $\bigwedge x g. \text{get } (\lambda s. \text{bind } (f x) (g s)) = \text{bind } (f x) (\lambda a. \text{get } (\lambda s. g s a))$
shows $\text{bind } (\text{memo } f x) (\lambda a. \text{bind } (\text{memo } f y) (g a)) = \text{bind } (\text{memo } f y) (\lambda b. \text{bind } (\text{memo } f x) (\lambda a. g a b))$
 <proof>

end

7.1.3 Probabilistic interpreter

locale *memo-exp-base* =
exp-base return bind fail +
memoization-base return bind get put
for *return* :: $(\text{int}, 'm)$ *return*
and *bind* :: $(\text{int}, 'm)$ *bind*
and *fail* :: $'m$ *fail*
and *get* :: $('v \rightarrow \text{int}, 'm)$ *get*
and *put* :: $('v \rightarrow \text{int}, 'm)$ *put*
begin

definition *lookup* :: $'v \Rightarrow 'm$
where *lookup* $x = \text{get } (\lambda s. \text{case } s \text{ of } \text{None} \Rightarrow \text{fail} \mid \text{Some } y \Rightarrow \text{return } y)$

lemma *lookup-alt-def*: $\text{lookup } x = \text{get } (\lambda s. \text{case } \text{apply } s \text{ of } \text{None} \Rightarrow \text{fail} \mid \text{Some } y \Rightarrow \text{return } y)$
 <proof>

end

locale *prob-exp-base* =
memo-exp-base return bind fail get put +
monad-prob-base return bind sample
for *return* :: $(\text{int}, 'm)$ *return*
and *bind* :: $(\text{int}, 'm)$ *bind*
and *fail* :: $'m$ *fail*
and *get* :: $('v \rightarrow \text{int}, 'm)$ *get*
and *put* :: $('v \rightarrow \text{int}, 'm)$ *put*
and *sample* :: $(\text{int}, 'm)$ *sample*
begin

definition *sample-var* :: $('v \Rightarrow \text{int pmf}) \Rightarrow 'v \Rightarrow 'm$
where *sample-var* $X x = \text{sample } (X x) \text{return}$

definition *lazy* :: $('v \Rightarrow \text{int pmf}) \Rightarrow 'v \text{exp} \Rightarrow 'm$
where *lazy* $X \equiv \text{eval } (\text{memo } (\text{sample-var } X))$

definition *sample-vars* :: $('v \Rightarrow \text{int pmf}) \Rightarrow 'v \text{set} \Rightarrow 'm \Rightarrow 'm$
where *sample-vars* $X A m = \text{Finite-Set.fold } (\lambda x m. \text{bind } (\text{memo } (\text{sample-var } X))$

$x) (\lambda-. m)) m A$

definition *eager* :: ('v \Rightarrow int pmf) \Rightarrow 'v exp \Rightarrow 'm **where**
 eager p e = *sample-vars* p (vars e) (eval lookup e)

end

lemmas [*code*] =
 prob-exp-base.sample-var-def
 prob-exp-base.lazy-def
 prob-exp-base.eager-def

locale *prob-exp* = *prob-exp-base* return bind fail get put sample +
 memoization return bind get put +
 monad-state-prob return bind get put sample +
 monad-fail return bind fail
 for *return* :: (int, 'm) return
 and *bind* :: (int, 'm) bind
 and *fail* :: 'm fail
 and *get* :: ('v \rightarrow int, 'm) get
 and *put* :: ('v \rightarrow int, 'm) put
 and *sample* :: (int, 'm) sample
begin

lemma *comp-fun-commute-sample-var*: *comp-fun-commute* ($\lambda x m. \text{bind} (\text{memo} (\text{sample-var } X) x) (\lambda-. m)$)
 $\langle \text{proof} \rangle$

interpretation *sample-var*: *comp-fun-commute* $\lambda x m. \text{bind} (\text{memo} (\text{sample-var } X) x) (\lambda-. m)$
 rewrites $\bigwedge X m A. \text{Finite-Set.fold} (\lambda x m. \text{bind} (\text{memo} (\text{sample-var } X) x) (\lambda-. m)) m A \equiv \text{sample-vars } X A m$
 for X
 $\langle \text{proof} \rangle$

lemma *comp-fun-idem-sample-var*: *comp-fun-idem* ($\lambda x m. \text{bind} (\text{memo} (\text{sample-var } X) x) (\lambda-. m)$)
 $\langle \text{proof} \rangle$

interpretation *sample-var*: *comp-fun-idem* $\lambda x m. \text{bind} (\text{memo} (\text{sample-var } X) x) (\lambda-. m)$
 rewrites $\bigwedge X m A. \text{Finite-Set.fold} (\lambda x m. \text{bind} (\text{memo} (\text{sample-var } X) x) (\lambda-. m)) m A \equiv \text{sample-vars } X A m$
 for X
 $\langle \text{proof} \rangle$

lemma *sample-vars-empty* [*simp*]: *sample-vars* $X \{\}$ $m = m$
 $\langle \text{proof} \rangle$

lemma *sample-vars-insert*:

finite A \implies *sample-vars X (insert x A) m = bind (memo (sample-var X) x) (λ -.
sample-vars X A m)*
<proof>

lemma *sample-vars-insert2*:

finite A \implies *sample-vars X (insert x A) m = sample-vars X A (bind (memo
(sample-var X) x) (λ -. m))*
<proof>

lemma *sample-vars-union*:

\llbracket *finite A*; *finite B* $\rrbracket \implies$ *sample-vars X (A \cup B) m = sample-vars X A (sample-vars
X B m)*
<proof>

lemma *memo-lookup*:

bind (memo f x) (λ i. bind (lookup x) (g i)) = bind (memo f x) (λ i. g i i)
<proof>

lemma *lazy-eq-eager*:

assumes *put-fail*: $\bigwedge s$. *put s fail = fail*
shows *lazy X e = eager X e*
<proof>

end

interpretation *F*: *exp-base*

return-option return-id
bind-option return-id bind-id
fail-option return-id
<proof>

value [code] *F.eval* (λ x. *return-option return-id 5*) (*Plus (Var "a") (Const 7)*)

7.1.4 Moving between monad instances

global-interpretation *SFI*: *memo-exp-base*

return-state (return-option (return-id :: ((int \times ('b \rightarrow int)) option, -) return))
bind-state (bind-option return-id bind-id)
fail-state (fail-option return-id)
get-state
put-state
defines *SFI-lookup = SFI.lookup*
<proof>

interpretation *SFI*: *memoization*

return-state (return-option (return-id :: ((int \times ('b \rightarrow int)) option, -) return))
bind-state (bind-option return-id bind-id)
get-state

put-state
<proof>

global-interpretation *SFP: prob-exp*
return-state (*return-option* *return-pmf*)
bind-state (*bind-option* *return-pmf* *bind-pmf*)
fail-state (*fail-option* *return-pmf*)
get-state
put-state
sample-state (*sample-option* *bind-pmf*)
defines *SFP-lookup* = *SFP.lookup*
<proof>

interpretation *FSP: prob-exp*
return-option (*return-state* (*return-pmf* :: (*int option* × (*'b* → *int*), -) *return*))
bind-option (*return-state* *return-pmf*) (*bind-state* *bind-pmf*)
fail-option (*return-state* *return-pmf*)
get-option *get-state*
put-option *put-state*
sample-option (*sample-state* *bind-pmf*)
<proof>

locale *reader-exp-base* = *exp-base* *return* *bind* *fail* + *monad-reader-base* *return* *bind*
ask
for *return* :: (*int*, *'m*) *return*
and *bind* :: (*int*, *'m*) *bind*
and *fail* :: *'m* *fail*
and *ask* :: (*'v* → *int*, *'m*) *ask*
begin

definition *lookup* :: *'v* ⇒ *'m* **where**
lookup *x* = *ask* ($\lambda s. \text{case } s \text{ of } \text{None} \Rightarrow \text{fail} \mid \text{Some } y \Rightarrow \text{return } y$)

lemma *lookup-alt-def*:
lookup *x* = *ask* ($\lambda s. \text{apply } s \text{ of } \text{None} \Rightarrow \text{fail} \mid \text{Some } y \Rightarrow \text{return } y$)
<proof>

end

locale *exp-commute* = *exp-base* *return* *bind* *fail* + *monad-commute* *return* *bind*
for *return* :: (*int*, *'m*) *return*
and *bind* :: (*int*, *'m*) *bind*
and *fail* :: *'m* *fail*
begin

lemma *eval-reverse*:
eval *E* (*Var* *x*) = *E* *x*

```

    eval E (Const i) = return i
    eval E (Plus e1 e2) = bind (eval E e2) (λj. bind (eval E e1) (λi. return (i +
j)))
    eval E (Div e1 e2) = bind (eval E e2) (λj. bind (eval E e1) (λi. if j = 0 then
fail else return (i div j)))
⟨proof⟩

```

end

global-interpretation *RFI: reader-exp-base*

```

    return-env (return-option return-id)
    bind-env (bind-option return-id bind-id)
    fail-env (fail-option return-id)
    ask-env
    defines RFI-lookup = RFI.lookup
⟨proof⟩

```

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

lemma *cr-id-prob-eval*:

```

    notes [transfer-rule] = cr-id-prob-transfer shows
    rel-stateT (=) (rel-optionT (cr-id-prob (=)))
      (SFI.eval SFI-lookup e)
      (SFP.eval SFP-lookup e)
⟨proof⟩

```

lemma *cr-envT-stateT-lookup'*:

```

    notes [transfer-rule] = cr-envT-stateT-transfer apply-eq-onp shows
    ((=) ===> cr-envT-stateT X (rel-optionT (rel-id (rel-option (cr-prod1 X (=))))))
      RFI-lookup SFI-lookup
⟨proof⟩

```

lemma *cr-envT-stateT-eval'*:

```

    notes [transfer-rule] = cr-envT-stateT-transfer cr-envT-stateT-lookup' shows
    ((=) ===> cr-envT-stateT X (rel-optionT (rel-id (rel-option (cr-prod1 X (=))))))
      (RFI.eval RFI-lookup) (SFI.eval SFI-lookup)
⟨proof⟩

```

lemma *cr-envT-stateT-lookup* [*cr-envT-stateT-transfer*]:

```

    notes [transfer-rule] = cr-id-prob-transfer cr-envT-stateT-transfer apply-eq-onp
shows
    ((=) ===> cr-envT-stateT X (rel-optionT (cr-id-prob (rel-option (cr-prod1 X
(=))))))
      RFI-lookup SFP-lookup
⟨proof⟩

```

lemma *cr-envT-stateT-eval* [*cr-envT-stateT-transfer*]:

```

    notes [transfer-rule] = cr-id-prob-transfer cr-envT-stateT-transfer shows
    ((=) ===> cr-envT-stateT X (rel-optionT (cr-id-prob (rel-option (cr-prod1 X

```

```

(=))))))
  (RFI.eval RFI-lookup) (SFP.eval SFP-lookup)
⟨proof⟩

```

```

lemma prob-eval-lookup:
  run-state (SFP.eval SFP-lookup e) E =
  map-optionT (return-pmf ◦ map-option (λb. (b, E)) ◦ extract) (run-env (RFI.eval
RFI-lookup e) E)
⟨proof⟩

```

end

7.2 Non-deterministic interpreter

```

locale choose-base = monad-altc-base return bind altc
  for return :: (int, 'm) return
  and bind :: (int, 'm) bind
  and altc :: (int, 'm) altc
begin

```

```

definition choose-var :: ('v ⇒ int cset) ⇒ 'v ⇒ 'm where
  choose-var X x = altc (X x) return

```

end

```

declare choose-base.choose-var-def [code]

```

```

locale nondet-exp-base = choose-base return bind altc
  for return :: (int, 'm) return
  and bind :: (int, 'm) bind
  and get :: ('v → int, 'm) get
  and put :: ('v → int, 'm) put
  and altc :: (int, 'm) altc
begin

```

```

sublocale memo-exp-base return bind fail get put ⟨proof⟩

```

```

definition lazy where lazy X = eval (memo (choose-var X))

```

end

```

locale nondet-exp =
  monad-state-altc return bind get put altc +
  nondet-exp-base return bind get put altc +
  memoization return bind get put
  for return :: (int, 'm) return
  and bind :: (int, 'm) bind
  and get :: ('v → int, 'm) get
  and put :: ('v → int, 'm) put

```

```

    and altc :: (int, 'm) altc
begin

sublocale monad-fail return bind fail ⟨proof⟩

end

global-interpretation NI: cset-nondetM return-id bind-id merge-id merge-id
  defines NI-return = NI.return-nondet
    and NI-bind = NI.bind-nondet
    and NI-altc = NI.altc-nondet
  ⟨proof⟩

global-interpretation SNI: nondet-exp
  return-state NI-return
  bind-state NI-bind
  get-state
  put-state
  altc-state NI-altc
  defines SNI-lazy = SNI.lazy
  ⟨proof⟩

value run-state (SNI-lazy (λx. cinsert 0 (cinsert 1 cempty)) (Div (Const 2) (Var
(CHR "x")))) Map.empty

locale nondet-fail-exp-base = choose-base return bind altc
  for return :: (int, 'm) return
  and bind :: (int, 'm) bind
  and fail :: 'm fail
  and get :: ('v → int, 'm) get
  and put :: ('v → int, 'm) put
  and altc :: (int, 'm) altc
begin

sublocale memo-exp-base return bind fail get put ⟨proof⟩

definition lazy where lazy X = eval (memo (choose-var X))

end

locale nondet-fail-exp =
  monad-state-altc return bind get put altc +
  nondet-fail-exp-base return bind fail get put altc +
  memoization return bind get put +
  fail: monad-fail return bind fail
  for return :: (int, 'm) return
  and bind :: (int, 'm) bind
  and fail :: 'm fail
  and get :: ('v → int, 'm) get

```

```

and put :: ('v  $\rightarrow$  int, 'm) put
and altc :: (int, 'm) altc

```

global-interpretation *SFNI: nondet-fail-exp*

```

return-state (return-option NI-return)
bind-state (bind-option NI-return NI-bind)
fail-state (fail-option NI-return)
get-state
put-state
altc-state (altc-option NI-altc)
defines SFNI-lazy = SFNI.lazy
<proof>

```

```

value run-state (SFP.lazy ( $\lambda x$ . pmf-of-set {0, 1}) (Div (Const 2) (Var (CHR
"x")))) Map.empty

```

```

value run-state (SFNI-lazy ( $\lambda x$ . cinsert 0 (cinsert 1 cempty)) (Div (Const 2) (Var
(CHR "x")))) Map.empty

```

end

theory *Just-Do-It-Examples* **imports** *Monomorphic-Monad* **begin**

Examples adapted from Gibbons and Hinze (ICFP 2011)

7.3 Towers of Hanoi

```

type-synonym 'm tick = 'm  $\Rightarrow$  'm

```

locale *monad-count-base* = *monad-base* return bind

```

for return :: ('a, 'm) return
and bind :: ('a, 'm) bind
+
fixes tick :: 'm tick

```

locale *monad-count* = *monad-count-base* return bind tick + *monad* return bind

```

for return :: ('a, 'm) return
and bind :: ('a, 'm) bind
and tick :: 'm tick
+
assumes bind-tick: bind (tick m) f = tick (bind m f)

```

locale *hanoi-base* = *monad-count-base* return bind tick

```

for return :: (unit, 'm) return
and bind :: (unit, 'm) bind
and tick :: 'm tick

```

begin

primrec *hanoi* :: nat \Rightarrow 'm **where**

```

hanoi 0 = return ()

```

| *hanoi* (*Suc* *n*) = *bind* (*hanoi* *n*) (λ -. *tick* (*hanoi* *n*))

primrec *repeat* :: *nat* \Rightarrow 'm \Rightarrow 'm

where

repeat 0 *mx* = *return* ()

| *repeat* (*Suc* *n*) *mx* = *bind* *mx* (λ -. *repeat* *n* *mx*)

end

locale *hanoi* = *hanoi-base* *return* *bind* *tick* + *monad-count* *return* *bind* *tick*

for *return* :: (*unit*, 'm) *return*

and *bind* :: (*unit*, 'm) *bind*

and *tick* :: 'm *tick*

begin

lemma *repeat-1*: *repeat* 1 *mx* = *mx*

<proof>

lemma *repeat-add*: *repeat* (*n* + *m*) *mx* = *bind* (*repeat* *n* *mx*) (λ -. *repeat* *m* *mx*)

<proof>

lemma *hanoi-correct*: *hanoi* *n* = *repeat* (2^{\wedge} *n* - 1) (*tick* (*return* ()))

<proof>

end

7.4 Fast product

locale *fast-product-base* = *monad-catch-base* *return* *bind* *fail* *catch*

for *return* :: (*int*, 'm) *return*

and *bind* :: (*int*, 'm) *bind*

and *fail* :: 'm *fail*

and *catch* :: 'm *catch*

begin

primrec *work* :: *int list* \Rightarrow 'm

where

work [] = *return* 1

| *work* (*x* # *xs*) = (*if* *x* = 0 *then* *fail* *else* *bind* (*work* *xs*) (λ *i*. *return* (*x* * *i*)))

definition *fastprod* :: *int list* \Rightarrow 'm

where *fastprod* *xs* = *catch* (*work* *xs*) (*return* 0)

end

locale *fast-product* = *fast-product-base* *return* *bind* *fail* *catch* + *monad-catch* *return* *bind* *fail* *catch*

for *return* :: (*int*, 'm) *return*

and *bind* :: (*int*, 'm) *bind*

```
  and fail :: 'm fail
  and catch :: 'm catch
begin
```

```
lemma work-alt-def: work xs = (if 0 ∈ set xs then fail else return (prod-list xs))
⟨proof⟩
```

```
lemma fastprod-correct: fastprod xs = return (prod-list xs)
⟨proof⟩
```

```
end
```

```
end
```