Effect Polymorphism in Higher-Order Logic

Andreas Lochbihler

March 17, 2025

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

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*:
$[\![$ *finite A*; *finite B* $]\!] \Longrightarrow$ *Finite-Set.fold f x* $(A \cup B)$ = *Finite-Set.fold f* (*Finite-Set.fold f x A*) *B*
⟨*proof*⟩

lemma (**in** *comp-fun-idem*) *ffold-set-union*: *ffold f x* (A |∪| B) = *ffold f* (*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* ⇒ *'b itself* ⇒ *bool*
where *rel-itself TYPE*(-) *TYPE*(-)

lemma *type-parametric* [*transfer-rule*]: *rel-itself TYPE*(*'a*) *TYPE*(*'b*)
⟨*proof*⟩
lemma *plus-multiset-parametric* [*transfer-rule*]:
(*rel-mset A* ===> *rel-mset A* ===> *rel-mset A*) (+) (+)
⟨*proof*⟩

lemma *Mempty-parametric* [*transfer-rule*]: *rel-mset A* {#} {#}
⟨*proof*⟩

lemma *fold-mset-parametric*:
assumes *12*: (*A* ===> *B* ===> *B*) *f1 f2*
and *comp-fun-commute f1 comp-fun-commute f2*
shows (*B* ===> *rel-mset A* ===> *B*) (*fold-mset f1*) (*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*: $\bigwedge X\ Y\ x\ y.$ ⟦ *rel-fset* $A\ X\ Y$; $P\ X\ Y$; $A\ x\ y$; x |∉| $X \vee y$ |∉| Y ⟧ $\Longrightarrow$ P (*finsert* $x\ X$) (*finsert* $y\ Y$)
shows $P\ X\ Y$
⟨*proof*⟩

lemma *ffold-parametric*:
assumes *12*: (A ===> B ===> B) *f1 f2*
and *comp-fun-idem f1 comp-fun-idem f2*
shows (B ===> *rel-fset* A ===> B) (*ffold f1*) (*ffold f2*)
⟨*proof*⟩

end

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

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

lemma *cUNION-assoc*: *cUNION* (*cUNION* A f) g = *cUNION* A ($\lambda x.$ *cUNION* ($f\ x$) g)
⟨*proof*⟩

lemma *cUnion-cempty* [*simp*]: *cUnion cempty* = *cempty*
⟨*proof*⟩

lemma *cUNION-cempty* [*simp*]: *cUNION cempty* f = *cempty*
⟨*proof*⟩

lemma *cUnion-cinsert*: *cUnion* (*cinsert* x A) = *cUn* x (*cUnion* A)
⟨*proof*⟩

lemma *cUNION-cinsert*: *cUNION* (*cinsert* x A) f = *cUn* ($f\ x$) (*cUNION* A f)
⟨*proof*⟩

lemma *cUnion-csingle* [*simp*]: *cUnion* (*csingle* x) = x
⟨*proof*⟩

lemma *cUNION-csingle* [*simp*]: *cUNION* (*csingle* x) f = $f\ x$
⟨*proof*⟩

lemma *cUNION-csingle2* [*simp*]: *cUNION* A *csingle* = A
⟨*proof*⟩

lemma *cUNION-cUn*: *cUNION* (*cUn* A B) f = *cUn* (*cUNION* A f) (*cUNION* B f)
⟨*proof*⟩

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

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 $three_1$ :: *'a* (‹**1**›) **where** **1** = *fst threes*
definition $three_2$ :: *'a* (‹**2**›) **where** **2** = *fst* (*snd threes*)
definition $three_3$ :: *'a* (‹**3**›) **where** **3** = *snd* (*snd* (*threes*))

lemma *three-neq-aux*: **1** ≠ **2** **1** ≠ **3** **2** ≠ **3**
⟨*proof*⟩

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

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

lemma *bi-unique-rel-12-23* [*simp*, *transfer-rule*]: *bi-unique rel-12-23*
⟨*proof*⟩

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

lemma *bi-unique-rel-12-21* [*simp*, *transfer-rule*]: *bi-unique rel-12-21*
⟨*proof*⟩

end

lemma *bernoulli-pmf-0*: *bernoulli-pmf 0* = *return-pmf False*
⟨*proof*⟩

lemma *bernoulli-pmf-1*: *bernoulli-pmf 1* = *return-pmf True*
⟨*proof*⟩

lemma *bernoulli-Not*: *map-pmf Not* (*bernoulli-pmf r*) = *bernoulli-pmf* (*1* − *r*)
⟨*proof*⟩

lemma *pmf-eqI-avoid*: *p* = *q* **if** $\bigwedge i.\ i \neq x \Longrightarrow pmf\ p\ i = pmf\ q\ i$
⟨*proof*⟩

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 $(\lambda a.$ *sequence* xs $(f \circ (\#)\ a))$

definition *lift* :: $('a \Rightarrow 'a) \Rightarrow 'm \Rightarrow 'm$
where *lift* $f\ x$ = *bind* x $(\lambda x.$ *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*]:
  $((M ===> (A ===> M) ===> M) ===>$ *list-all2* $M ===> ($*list-all2* $A ===> M) ===> M)$ *monad-base.sequence monad-base.sequence*
⟨*proof*⟩

lemma *lift-parametric* [*transfer-rule*]:
  $((A ===> M) ===> (M ===> (A ===> M) ===> M) ===> (A ===> A) ===> M ===> M)$ *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 $(\lambda y.$ *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*]: *bind* (*lift f x*) *g* = *bind x* (*g* ∘ *f*)
⟨*proof*⟩

lemma *lift-bind* [*simp*]: *lift f* (*bind m g*) = *bind m* (λ*x*. *lift f* (*g x*))
⟨*proof*⟩

end

2.2 State

type-synonym (′*s*, ′*m*) *get* = (′*s* ⇒ ′*m*) ⇒ ′*m*
type-synonym (′*s*, ′*m*) *put* = ′*s* ⇒ ′*m* ⇒ ′*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* ⇒ ′*s*) ⇒ ′*m* ⇒ ′*m*
where *update f m* = *get* (λ*s*. *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*: ⋀*f*. *put s* (*get f*) = *put s* (*f s*)
  and *get-get*: ⋀*f*. *get* (λ*s*. *get* (*f s*)) = *get* (λ*s*. *f s s*)
  and *put-put*: *put s* (*put s′ m*) = *put s′ m*
  and *get-put*: *get* (λ*s*. *put s m*) = *m*
  and *get-const*: ⋀*m*. *get* (λ-. *m*) = *m*
  and *bind-get*: ⋀*f g*. *bind* (*get f*) *g* = *get* (λ*s*. *bind* (*f s*) *g*)
  and *bind-put*: ⋀*f*. *bind* (*put s m*) *f* = *put s* (*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*: $update\ (\lambda\text{-}.\ s)\ m = put\ s\ m$
⟨*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 ===> (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*: $\bigwedge f.$ *bind fail f* = *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 return 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* ($\lambda s.$ *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 $('r, 'm)$ *ask* $= ('r \Rightarrow 'm) \Rightarrow 'm$

locale *monad-reader-base* = *monad-base return bind*
  for *return* :: $('a, 'm)$ *return*
  and *bind* :: $('a, 'm)$ *bind*
  +
  fixes *ask* :: $('r, 'm)$ *ask*

locale *monad-reader* = *monad-reader-base return bind ask* + *monad return bind*
  for *return* :: $('a, 'm)$ *return*
  and *bind* :: $('a, 'm)$ *bind*
  and *ask* :: $('r, 'm)$ *ask*
  +
  assumes *ask-ask*: $\bigwedge f.\ ask\ (\lambda r.\ ask\ (f\ r)) = ask\ (\lambda r.\ f\ r\ r)$
  and *ask-const*: $ask\ (\lambda\text{-}.\ m) = m$
  and *bind-ask*: $\bigwedge f\ g.\ bind\ (ask\ f)\ g = ask\ (\lambda r.\ bind\ (f\ r)\ g)$
  and *bind-ask2*: $\bigwedge f.\ bind\ m\ (\lambda x.\ ask\ (f\ x)) = ask\ (\lambda r.\ bind\ m\ (\lambda x.\ f\ x\ r))$
begin

lemma *ask-bind*: $ask\ (\lambda r.\ bind\ (f\ r)\ (g\ r)) = bind\ (ask\ f)\ (\lambda x.\ ask\ (\lambda r.\ g\ r\ x))$
$\langle proof \rangle$

end

locale *monad-reader-state* =
  monad-reader return bind ask +
  monad-state return bind get put
  for *return* :: $('a, 'm)$ *return*
  and *bind* :: $('a, 'm)$ *bind*
  and *ask* :: $('r, 'm)$ *ask*
  and *get* :: $('s, 'm)$ *get*
  and *put* :: $('s, 'm)$ *put*
  +
  assumes *ask-get*: $\bigwedge f.\ ask\ (\lambda r.\ get\ (f\ r)) = get\ (\lambda s.\ ask\ (\lambda r.\ f\ r\ s))$
  and *put-ask*: $\bigwedge f.\ put\ s\ (ask\ f) = ask\ (\lambda r.\ put\ s\ (f\ r))$

2.6 Probability

type-synonym $('p, 'm)$ *sample* $= 'p\ pmf \Rightarrow ('p \Rightarrow 'm) \Rightarrow 'm$

locale *monad-prob-base* = *monad-base return bind*
  for *return* :: $('a, 'm)$ *return*
  and *bind* :: $('a, 'm)$ *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*: $\bigwedge p\ m.\ sample\ p\ (\lambda\text{-}.\ m) = m$
and *sample-return-pmf*: $\bigwedge x\ f.\ sample\ (return\text{-}pmf\ x)\ f = f\ x$
and *sample-bind-pmf*: $\bigwedge p\ f\ g.\ sample\ (bind\text{-}pmf\ p\ f)\ g = sample\ p\ (\lambda x.\ sample\ (f\ x)\ g)$
and *sample-commute*: $\bigwedge p\ q\ f.\ sample\ p\ (\lambda x.\ sample\ q\ (f\ x)) = sample\ q\ (\lambda y.\ sample\ p\ (\lambda x.\ f\ x\ y))$
— 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*: $\bigwedge p\ f\ g.\ bind\ (sample\ p\ f)\ g = sample\ p\ (\lambda x.\ bind\ (f\ x)\ g)$
and *bind-sample2*: $\bigwedge m\ f\ p.\ bind\ m\ (\lambda y.\ sample\ p\ (f\ y)) = sample\ p\ (\lambda x.\ bind\ m\ (\lambda y.\ f\ y\ x))$
and *sample-parametric*: $\bigwedge R.\ bi\text{-}unique\ R \Longrightarrow rel\text{-}fun\ (rel\text{-}pmf\ R)\ (rel\text{-}fun\ (rel\text{-}fun\ R\ (=))\ (=))\ sample\ sample$
begin

lemma *sample-cong*: $(\bigwedge x.\ x \in set\text{-}pmf\ p \Longrightarrow f\ x = g\ x) \Longrightarrow sample\ p\ f = sample\ q\ g$ **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* :: $real \Rightarrow 'm \Rightarrow 'm \Rightarrow 'm$ **where**
$pchoose\ r\ m\ m' = sample\ (map\text{-}pmf\ (\lambda b.\ if\ b\ then\ \mathbf{1}\ else\ \mathbf{2})\ (bernoulli\text{-}pmf\ r))\ (\lambda x.\ if\ x = \mathbf{1}\ then\ m\ else\ m')$

abbreviation *pchoose-syntax* :: $'m \Rightarrow real \Rightarrow 'm \Rightarrow 'm$ (‹- ◁ - ▷ -› [*100*, *0*, *100*] *99*) **where**
$m \lhd r \rhd m' \equiv pchoose\ r\ m\ m'$

lemma *pchoose-0*: $m \lhd 0 \rhd m' = m'$
⟨*proof*⟩

lemma *pchoose-1*: $m \lhd 1 \rhd m' = m$

⟨*proof*⟩

lemma *pchoose-idemp*: $m \lhd r \rhd m = m$
⟨*proof*⟩

lemma *pchoose-bind1*: $bind\ (m \lhd r \rhd m')\ f = bind\ m\ f \lhd r \rhd bind\ m'\ f$
⟨*proof*⟩

lemma *pchoose-bind2*: $bind\ m\ (\lambda x.\ f\ x \lhd p \rhd g\ x) = bind\ m\ f \lhd p \rhd bind\ m\ g$
⟨*proof*⟩

lemma *pchoose-commute*: $m \lhd 1 - r \rhd m' = m' \lhd r \rhd m$
⟨*proof*⟩

lemma *pchoose-assoc*: $m \lhd p \rhd (m' \lhd q \rhd m'') = (m \lhd r \rhd m') \lhd s \rhd m''$ (**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′*: $m \lhd p \rhd (m' \lhd q \rhd m'') = (m \lhd r \rhd m') \lhd s \rhd m''$
if $p = r * s$ **and** $1 - s = (1 - p) * (1 - q)$
and $0 \leq p\ p \leq 1\ 0 \leq q\ q \leq 1\ 0 \leq r\ r \leq 1\ 0 \leq s\ s \leq 1$
⟨*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*: $sample\ p\ (\lambda x.\ get\ (f\ x)) = get\ (\lambda s.\ sample\ p\ (\lambda x.\ f\ x\ s))$
begin

lemma *sample-put*: $sample\ p\ (\lambda x.\ put\ s\ (m\ x)) = put\ s\ (sample\ p\ m)$
⟨*proof*⟩

lemma *sample-update*: $sample\ p\ (\lambda x.\ update\ f\ (m\ x)) = update\ f\ (sample\ p\ m)$
⟨*proof*⟩

end

2.7 Nondeterministic choice

2.7.1 Binary choice

type-synonym $'m$ *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* :: $'m$ *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* :: $'m$ *alt*
  + — Laws taken from Gibbons, Hinze: Just do it
  assumes *alt-assoc*: $alt\ (alt\ m1\ m2)\ m3 = alt\ m1\ (alt\ m2\ m3)$
  and *bind-alt1*: $bind\ (alt\ m\ m')\ f = alt\ (bind\ m\ f)\ (bind\ m'\ f)$

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* :: $'m$ *fail*
  and *alt* :: $'m$ *alt*
  +
  assumes *alt-fail1*: $alt\ fail\ m = m$
  and *alt-fail2*: $alt\ m\ fail = m$
begin

lemma *assert-alt*: $assert\ P\ (alt\ m\ m') = alt\ (assert\ P\ m)\ (assert\ P\ m')$
$\langle proof \rangle$

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* :: $'m$ *alt*
  +
  assumes *alt-get*: $alt\ (get\ f)\ (get\ g) = get\ (\lambda x.\ alt\ (f\ x)\ (g\ x))$
  and *alt-put*: $alt\ (put\ s\ m)\ (put\ s\ m') = put\ s\ (alt\ m\ m')$
  — 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*: $alt\ (update\ f\ m)\ (update\ f\ m') = update\ f\ (alt\ m\ m')$

$\langle proof \rangle$

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*: $\bigwedge C\ g\ f.\ bind\ (altc\ C\ g)\ f = altc\ C\ (\lambda c.\ bind\ (g\ c)\ f)$
  and *altc-single*: $\bigwedge x\ f.\ altc\ (csingle\ x)\ f = f\ x$
  and *altc-cUNION*: $\bigwedge C\ f\ g.\ altc\ (cUNION\ C\ f)\ g = altc\ C\ (\lambda x.\ altc\ (f\ x)\ g)$
  — We do not assume *altc-const* like for *sample* because the choice set might be empty
  and *altc-parametric*: $\bigwedge R.\ bi\text{-}unique\ R \Longrightarrow rel\text{-}fun\ (rel\text{-}cset\ R)\ (rel\text{-}fun\ (rel\text{-}fun\ R\ (=))\ (=))\ altc\ altc$
begin

lemma *altc-cong*: $cBall\ C\ (\lambda x.\ f\ x = g\ x) \Longrightarrow altc\ C\ f = altc\ C\ g$
  $\langle proof \rangle$

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

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**)) (λ*c*. *if c* = **1** *then m1 else m2*)

lemma *monad-alt*: *monad-alt return bind alt*
⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩ **including** *cset.lifting* ⟨*proof*⟩

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*: $\bigwedge C\, f.$ *altc C* (λ*c*. *get* (*f c*)) = *get* (λ*s*. *altc C* (λ*c*. *f c s*))
and *altc-put*: $\bigwedge C\, f.$ *altc C* (λ*c*. *put s* (*f c*)) = *put s* (*altc C f*)

2.8 Writer monad

type-synonym (*'w*, *'m*) *tell* = *'w* ⇒ *'m* ⇒ *'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*. *bind* (*tell w m*) *f* = *tell w* (*bind m f*)

2.9 Resumption monad

type-synonym (*'o*, *'i*, *'m*) *pause* = *'o* ⇒ (*'i* ⇒ *'m*) ⇒ *'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*: $'i \Rightarrow ('o, 'i, 'a)$ *resumption*)

3.3.1 Plain monad

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

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

definition *pause-resumption* :: $'o \Rightarrow ('i \Rightarrow ('o, 'i, 'a)$ *resumption*$) \Rightarrow ('o, 'i, 'a)$ *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*

$\langle proof \rangle$

lemma *map-optionT'-phantom*: *map-optionT' f = map-optionT' undefined*
$\langle proof \rangle$

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*
$\langle proof \rangle$

lemma *rel-optionT-simps* [*simp*]: *rel-optionT M (OptionT m) (OptionT m') ⟷ M m m'*
$\langle proof \rangle$

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

lemma *rel-optionT-mono* [*relator-mono*]: *rel-optionT A ≤ rel-optionT B* **if** *A ≤ B*
$\langle proof \rangle$

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

lemma *rel-optionT-Grp*: *rel-optionT (BNF-Def.Grp A f) = BNF-Def.Grp {x. set-optionT x ⊆ A} (map-optionT f)*
$\langle proof \rangle$

lemma *OptionT-parametric* [*transfer-rule*]: *(M ===> rel-optionT M) OptionT OptionT*
$\langle proof \rangle$

lemma *run-option-parametric* [*transfer-rule*]: *(rel-optionT M ===> M) run-option run-option*
$\langle proof \rangle$

lemma *case-optionT-parametric* [*transfer-rule*]:
  ((M ===> X) ===> rel-optionT M ===> X) case-optionT case-optionT
$\langle proof \rangle$

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

$\langle proof \rangle$

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*))
$\langle proof \rangle$

lemma *run-return-option* [*simp*]: *run-option* (*return-option x*) = *return* (*Some x*)
$\langle proof \rangle$

lemma *run-fail-option* [*simp*]: *run-option fail-option* = *return None*
$\langle proof \rangle$

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*)
$\langle proof \rangle$

context
  assumes *monad*: *monad return bind*
begin

interpretation *monad return bind* $\langle proof \rangle$

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* ≡ *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* ≡ *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* (λ*i*. *run-option* (*c i*)))

lemma *run-pause-option* [*simp*]: *run-option* (*pause-option out c*) = *pause out* (λ*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 return-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*]:

$(((R ===> M) ===> M) ===> (R ===> \mathit{rel\text{-}optionT}\ M) ===> \mathit{rel\text{-}optionT}\ M)$ *ask-option ask-option*
⟨*proof*⟩

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

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

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

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

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

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

end

3.5 Reader monad transformer

datatype $('r, 'm)\ \mathit{envT} = \mathit{EnvT}\ (\mathit{run\text{-}env}: 'r \Rightarrow 'm)$

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

definition $\mathit{rel\text{-}envT} :: ('r \Rightarrow 'r' \Rightarrow \mathit{bool}) \Rightarrow ('m \Rightarrow 'm' \Rightarrow \mathit{bool}) \Rightarrow ('r, 'm)\ \mathit{envT} \Rightarrow ('r', 'm')\ \mathit{envT} \Rightarrow \mathit{bool}$
where $\mathit{rel\text{-}envT}\ R\ M = \mathit{BNF\text{-}Def.vimage2p}\ \mathit{run\text{-}env}\ \mathit{run\text{-}env}\ (R ===> M)$

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

g)
⟨*proof*⟩

lemma *rel-envT-simps*: *rel-envT R M* (*EnvT f*) (*EnvT g*) $\longleftrightarrow$ (*R* ===> *M*) *f g*
⟨*proof*⟩

lemma *rel-envTE* [*cases pred*]:
assumes *rel-envT R M m m'*
obtains *f g* **where** m = *EnvT f* m' = *EnvT g* (*R* ===> *M*) *f g*
⟨*proof*⟩

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

lemma *rel-envT-mono* [*relator-mono*]: $[\![\ R \leq R'; M \leq M'\ ]\!] \Longrightarrow$ *rel-envT R' M* $\leq$ *rel-envT R M'*
⟨*proof*⟩

lemma *EnvT-parametric* [*transfer-rule*]: ((*R* ===> *M*) ===> *rel-envT R M*) *EnvT EnvT*
⟨*proof*⟩

lemma *run-env-parametric* [*transfer-rule*]: (*rel-envT R M* ===> *R* ===> *M*) *run-env run-env*
⟨*proof*⟩

lemma *rec-envT-parametric* [*transfer-rule*]:
(((*R* ===> *M*) ===> *X*) ===> *rel-envT R M* ===> *X*) *rec-envT rec-envT*
⟨*proof*⟩

lemma *case-envT-parametric* [*transfer-rule*]:
(((*R* ===> *M*) ===> *X*) ===> *rel-envT R M* ===> *X*) *case-envT case-envT*
⟨*proof*⟩

end

3.5.1 Plain monad and ask

context
fixes *return* :: ('*a*, '*m*) *return*
and *bind* :: ('*a*, '*m*) *bind*
begin

definition *return-env* :: ('*a*, ('*r*, '*m*) *envT*) *return*
where *return-env x* = *EnvT* (λ-. *return x*)

primrec *bind-env* :: ('*a*, ('*r*, '*m*) *envT*) *bind*
where *bind-env* (*EnvT x*) *f* = *EnvT* (λ*r*. *bind* (*x r*) (λ*y*. *run-env* (*f y*) *r*))

definition *ask-env* :: ('*r*, ('*r*, '*m*) *envT*) *ask*
where *ask-env f* = *EnvT* (λ*r*. *run-env* (*f r*) *r*)

lemma *run-bind-env* [*simp*]: *run-env* (*bind-env x f*) *r* = *bind* (*run-env x r*) (λ*y*. *run-env* (*f y*) *r*)
⟨*proof*⟩

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

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

context
 assumes *monad*: *monad return bind*
begin

interpretation *monad return bind* :: ('*a*, '*m*) *bind* ⟨*proof*⟩

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

lemma *monad-reader-envT* [*locale-witness*]:
 monad-reader return-env bind-env ask-env
⟨*proof*⟩

end

3.5.2 Failure

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

definition *fail-env* :: ('*r*, '*m*) *envT fail*
where *fail-env* = *EnvT* (λ*r*. *fail*)

lemma *run-fail-env* [*simp*]: *run-env fail-env r* = *fail*
⟨*proof*⟩

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

context
 fixes *catch* :: '*m catch*
begin

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

lemma *run-catch-env* [*simp*]: *run-env* (*catch-env m1 m2*) *r* = *catch* (*run-env m1 r*) (*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* (λ*r*. *get* (λ*s*. *run-env* (*f s*) *r*))

definition *put-env* :: ('*s*, ('*r*, '*m*) *envT*) *put*
where *put-env s m* = *EnvT* (λ*r*. *put s* (*run-env m r*))

lemma *run-get-env* [*simp*]: *run-env* (*get-env f*) *r* = *get* (λ*s*. *run-env* (*f s*) *r*)
⟨*proof*⟩

lemma *run-put-env* [*simp*]: *run-env* (*put-env s m*) *r* = *put s* (*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* (λ*r*. *sample p* (λ*x*. *run-env* (*f x*) *r*))

lemma *run-sample-env* [*simp*]: *run-env* (*sample-env p f*) *r* = *sample p* (λ*x*. *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* (λ*r*. *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* (λ*r*. *altc C* (λ*c*. *run-env* (*f c*) *r*))

lemma *run-altc-env* [*simp*]: *run-env* (*altc-env C f*) *r* = *altc C* (λ*c*. *run-env* (*f c*) *r*)
⟨*proof*⟩

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

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

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*
⟨*proof*⟩

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* (λ*r*. *pause out* (λ*i*. *run-env* (*c i*) *r*))

lemma *run-pause-env* [*simp*]:
  run-env (*pause-env out c*) *r* = *pause out* (λ*i*. *run-env* (*c i*) *r*)
⟨*proof*⟩

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

end

3.5.8 Writer

context
  fixes *tell* :: ($'w$, $'m$) *tell*
begin

definition *tell-env* :: ($'w$, ($'r$, $'m$) *envT*) *tell*
where *tell-env w m* = *EnvT* (λr. *tell w* (*run-env m r*))

lemma *run-tell-env* [*simp*]: *run-env* (*tell-env w m*) *r* = *tell w* (*run-env m r*)
⟨*proof*⟩

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

end

3.5.9 Commutativity

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

3.5.10 Discardability

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

3.5.11 Duplicability

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

end

3.5.12 Parametricity

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

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

$((A ===> M) ===> A ===> \mathit{rel\text{-}envT}\ R\ M)$ *return-env return-env*
⟨*proof*⟩

lemma *bind-env-parametric* [*transfer-rule*]:
$((M ===> (A ===> M) ===> M) ===> \mathit{rel\text{-}envT}\ R\ M ===> (A ===> \mathit{rel\text{-}envT}\ R\ M) ===> \mathit{rel\text{-}envT}\ R\ M)$
bind-env bind-env
⟨*proof*⟩

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

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

lemma *catch-env-parametric* [*transfer-rule*]:
$((M ===> M ===> M) ===> \mathit{rel\text{-}envT}\ R\ M ===> \mathit{rel\text{-}envT}\ R\ M ===> \mathit{rel\text{-}envT}\ R\ M)$ *catch-env catch-env*
⟨*proof*⟩

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

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

lemma *sample-env-parametric* [*transfer-rule*]:
$((\mathit{rel\text{-}pmf}\ P ===> (P ===> M) ===> M) ===> \mathit{rel\text{-}pmf}\ P ===> (P ===> \mathit{rel\text{-}envT}\ R\ M) ===> \mathit{rel\text{-}envT}\ R\ M)$
sample-env sample-env
⟨*proof*⟩

lemma *alt-env-parametric* [*transfer-rule*]:
$((M ===> M ===> M) ===> \mathit{rel\text{-}envT}\ R\ M ===> \mathit{rel\text{-}envT}\ R\ M ===> \mathit{rel\text{-}envT}\ R\ M)$ *alt-env alt-env*
⟨*proof*⟩

lemma *altc-env-parametric* [*transfer-rule*]:
$((\mathit{rel\text{-}cset}\ C ===> (C ===> M) ===> M) ===> \mathit{rel\text{-}cset}\ C ===> (C ===> \mathit{rel\text{-}envT}\ R\ M) ===> \mathit{rel\text{-}envT}\ R\ M)$
altc-env altc-env
⟨*proof*⟩

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

((*Out* ===> (*In* ===> *M*) ===> *M*) ===> *Out* ===> (*In* ===> *rel-envT* *R* *M*) ===> *rel-envT* *R* *M*)
pause-env pause-env
⟨*proof*⟩

lemma *tell-env-parametric* [*transfer-rule*]:
((*W* ===> *M* ===> *M*) ===> *W* ===> *rel-envT* *R* *M* ===> *rel-envT* *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* ≡ {*x*}
abbreviation (*input*) *bind-set* :: (′*a*, ′*a set*) *bind* **where** *bind-set* ≡ λ*A f*. ⋃ (*f* ‘ *A*)
abbreviation (*input*) *fail-set* :: ′*a set fail* **where** *fail-set* ≡ {}
abbreviation (*input*) *alt-set* :: ′*a set alt* **where** *alt-set* ≡ (∪)
abbreviation (*input*) *altc-set* :: (′*c*, ′*a set*) *altc* **where** *altc-set C* ≡ λ*f*. ⋃ (*f* ‘ *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* **and** *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 = *rel-nondetT′* *top*
⟨*proof*⟩

lemma *map-nondetT′-phantom*: *map-nondetT′* f = *map-nondetT′* *undefined*
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *rel-nondetT-distr* [*relator-distr*]: *rel-nondetT* A *OO* *rel-nondetT* B = *rel-nondetT* (A *OO* B)

$\langle proof \rangle$

lemma *rel-nondetT-Grp*: *rel-nondetT* (*BNF-Def.Grp A f*) = *BNF-Def.Grp* {*x*. *set-nondetT x* ⊆ *A*} (*map-nondetT f*)
$\langle proof \rangle$

lemma *NondetT-parametric* [*transfer-rule*]: (*M* ===> *rel-nondetT M*) *NondetT* *NondetT*
$\langle proof \rangle$

lemma *run-nondet-parametric* [*transfer-rule*]: (*rel-nondetT M* ===> *M*) *run-nondet* *run-nondet*
$\langle proof \rangle$

lemma *case-nondetT-parametric* [*transfer-rule*]:
((*M* ===> *X*) ===> *rel-nondetT M* ===> *X*) *case-nondetT case-nondetT*
$\langle proof \rangle$

lemma *rec-nondetT-parametric* [*transfer-rule*]:
((*M* ===> *X*) ===> *rel-nondetT M* ===> *X*) *rec-nondetT rec-nondetT*
$\langle proof \rangle$

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** ‹∪› *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* ∘ *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) (λB. *return* ($A \cup B$))))

definition *get-nondet* :: (*'state*, *'m*) *get* ⇒ (*'state*, (*'a*, *'m*) *nondetT*) *get*
where *get-nondet get f* = *NondetT* (*get* (λs. *run-nondet* (*f s*))) **for** *get*

definition *put-nondet* :: (*'state*, *'m*) *put* ⇒ (*'state*, (*'a*, *'m*) *nondetT*) *put*
where *put-nondet put s m* = *NondetT* (*put s* (*run-nondet m*)) **for** *put*

definition *ask-nondet* :: (*'r*, *'m*) *ask* ⇒ (*'r*, (*'a*, *'m*) *nondetT*) *ask*
where *ask-nondet ask f* = *NondetT* (*ask* (λr. *run-nondet* (*f r*)))

The canonical lift of sampling into (-, -) *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 *run-return-nondet* [*simp*]: *run-nondet* (*return-nondet x*) = *return* (*single x*)
⟨*proof*⟩

lemma *run-bind-nondet* [*simp*]:
run-nondet (*bind-nondet m f*) = *bind* (*run-nondet m*) (λA. *merge A* (*run-nondet* ∘ *f*))
⟨*proof*⟩

lemma *run-fail-nondet* [*simp*]: *run-nondet fail-nondet* = *return empty*
⟨*proof*⟩

lemma *run-alt-nondet* [*simp*]:
run-nondet (*alt-nondet m1 m2*) = *bind* (*run-nondet m1*) (λA. *bind* (*run-nondet m2*) (λB. *return* ($A \cup B$)))
⟨*proof*⟩

lemma *run-get-nondet* [*simp*]: *run-nondet* (*get-nondet get f*) = *get* (λs. *run-nondet* (*f s*)) **for** *get*
⟨*proof*⟩

lemma *run-put-nondet* [*simp*]: *run-nondet* (*put-nondet put s m*) = *put s* (*run-nondet m*) **for** *put*
⟨*proof*⟩

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

end

lemma *bind-nondet-cong* [*cong*]:
nondetM-base.bind-nondet bind merge = *nondetM-base.bind-nondet bind merge*

for *bind merge* ⟨*proof*⟩

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* ⇒ '*s*
and *union* :: '*s* ⇒ '*s* ⇒ '*s* (**infixl** ‹∪› *65*)
+
assumes *bind-merge-merge*:
$\bigwedge y\, f\, g.$ *bind* (*merge y f*) ($\lambda A.$ *merge A g*) = *merge y* ($\lambda x.$ *bind* (*f x*) ($\lambda A.$ *merge A g*))
and *merge-empty*: $\bigwedge f.$ *merge empty f* = *return empty*
and *merge-single*: $\bigwedge x\, f.$ *merge* (*single x*) *f* = *f x*
and *merge-single2*: $\bigwedge A.$ *merge A* ($\lambda x.$ *return* (*single x*)) = *return A*
and *merge-union*: $\bigwedge A\ B\ f.$ *merge* ($A \cup B$) *f* = *bind* (*merge A f*) ($\lambda A'.$ *bind* (*merge B f*) ($\lambda B'.$ *return* ($A' \cup B'$)))
and *union-assoc*: $\bigwedge A\ B\ C.$ $(A \cup B) \cup C = A \cup (B \cup C)$
and *empty-union*: $\bigwedge A.$ *empty* $\cup A = A$
and *union-empty*: $\bigwedge A.$ $A \cup$ *empty* $= A$
begin

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

lemma *monad-fail-nondetT* [*locale-witness*]: *monad-fail return-nondet bind-nondet fail-nondet*
⟨*proof*⟩

lemma *monad-alt-nondetT* [*locale-witness*]: *monad-alt return-nondet bind-nondet alt-nondet*
⟨*proof*⟩

lemma *monad-fail-alt-nondetT* [*locale-witness*]:
monad-fail-alt return-nondet bind-nondet fail-nondet alt-nondet
⟨*proof*⟩

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** ‹∪› *65*)
+
assumes *monad-reader*: *monad-reader return bind ask*
assumes *merge-ask*:
$\bigwedge A$ (f :: $'a \Rightarrow 'r \Rightarrow$ ($'a$, $'m$) *nondetT*). *merge A* (λx. *ask* (λr. *run-nondet* (f x r))) =
ask (λr. *merge A* (λx. *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* ===> *M*) ===> (*A* ===> *S*) ===> *A* ===> *rel-nondetT M*)
nondetM-base.return-nondet nondetM-base.return-nondet
⟨*proof*⟩

lemma *bind-nondet-parametric* [*transfer-rule*]:
((*M* ===> (*S* ===> *M*) ===> *M*) ===> (*S* ===> (*A* ===> *M*) ===> *M*) ===>
rel-nondetT M ===> (*A* ===> *rel-nondetT M*) ===> *rel-nondetT M*)
nondetM-base.bind-nondet nondetM-base.bind-nondet
⟨*proof*⟩

lemma *fail-nondet-parametric* [*transfer-rule*]:
((*S* ===> *M*) ===> *S* ===> *rel-nondetT M*) *nondetM-base.fail-nondet nondetM-base.fail-nondet*
⟨*proof*⟩

lemma *alt-nondet-parametric* [*transfer-rule*]:
((*S* ===> *M*) ===> (*M* ===> (*S* ===> *M*) ===> *M*) ===> (*S* ===> *S* ===> *S*) ===>
rel-nondetT M ===> *rel-nondetT M* ===> *rel-nondetT M*)
nondetM-base.alt-nondet nondetM-base.alt-nondet
⟨*proof*⟩

lemma *get-nondet-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *M*) ===> (*S* ===> *rel-nondetT M*) ===> *rel-nondetT M*)
nondetM-base.get-nondet nondetM-base.get-nondet
⟨*proof*⟩

lemma *put-nondet-parametric* [*transfer-rule*]:
((*S* ===> *M* ===> *M*) ===> *S* ===> *rel-nondetT M* ===> *rel-nondetT M*)
nondetM-base.put-nondet nondetM-base.put-nondet
⟨*proof*⟩

lemma *ask-nondet-parametric* [*transfer-rule*]:
(((*R* ===> *M*) ===> *M*) ===> (*R* ===> *rel-nondetT M*) ===> *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* (λA. *bind m2* (λB. *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* ⟨*proof*⟩
interpretation *nondetM-base return bind lmerge* [] λx. [x] (@) ⟨*proof*⟩

lemma *lUnionM-empty* [*simp*]: *lUnionM* [] = *return* [] ⟨*proof*⟩
lemma *lUnionM-Cons* [*simp*]: *lUnionM* (x # M) = *lunionM* x (*lUnionM* M) **for** x M
⟨*proof*⟩
lemma *lunionM-return-empty1* [*simp*]: *lunionM* (*return* []) x = x **for** x
⟨*proof*⟩
lemma *lunionM-return-empty2* [*simp*]: *lunionM* x (*return* []) = x **for** x
⟨*proof*⟩
lemma *lunionM-return-return* [*simp*]: *lunionM* (*return* A) (*return* B) = *return* (A @ B) **for** A B
⟨*proof*⟩
lemma *lunionM-assoc*: *lunionM* (*lunionM* x y) z = *lunionM* x (*lunionM* y z) **for** x y z
⟨*proof*⟩
lemma *lunionM-lUnionM1*: *lunionM* (*lUnionM* A) x = *foldr lunionM* A x **for** A x
⟨*proof*⟩
lemma *lUnionM-append* [*simp*]: *lUnionM* (A @ B) = *lunionM* (*lUnionM* A) (*lUnionM* B) **for** A B
⟨*proof*⟩
lemma *lUnionM-return* [*simp*]: *lUnionM* (*map* (λx. *return* [x]) A) = *return* A **for** A
⟨*proof*⟩
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) (λx. *bind* (f B) (λy. *return* (x @ y))) **for** m m' f
⟨*proof*⟩

lemma *list-nondetM*: *nondetM return bind lmerge* [] (λx. [x]) (@)
⟨*proof*⟩

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* $\equiv$ *bind m1* (λA. *bind m2* (λB. *return* ($A + B$)))
    and *mUnionM* $\equiv$ *fold-mset munionM* (*return* {#})
begin

definition *mmerge* :: $'a$ *multiset* $\Rightarrow$ ($'a \Rightarrow 'm$) $\Rightarrow$ $'m$
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*

⟨*proof*⟩
lemma *munionM-return-empty1* [*simp*]: *munionM* (*return* {#}) *x* = *x* **for** *x*
⟨*proof*⟩
lemma *munionM-return-empty2* [*simp*]: *munionM x* (*return* {#}) = *x* **for** *x*
⟨*proof*⟩
lemma *munionM-return-return* [*simp*]: *munionM* (*return A*) (*return B*) = *return* (*A* + *B*) **for** *A B*
⟨*proof*⟩
lemma *munionM-assoc*: *munionM* (*munionM x y*) *z* = *munionM x* (*munionM y z*) **for** *x y z*
⟨*proof*⟩
lemma *munionM-commute*: *munionM x y* = *munionM y x* **for** *x y*
⟨*proof*⟩
lemma *munionM-mUnionM1*: *munionM* (*mUnionM A*) *x* = *fold-mset munionM x A* **for** *A x*
⟨*proof*⟩
lemma *munionM-mUnionM2*: *munionM x* (*mUnionM A*) = *fold-mset munionM x A* **for** *x A*
⟨*proof*⟩
lemma *mUnionM-add* [*simp*]: *mUnionM* (*A* + *B*) = *munionM* (*mUnionM A*) (*mUnionM B*) **for** *A B*
⟨*proof*⟩
lemma *mUnionM-return* [*simp*]: *mUnionM* (*image-mset* (*λx. return* {#*x*#}) *A*) = *return A* **for** *A*
⟨*proof*⟩
lemma *bind-munionM*: *bind* (*munionM m m′*) *f* = *munionM* (*bind m f*) (*bind m′ f*)
if ⋀*A B. f* (*A* + *B*) = *bind* (*f A*) (*λx. bind* (*f B*) (*λy. return* (*x* + *y*))) **for** *m m′ f*
⟨*proof*⟩

lemma *mset-nondetM*: *nondetM return bind mmerge* {#} (*λx.* {#*x*#}) (+)
⟨*proof*⟩

lemma *mset-nondetM-ask*:
notes *mset-nondetM*[*locale-witness*]
assumes [*locale-witness*]: *monad-reader return bind ask*
shows *nondetM-ask return bind ask mmerge* {#} (*λx.* {#*x*#}) (+)
⟨*proof*⟩

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* ===> *M*) *return1 return2*
and *bind* [*transfer-rule*]: (*M* ===> (*rel-mset A* ===> *M*) ===> *M*) *bind1 bind2*
and *comm1*: *monad-commute return1 bind1*
and *comm2*: *monad-commute return2 bind2*
shows (*rel-mset A* ===> (*A* ===> *M*) ===> *M*) (*mmerge return1 bind1*) (*mmerge return2 bind2*)
⟨*proof*⟩

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* ≡ *bind m1* (λ*A*. *bind m2* (λ*B*. *return* (*A* |∪| *B*)))
and *fUnionM* ≡ *ffold funionM* (*return* {||})
begin

definition *fmerge* :: '*a fset* ⇒ ('*a* ⇒ '*m*) ⇒ '*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* ⟨*proof*⟩
interpretation *monad-duplicate return bind* ⟨*proof*⟩
interpretation *nondetM-base return bind fmerge* {||} λ*x*. {|*x*|} (|∪|) ⟨*proof*⟩

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

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

lemma *funionM-comp-fun-idem*: *comp-fun-idem funionM*
⟨*proof*⟩

interpretation *comp-fun-idem funionM* ⟨*proof*⟩

lemma *fUnionM-empty* [*simp*]: *fUnionM* {||} = *return* {||} ⟨*proof*⟩
lemma *fUnionM-finset* [*simp*]: *fUnionM* (*finsert x M*) = *funionM x* (*fUnionM M*) **for** *x M*
⟨*proof*⟩
lemma *funionM-return-empty1* [*simp*]: *funionM* (*return* {||}) *x* = *x* **for** *x*
⟨*proof*⟩
lemma *funionM-return-empty2* [*simp*]: *funionM x* (*return* {||}) = *x* **for** *x*

⟨*proof*⟩
lemma *funionM-return-return* [*simp*]: *funionM* (*return A*) (*return B*) = *return* (*A* |∪| *B*) **for** *A B*
⟨*proof*⟩
lemma *funionM-assoc*: *funionM* (*funionM x y*) *z* = *funionM x* (*funionM y z*) **for** *x y z*
⟨*proof*⟩
lemma *funionM-commute*: *funionM x y* = *funionM y x* **for** *x y*
⟨*proof*⟩
lemma *funionM-fUnionM1*: *funionM* (*fUnionM A*) *x* = *ffold funionM x A* **for** *A x*
⟨*proof*⟩
lemma *funionM-fUnionM2*: *funionM x* (*fUnionM A*) = *ffold funionM x A* **for** *x A*
⟨*proof*⟩
lemma *fUnionM-funion* [*simp*]: *fUnionM* (*A* |∪| *B*) = *funionM* (*fUnionM A*) (*fUnionM B*) **for** *A B*
⟨*proof*⟩
lemma *fUnionM-return* [*simp*]: *fUnionM* (*fimage* (λ*x*. *return* {|*x*|}) *A*) = *return A* **for** *A*
⟨*proof*⟩
lemma *bind-funionM*: *bind* (*funionM m m'*) *f* = *funionM* (*bind m f*) (*bind m' f*)
if ⋀*A B*. *f* (*A* |∪| *B*) = *bind* (*f A*) (λ*x*. *bind* (*f B*) (λ*y*. *return* (*x* |∪| *y*))) **for** *m m' f*
⟨*proof*⟩
lemma *fUnionM-return-fempty* [*simp*]: *fUnionM* (*fimage* (λ*x*. *return* {||}) *A*) = *return* {||} **for** *A*
⟨*proof*⟩
lemma *funionM-bind*: *funionM* (*bind m f*) (*bind m g*) = *bind m* (λ*x*. *funionM* (*f x*) (*g x*)) **for** *m f g*
⟨*proof*⟩
lemma *fUnionM-funionM*:
fUnionM ((λ*y*. *funionM* (*f y*) (*g y*)) |'| *A*) = *funionM* (*fUnionM* (*f* |'| *A*)) (*fUnionM* (*g* |'| *A*)) **for** *f g A*
⟨*proof*⟩

lemma *fset-nondetM*: *nondetM return bind fmerge* {||} (λ*x*. {|*x*|}) (|∪|)
⟨*proof*⟩

lemma *fset-nondetM-ask*:
notes *fset-nondetM*[*locale-witness*]
assumes [*locale-witness*]: *monad-reader return bind ask*
shows *nondetM-ask return bind ask fmerge* {||} (λ*x*. {|*x*|}) (|∪|)
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *fmerge-bind*:
  fmerge A (λ*x*. *bind m*′ (λ*A*′. *fmerge A*′ (*f x*))) = *bind m*′ (λ*A*′. *fmerge A* (λ*x*. *fmerge A*′ (*f x*)))
  ⟨*proof*⟩

lemma *fmerge-commute*: *fmerge A* (λ*x*. *fmerge B* (*f x*)) = *fmerge B* (λ*y*. *fmerge A* (λ*x*. *f x y*))
  ⟨*proof*⟩

lemma *monad-commute-nondetT-fset* [*locale-witness*]:
  monad-commute return-nondet bind-nondet
⟨*proof*⟩

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*)
  ⟨*proof*⟩

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 *mergec* :: ($'c$, $'m$, $'c$ *cset*) *merge*
begin

definition *altc-nondet* :: ($'c$, ($'a$, $'m$) *nondetT*) *altc* **where**
altc-nondet A f = *NondetT* (*mergec A* (*run-nondet* $\circ$ *f*))

lemma *run-altc-nondet* [*simp*]: *run-nondet* (*altc-nondet A f*) = *mergec A* (*run-nondet* $\circ$ *f*)
⟨*proof*⟩

end

locale *cset-nondetM* =
cset-nondetM-base return bind merge mergec
+
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 *mergec* :: ($'c$, $'m$, $'c$ *cset*) *merge*
+
assumes *bind-merge-merge*:
$\bigwedge$*y f g*. *bind* (*merge y f*) (λ*A*. *merge A g*) = *merge y* (λ*x*. *bind* (*f x*) (λ*A*. *merge A g*))
and *merge-empty*: $\bigwedge$*f*. *merge cempty f* = *return cempty*
and *merge-single*: $\bigwedge$*x f*. *merge* (*csingle x*) *f* = *f x*
and *merge-single2*: $\bigwedge$*A*. *merge A* (λ*x*. *return* (*csingle x*)) = *return A*
and *merge-union*: $\bigwedge$*A B f*. *merge* (*cUn A B*) *f* = *bind* (*merge A f*) ($\lambda A'$. *bind* (*merge B f*) ($\lambda B'$. *return* (*cUn* A' B')))
and *bind-mergec-merge*:
$\bigwedge$*y f g*. *bind* (*mergec y f*) (λ*A*. *merge A g*) = *mergec y* (λ*x*. *bind* (*f x*) (λ*A*. *merge A g*))
and *mergec-single*: $\bigwedge$*x f*. *mergec* (*csingle x*) *f* = *f x*
and *mergec-UNION*: $\bigwedge$*C f g*. *mergec* (*cUNION C f*) *g* = *mergec C* (λ*x*. *mergec* (*f x*) *g*)
and *mergec-parametric* [*transfer-rule*]:
$\bigwedge$*R*. *bi-unique R* $\Longrightarrow$ *rel-fun* (*rel-cset R*) (*rel-fun* (*rel-fun R* (=)) (=)) *mergec mergec*
begin

interpretation *nondetM return bind merge cempty 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* ∘ *f*))

lemma *extract-merge-id* [*simp*]: *extract* (*merge-id A f*) = *cUNION A* (*extract* ∘ *f*)
  ⟨*proof*⟩

lemma *merge-id-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
  (*rel-cset A* ===> (*A* ===> *rel-id* (*rel-cset A*)) ===> *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* ⇒ (*'c*, (*'r*, *'m*) *envT*, *'c cset*) *merge* **where**
  merge-env merge A f = *EnvT* (λ*r*. *merge A* (λ*a*. *run-env* (*f a*) *r*)) **for** *merge*

lemma *run-merge-env* [*simp*]: *run-env* (*merge-env merge A f*) *r* = *merge A* (λ*a*. *run-env* (*f a*) *r*) **for** *merge*
  ⟨*proof*⟩

lemma *merge-env-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
  ((*rel-cset C* ===> (*C* ===> *M*) ===> *M*) ===> *rel-cset C* ===> (*C* ===> *rel-envT R M*) ===> *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* :: ($'a \times 's$, $'m$) *return*
  and *bind* :: ($'a \times 's$, $'m$) *bind*
begin

primrec *bind-state* :: ($'a$, ($'s$, $'m$) *stateT*) *bind*
where *bind-state* (*StateT x*) f = *StateT* ($\lambda s.$ *bind* (x s) ($\lambda(a, s')$. *run-state* (f a) s'))

definition *return-state* :: ($'a$, ($'s$, $'m$) *stateT*) *return*
where *return-state* x = *StateT* ($\lambda s.$ *return* (x, s))

definition *get-state* :: ($'s$, ($'s$, $'m$) *stateT*) *get*
where *get-state* f = *StateT* ($\lambda s.$ *run-state* (f s) s)

primrec *put-state* :: ($'s$, ($'s$, $'m$) *stateT*) *put*
where *put-state* s (*StateT* f) = *StateT* (λ-. f s)

lemma *run-put-state* [*simp*]: *run-state* (*put-state* s m) s' = *run-state* m s
⟨*proof*⟩

lemma *run-get-state* [*simp*]: *run-state* (*get-state* f) s = *run-state* (f s) s
⟨*proof*⟩

lemma *run-bind-state* [*simp*]:
  run-state (*bind-state* x f) s = *bind* (*run-state* x s) ($\lambda(a, s')$. *run-state* (f a) s')
⟨*proof*⟩

lemma *run-return-state* [*simp*]:
  run-state (*return-state* x) s = *return* (x, s)
⟨*proof*⟩

context
  assumes *monad*: *monad return bind*
begin

interpretation *monad return bind* ⟨*proof*⟩

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

lemma *monad-state-stateT* [*locale-witness*]:
  monad-state return-state bind-state get-state put-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* => ((*'a* × *'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* (λ*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* (λ*s*. *ask* (λ*r*. *run-state* (*f r*) *s*))

lemma *run-ask-state* [*simp*]:
  run-state (*ask-state f*) *s* = *ask* (*λr. run-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* :: ($'x$ ⇒ ($'b$ ⇒ $'m$) ⇒ $'m$) ⇒ $'x$ ⇒ ($'b$ ⇒ ($'s$, $'m$) *stateT*) ⇒ ($'s$, $'m$) *stateT*
where *altc-sample-state altc-sample p f* = *StateT* (*λs. altc-sample p* (*λx. run-state* (*f x*) *s*))

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

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

abbreviation *sample-state* :: ($'p$, ($'s$, $'m$) *stateT*) *sample* **where**
  sample-state ≡ *altc-sample-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*) *tell*
begin

definition *tell-state* :: ('*w*, ('*s*, '*m*) *stateT*) *tell*
where *tell-state w m* = *StateT* (λ*s*. *tell w* (*run-state m s*))

lemma *run-tell-state* [*simp*]: *run-state* (*tell-state w m*) *s* = *tell w* (*run-state m s*)
⟨*proof*⟩

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

end

3.8.6 Binary Non-determinism

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

definition *alt-state* :: ('*s*, '*m*) *stateT alt*
where *alt-state m1 m2* = *StateT* (λ*s*. *alt* (*run-state m1 s*) (*run-state m2 s*))

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

context assumes *monad-alt return bind alt* **begin**

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

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

lemma *monad-state-alt-stateT* [*locale-witness*]:
  monad-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* ≡ *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* (λ*i*. *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-prod A S* ===> *M*) ===> *A* ===> *rel-stateT S M*) *return-state return-state*
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *pause-state-parametric* [*transfer-rule*]:
$((Out ===> (In ===> M) ===> M) ===> Out ===> (In ===> rel\text{-}stateT\ S\ M) ===> rel\text{-}stateT\ S\ M)$
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* :: $('a \times 'w\ list, 'm)$ *return*
and *bind* :: $('a \times 'w\ list, 'm)$ *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) ($\lambda(a, ws)$. *bind* (*run-writer* $(f\ a)$) ($\lambda(b, ws')$. *return* $(b, ws\ @\ ws')$)))

definition *tell-writer* :: $('w, ('w, 'a, 'm)\ writerT)$ *tell*
where *tell-writer* $w\ m$ = *WriterT* (*bind* (*run-writer* m) ($\lambda(a, ws)$. *return* $(a, w\ \#\ ws)$))

lemma *run-return-writer* [*simp*]: *run-writer* (*return-writer* x) = *return* $(x, [])$
⟨*proof*⟩

lemma *run-bind-writer* [*simp*]:
run-writer (*bind-writer m f*) = *bind* (*run-writer m*) (λ(*a*, *ws*). *bind* (*run-writer* (*f a*)) (λ(*b*, *ws'*). *return* (*b*, *ws* @ *ws'*)))
⟨*proof*⟩

lemma *run-tell-writer* [*simp*]:
run-writer (*tell-writer w m*) = *bind* (*run-writer m*) (λ(*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* (*λs. 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* (*λs. run-writer* (*f s*))
⟨*proof*⟩

lemma *run-put-writer* [*simp*]: *run-writer* (*put-writer s m*) = *put s* (*run-writer m*)
⟨*proof*⟩

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

3.9.3 Probability

definition *altc-sample-writer* :: ('*x* ⇒ ('*b* ⇒ '*m*) ⇒ '*m*) ⇒ '*x* ⇒ ('*b* ⇒ ('*w*, '*a*, '*m*) *writerT*) ⇒ ('*w*, '*a*, '*m*) *writerT*
where *altc-sample-writer altc-sample p f* = *WriterT* (*altc-sample p* (*λp. run-writer* (*f p*)))

lemma *run-altc-sample-writer* [*simp*]:
run-writer (*altc-sample-writer altc-sample p f*) = *altc-sample p* (*λp. run-writer* (*f p*))
⟨*proof*⟩

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

abbreviation *sample-writer* :: ('*p*, ('*w*, '*a*, '*m*) *writerT*) *sample*
where *sample-writer* ≡ *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*
⟨*proof*⟩ **including** *lifting-syntax*
⟨*proof*⟩

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* (λ*input*. *run-writer* (*c input*)))

lemma *run-pause-writer* [*simp*]:
run-writer (*pause-writer out c*) = *pause out* (λ*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 alt*
begin

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

lemma *run-alt-writer* [*simp*]: *run-writer* (*alt-writer m m'*) = *alt* (*run-writer m*) (*run-writer m'*)
⟨*proof*⟩

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

lemma *monad-fail-alt-writerT* [*locale-witness*]:
  assumes *monad-fail-alt return bind fail alt*
  shows *monad-fail-alt return-writer bind-writer fail-writer alt-writer*
⟨*proof*⟩

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

end

3.9.7 Countable Non-determinism

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

abbreviation *altc-writer* :: (*'c*, (*'w*, *'a*, *'m*) *writerT*) *altc*
where *altc-writer* ≡ *altc-sample-writer altc*

lemma *monad-altc-writerT* [*locale-witness*]:
  assumes *monad-altc return bind altc*
  shows *monad-altc return-writer bind-writer altc-writer*
⟨*proof*⟩ **including** *lifting-syntax*
  ⟨*proof*⟩

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

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-prod A* (*list-all2 W*) ===> *M*) ===> *A* ===> *rel-writerT W A M*) *return-writer return-writer*
⟨*proof*⟩

lemma *bind-writer-parametric* [*transfer-rule*]:
((*rel-prod A* (*list-all2 W*) ===> *M*) ===> (*M* ===> (*rel-prod A* (*list-all2 W*) ===> *M*) ===> *M*)
===> *rel-writerT W A M* ===> (*A* ===> *rel-writerT W A M*) ===> *rel-writerT W A M*)
bind-writer bind-writer
⟨*proof*⟩

lemma *tell-writer-parametric* [*transfer-rule*]:
((*rel-prod A* (*list-all2 W*) ===> *M*) ===> (*M* ===> (*rel-prod A* (*list-all2 W*) ===> *M*) ===> *M*)
===> *W* ===> *rel-writerT W A M* ===> *rel-writerT W A M*)
tell-writer tell-writer
⟨*proof*⟩

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

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

lemma *get-writer-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *M*) ===> (*S* ===> *rel-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* ===> *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*) *callcc* = (('*a* ⇒ '*m*) ⇒ '*m*) ⇒ '*m*

definition *callcc-cont* :: ('*a*, ('*a*, '*m*) *contT*) *callcc*
where *callcc-cont f* = *ContT* (λ*k*. *run-cont* (*f* (λ*x*. *ContT* (λ-. *k x*))) *k*)

lemma *run-callcc-cont* [*simp*]: *run-cont* (*callcc-cont f*) *k* = *run-cont* (*f* (λ*x*. *ContT* (λ-. *k x*))) *k*
⟨*proof*⟩

3.10.2 Plain monad

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

definition *bind-cont* :: ('*a*, ('*a*, '*m*) *contT*) *bind*
where *bind-cont m f* = *ContT* (λ*k*. *run-cont m* (λ*x*. *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* (λ*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* (λ-. *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* (λ*k*. *get* (λ*s*. *run-cont* (*f s*) *k*))

definition *put-cont* :: (′*s*, (′*a*, ′*m*) *contT*) *put*
where *put-cont s m* = *ContT* (λ*k*. *put s* (*run-cont m k*))

lemma *run-get-cont* [*simp*]: *run-cont* (*get-cont f*) *k* = *get* (λ*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* ⇒ ′*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* ⇒ ′*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* ⇒ '*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* ⇒ '*m2*
+
assumes *hom-altc* [*simp*]: *h* (*altc1 C f*) = *altc2 C* (*h* ∘ *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* ⇒ '*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* ⇒ '*m2*

+
assumes *hom-pause* [*simp*]: h (*pause1 out c*) = *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 ===> *cr-id-prob A*) *return-id return-pmf*
⟨*proof*⟩

lemma *cr-id-prob-bind* [*cr-id-prob-transfer*]:
(*cr-id-prob A* ===> (A ===> *cr-id-prob B*) ===> *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* :: $'c \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \times 'c \Rightarrow bool$
where *cr-prod1* c' $A = (\lambda a\ (b,\ c).\ A\ a\ b \wedge c' = c)$

lemma *cr-prod1-simps* [*simp*]: *cr-prod1* $c'\ A\ a\ (b,\ c) \longleftrightarrow A\ a\ b \wedge c' = c$
⟨*proof*⟩

lemma *cr-prod1I*: $A\ a\ b \Longrightarrow$ *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*) ($\lambda a.\ a$) *fst*
⟨*proof*⟩

lemma *cr-prod1-case-prod-transfer* [*cr-envT-stateT-transfer*]:
((*A* ===> *eq-onp* ((=) *c*) ===> *C*) ===> *cr-prod1 c A* ===> *C*) ($\lambda f\ a.\ f\ a$ *c*) *case-prod*
⟨*proof*⟩

lemma *cr-prod1-Grp*: *cr-prod1 c* (*BNF-Def.Grp A f*) = *BNF-Def.Grp A* ($\lambda b.$ (*f b*, *c*))
⟨*proof*⟩

definition *cr-envT-stateT* :: $'s \Rightarrow ('m1 \Rightarrow 'm2 \Rightarrow bool) \Rightarrow ('s,\ 'm1)$ *envT* $\Rightarrow ('s,\ 'm2)$ *stateT* $\Rightarrow bool$
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*) $\longleftrightarrow$ *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* $\Longrightarrow$ *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* $\longleftrightarrow$ *rel-spmf A* (*run-option p*) *q*

lemma *cr-spmf-prob-optionTI*: *rel-spmf A* (*run-option p*) *q* $\Longrightarrow$ *cr-spmf-prob-optionT A p q*
⟨*proof*⟩

lemma *cr-spmf-prob-optionTD*: *cr-spmf-prob-optionT A p q* $\Longrightarrow$ *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.
$(A$ ===> *cr-spmf-prob-optionT A*) (*return-option return-pmf*) *return-spmf*
⟨*proof*⟩

lemma *cr-spmf-prob-optionT-bind-transfer*:
(*cr-spmf-prob-optionT A* ===> (*A* ===> *cr-spmf-prob-optionT A*) ===> *cr-spmf-prob-optionT A*)
(*bind-option return-pmf bind-pmf*) *bind-spmf*
⟨*proof*⟩

lemma *cr-spmf-prob-optionT-fail-transfer*:
cr-spmf-prob-optionT A (*fail-option return-pmf*) (*return-pmf None*)
⟨*proof*⟩

abbreviation (*input*) *spmf-of-prob-optionT* :: ($'a$, $'a$ *option prob*) *optionT* ⇒ $'a$ *spmf*
where *spmf-of-prob-optionT* ≡ *run-option*

abbreviation (*input*) *prob-optionT-of-spmf* :: $'a$ *spmf* ⇒ ($'a$, $'a$ *option prob*) *optionT*
where *prob-optionT-of-spmf* ≡ *OptionT*

lemma *spmf-of-prob-optionT-transfer*: (*cr-spmf-prob-optionT A* ===> *rel-spmf A*) *spmf-of-prob-optionT* ($\lambda x.\ x$)
⟨*proof*⟩

lemma *prob-optionT-of-spmf-transfer*: (*rel-spmf A* ===> *cr-spmf-prob-optionT A*) *prob-optionT-of-spmf* ($\lambda x.\ x$)
⟨*proof*⟩

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* ⟨*proof*⟩

lift-definition *cset-pmf* :: $'a$ *pmf* ⇒ $'a$ *cset* **is** *set-pmf* ⟨*proof*⟩

inductive *cr-pmf-cset* :: $'a$ *pmf* ⇒ $'a$ *cset* ⇒ *bool* **for** *p* **where**
cr-pmf-cset p (*cset-pmf p*)

lemma *cr-pmf-cset-Grp*: *cr-pmf-cset* = *BNF-Def.Grp UNIV cset-pmf*
⟨*proof*⟩

lemma *cr-pmf-cset-return-pmf* [*cr-prob-ndi-transfer*]:
((=) ===> *cr-pmf-cset*) *return-pmf csingle*

⟨*proof*⟩

inductive *cr-prob-ndi* :: ($'a \Rightarrow 'b \Rightarrow bool$) $\Rightarrow$ $'a$ *prob* $\Rightarrow$ ($'b$, $'b$ *cset id*) *nondetT* $\Rightarrow$ *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′ ≡ *bind* :: (′*a*, ′*a id*) *bind*
return-id ≡ *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*))
⟨*proof*⟩

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

lemma *monad-id′* [*locale-witness*]: *monad return* (*bind* :: (′*a*, ′*a id*) *bind*)
⟨*proof*⟩

lemma *monad-commute-id′* [*locale-witness*]: *monad-commute return* (*bind* :: (′*a*, ′*a id*) *bind*)
⟨*proof*⟩

6.2 Probability monad

overloading
return-prob ≡ *return* :: (′*a*, ′*a prob*) *return*
bind-prob ≡ *bind* :: (′*a*, ′*a prob*) *bind*
sample-prob ≡ *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*)
⟨*proof*⟩

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* ⇒ '*m* ⇒ '*m* ⇒ '*m*
where *munionMT* - *m1 m2* = *bind m1* (λ*A*. *bind m2* (λ*B*. *return* (*A* + *B* :: '*a multiset*)))

definition *mUnionMT* :: '*a itself* ⇒ '*m multiset* ⇒ '*m*
where *mUnionMT* - = *fold-mset* (*munionMT TYPE*('*a*)) (*return* ({#} :: '*a multiset*))

end

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

lemma *run-bind-nondetT*:
  fixes *f* :: '*a* ⇒ ('*a*, '*m*) *nondetT* **shows**
   run-nondet (*bind m f*) = *bind* (*run-nondet m*) (λ*A*. *mUnionMT TYPE*('*a*) (*image-mset* (*run-nondet* ∘ *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* (λ*r*. *run-nondet* (*f r*))
⟨*proof*⟩

lemma *run-get-nondetT* [*simp*]: *run-nondet* (*get f*) = *get* (λ*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*) (λ*A* :: '*a multiset*. *bind* (*run-nondet m'*) (λ*B*. *return* (*A* + *B*)))
⟨*proof*⟩

end

lemma *monad-nondetT′* [*locale-witness*]:
monad-commute return (*bind* :: (′*a multiset*, ′*m*) *bind*)
⟹ *monad return* (*bind* :: (′*a*, (′*a*, ′*m*) *nondetT*) *bind*)
⟨*proof*⟩

lemma *monad-fail-nondetT′* [*locale-witness*]:
monad-commute return (*bind* :: (′*a multiset*, ′*m*) *bind*)
⟹ *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*)
⟹ *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*)
⟹ *monad-fail-alt return* (*bind* :: (′*a*, (′*a*, ′*m*) *nondetT*) *bind*) *fail alt*
⟨*proof*⟩

lemma *monad-reader-nondetT′* [*locale-witness*]:
⟦ *monad-commute return* (*bind* :: (′*a multiset*, ′*m*) *bind*);
monad-reader return (*bind* :: (′*a multiset*, ′*m*) *bind*) (*ask* :: (′*r*, ′*m*) *ask*) ⟧
⟹ *monad-reader return* (*bind* :: (′*a*, (′*a*, ′*m*) *nondetT*) *bind*) (*ask* :: (′*r*, (′*a*, ′*m*) *nondetT*) *ask*)
⟨*proof*⟩

6.4 State monad transformer

overloading
get-stateT ≡ *get* :: (′*s*, (′*s*, ′*m*) *stateT*) *get*
put-stateT ≡ *put* :: (′*s*, (′*s*, ′*m*) *stateT*) *put*
bind-stateT ≡ *bind* :: (′*a*, (′*s*, ′*m*) *stateT*) *bind* (**unchecked**)
return-stateT ≡ *return* :: (′*a*, (′*s*, ′*m*) *stateT*) *return* (**unchecked**)
fail-stateT ≡ *fail* :: (′*s*, ′*m*) *stateT fail*
ask-stateT ≡ *ask* :: (′*r*, (′*s*, ′*m*) *stateT*) *ask*
sample-stateT ≡ *sample* :: (′*p*, (′*s*, ′*m*) *stateT*) *sample*
tell-stateT ≡ *tell* :: (′*w*, (′*s*, ′*m*) *stateT*) *tell*
alt-stateT ≡ *alt* :: (′*s*, ′*m*) *stateT alt*
altc-stateT ≡ *altc* :: (′*c*, (′*s*, ′*m*) *stateT*) *altc*
pause-stateT ≡ *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-stateT* :: ('*a*, ('*s*, '*m*) *stateT*) *bind*
where [*code-unfold*, *monad-unfold*]: *bind-stateT* = *bind-state bind*

definition *return-stateT* :: ('*a*, ('*s*, '*m*) *stateT*) *return*
where [*code-unfold*, *monad-unfold*]: *return-stateT* = *return-state return*

definition *fail-stateT* :: ('*s*, '*m*) *stateT fail*
where [*code-unfold*, *monad-unfold*]: *fail-stateT* = *fail-state fail*

definition *ask-stateT* :: ('*r*, ('*s*, '*m*) *stateT*) *ask*
where [*code-unfold*, *monad-unfold*]: *ask-stateT* = *ask-state ask*

definition *sample-stateT* :: ('*p*, ('*s*, '*m*) *stateT*) *sample*
where [*code-unfold*, *monad-unfold*]: *sample-stateT* = *sample-state sample*

definition *tell-stateT* :: ('*w*, ('*s*, '*m*) *stateT*) *tell*
where [*code-unfold*, *monad-unfold*]: *tell-stateT* = *tell-state tell*

definition *alt-stateT* :: ('*s*, '*m*) *stateT alt*
where [*code-unfold*, *monad-unfold*]: *alt-stateT* = *alt-state alt*

definition *altc-stateT* :: ('*c*, ('*s*, '*m*) *stateT*) *altc*
where [*code-unfold*, *monad-unfold*]: *altc-stateT* = *altc-state altc*

definition *pause-stateT* :: ('*o*, '*i*, ('*s*, '*m*) *stateT*) *pause*
where [*code-unfold*, *monad-unfold*]: *pause-stateT* = *pause-state pause*

end

lemma *run-bind-stateT* [*simp*]:
  run-state (*bind x f*) *s* = *bind* (*run-state x s*) (λ(*a*, *s*'). *run-state* (*f a*) *s*')
⟨*proof*⟩

lemma *run-return-stateT* [*simp*]: *run-state* (*return x*) *s* = *return* (*x*, *s*)
⟨*proof*⟩

lemma *run-put-stateT* [*simp*]: *run-state* (*put s m*) *s*' = *run-state m s*
⟨*proof*⟩

lemma *run-get-state* [*simp*]: *run-state* (*get f*) *s* = *run-state* (*f s*) *s*
⟨*proof*⟩

lemma *run-fail-stateT* [*simp*]: *run-state fail s* = *fail*
⟨*proof*⟩

lemma *run-ask-stateT* [*simp*]: *run-state* (*ask f*) *s* = *ask* (λ*r*. *run-state* (*f r*) *s*)
⟨*proof*⟩

lemma *run-sample-stateT* [*simp*]: *run-state* (*sample p f*) *s* = *sample p* (λ*x*. *run-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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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 ≡ *return* :: ('*a*, ('*a*, '*m*) *optionT*) *return* (**unchecked**)
bind-optionT ≡ *bind* :: ('*a*, ('*a*, '*m*) *optionT*) *bind* (**unchecked**)
fail-optionT ≡ *fail* :: ('*a*, '*m*) *optionT fail* (**unchecked**)
catch-optionT ≡ *catch* :: ('*a*, '*m*) *optionT catch* (**unchecked**)
ask-optionT ≡ *ask* :: ('*r*, ('*a*, '*m*) *optionT*) *ask*
get-optionT ≡ *get* :: ('*s*, ('*a*, '*m*) *optionT*) *get*
put-optionT ≡ *put* :: ('*s*, ('*a*, '*m*) *optionT*) *put*
sample-optionT ≡ *sample* :: ('*p*, ('*a*, '*m*) *optionT*) *sample*
tell-optionT ≡ *tell* :: ('*w*, ('*a*, '*m*) *optionT*) *tell*
alt-optionT ≡ *alt* :: ('*a*, '*m*) *optionT alt*
altc-optionT ≡ *altc* :: ('*c*, ('*a*, '*m*) *optionT*) *altc*
pause-optionT ≡ *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 :: $'a \Rightarrow$ ($'a$, $'m$) *optionT* **shows**
run-option (*bind x f*) = *bind* (*run-option x*) (λx. *case x of None* $\Rightarrow$ *return* (*None* :: $'a$ *option*) | *Some y* $\Rightarrow$ *run-option* (*f y*))
⟨*proof*⟩

lemma *run-return-optionT* [*simp*]: *run-option* (*return x* :: ($'a$, $'m$) *optionT*) = *return* (*Some x*) **for** x :: $'a$
⟨*proof*⟩

lemma *run-fail-optionT* [*simp*]: *run-option* (*fail* :: ($'a$, $'m$) *optionT fail*) = *return* (*None* :: $'a$ *option*)
⟨*proof*⟩

lemma *run-catch-optionT* [*simp*]:
run-option (*catch m h* :: ($'a$, $'m$) *optionT*) =
bind (*run-option m*) (λx :: $'a$ *option*. *if x* = *None then run-option h else return x*)
⟨*proof*⟩

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

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

lemma *run-put-optionT* [*simp*]: *run-option* (*put s m*) = *put s* (*run-option m*)
⟨*proof*⟩

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

lemma *run-tell-optionT* [*simp*]: *run-option* (*tell w m*) = *tell w* (*run-option m*)
⟨*proof*⟩

lemma *run-alt-optionT* [*simp*]: *run-option* (*alt m m'*) = *alt* (*run-option m*) (*run-option m'*)
⟨*proof*⟩

lemma *run-altc-optionT* [*simp*]: *run-option* (*altc C f*) = *altc C* (*run-option* $\circ$ *f*)
⟨*proof*⟩

lemma *run-pause-optionT* [*simp*]: *run-option* (*pause out c*) = *pause out* (λ*input*.

run-option (*c input*))
⟨*proof*⟩

lemma *monad-optionT'* [*locale-witness*]:
monad return (*bind* :: ('*a option*, '*m*) *bind*)
$\Longrightarrow$ *monad return* (*bind* :: ('*a*, ('*a*, '*m*) *optionT*) *bind*)
⟨*proof*⟩

lemma *monad-fail-optionT'* [*locale-witness*]:
monad return (*bind* :: ('*a option*, '*m*) *bind*)
$\Longrightarrow$ *monad-fail return* (*bind* :: ('*a*, ('*a*, '*m*) *optionT*) *bind*) *fail*
⟨*proof*⟩

lemma *monad-catch-optionT'* [*locale-witness*]:
monad return (*bind* :: ('*a option*, '*m*) *bind*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *monad-resumption return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*pause* :: (*'o*, *'i*, (*'a*, *'m*) *optionT*) *pause*)
⟨*proof*⟩

lemma *monad-commute-optionT'* [*locale-witness*]:
⟦ *monad-commute return* (*bind* :: (*'a option*, *'m*) *bind*); *monad-discard return* (*bind* :: (*'a option*, *'m*) *bind*) ⟧
$\Longrightarrow$ *monad-commute return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*)
⟨*proof*⟩

6.6 Reader monad transformer

overloading
return-envT ≡ *return* :: (*'a*, (*'r*, *'m*) *envT*) *return*
bind-envT ≡ *bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*
fail-envT ≡ *fail* :: (*'r*, *'m*) *envT fail*
get-envT ≡ *get* :: (*'s*, (*'r*, *'m*) *envT*) *get*

$put\text{-}envT \equiv put :: ('s, ('r, 'm)\ envT)\ put$
$sample\text{-}envT \equiv sample :: ('p, ('r, 'm)\ envT)\ sample$
$ask\text{-}envT \equiv ask :: ('r, ('r, 'm)\ envT)\ ask$
$catch\text{-}envT \equiv catch :: ('r, 'm)\ envT\ catch$
$alt\text{-}envT \equiv alt :: ('r, 'm)\ envT\ alt$
$altc\text{-}envT \equiv altc :: ('c, ('r, 'm)\ envT)\ altc$
$pause\text{-}envT \equiv pause :: ('o, 'i, ('r, 'm)\ envT)\ pause$
$tell\text{-}envT \equiv tell :: ('w, ('r, 'm)\ envT)\ tell$
begin

definition $return\text{-}envT :: ('a, ('r, 'm)\ envT)\ return$
where [*code-unfold*, *monad-unfold*]: $return\text{-}envT = return\text{-}env\ return$

definition $bind\text{-}envT :: ('a, ('r, 'm)\ envT)\ bind$
where [*code-unfold*, *monad-unfold*]: $bind\text{-}envT = bind\text{-}env\ bind$

definition $ask\text{-}envT :: ('r, ('r, 'm)\ envT)\ ask$
where [*code-unfold*, *monad-unfold*]: $ask\text{-}envT = ask\text{-}env$

definition $fail\text{-}envT :: ('r, 'm)\ envT\ fail$
where [*code-unfold*, *monad-unfold*]: $fail\text{-}envT = fail\text{-}env\ fail$

definition $get\text{-}envT :: ('s, ('r, 'm)\ envT)\ get$
where [*code-unfold*, *monad-unfold*]: $get\text{-}envT = get\text{-}env\ get$

definition $put\text{-}envT :: ('s, ('r, 'm)\ envT)\ put$
where [*code-unfold*, *monad-unfold*]: $put\text{-}envT = put\text{-}env\ put$

definition $sample\text{-}envT :: ('p, ('r, 'm)\ envT)\ sample$
where [*code-unfold*, *monad-unfold*]: $sample\text{-}envT = sample\text{-}env\ sample$

definition $catch\text{-}envT :: ('r, 'm)\ envT\ catch$
where [*code-unfold*, *monad-unfold*]: $catch\text{-}envT = catch\text{-}env\ catch$

definition $alt\text{-}envT :: ('r, 'm)\ envT\ alt$
where [*code-unfold*, *monad-unfold*]: $alt\text{-}envT = alt\text{-}env\ alt$

definition $altc\text{-}envT :: ('c, ('r, 'm)\ envT)\ altc$
where [*code-unfold*, *monad-unfold*]: $altc\text{-}envT = altc\text{-}env\ altc$

definition $pause\text{-}envT :: ('o, 'i, ('r, 'm)\ envT)\ pause$
where [*code-unfold*, *monad-unfold*]: $pause\text{-}envT = pause\text{-}env\ pause$

definition $tell\text{-}envT :: ('w, ('r, 'm)\ envT)\ tell$
where [*code-unfold*, *monad-unfold*]: $tell\text{-}envT = tell\text{-}env\ tell$

end

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

$run\text{-}env\ (f\ y)\ r)$
⟨*proof*⟩

lemma $run\text{-}return\text{-}envT$ [$simp$]: $run\text{-}env\ (return\ x)\ r = return\ x$
⟨*proof*⟩

lemma $run\text{-}ask\text{-}envT$ [$simp$]: $run\text{-}env\ (ask\ f)\ r = run\text{-}env\ (f\ r)\ r$
⟨*proof*⟩

lemma $run\text{-}fail\text{-}envT$ [$simp$]: $run\text{-}env\ fail\ r = fail$
⟨*proof*⟩

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

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

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

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

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

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

lemma $run\text{-}pause\text{-}envT$ [$simp$]: $run\text{-}env\ (pause\ out\ c)\ r = pause\ out\ (\lambda input.\ run\text{-}env\ (c\ input)\ r)$
⟨*proof*⟩

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

lemma $monad\text{-}envT'$ [$locale\text{-}witness$]:
$monad\ return\ (bind :: ('a,\ 'm)\ bind)$
$\Longrightarrow monad\ return\ (bind :: ('a,\ ('r,\ 'm)\ envT)\ bind)$
⟨*proof*⟩

lemma $monad\text{-}reader\text{-}envT'$ [$locale\text{-}witness$]:
$monad\ return\ (bind :: ('a,\ 'm)\ bind)$
$\Longrightarrow monad\text{-}reader\ return\ (bind :: ('a,\ ('r,\ 'm)\ envT)\ bind)\ (ask :: ('r,\ ('r,\ 'm)\ envT)\ ask)$

⟨*proof*⟩

lemma *monad-fail-envT′* [*locale-witness*]:
monad-fail return (*bind* :: (′*a*, ′*m*) *bind*) *fail*
⟹ *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*
⟹ *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*
⟹ *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*)
⟹ *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*)
⟹ *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*
⟹ *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*
⟹ *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*
⟹ *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*)

$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *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*)
$\Longrightarrow$ *monad-commute return* (*bind* :: ('*a*, ('*r*, '*m*) *envT*) *bind*)
⟨*proof*⟩

lemma *monad-discard-envT'* [*locale-witness*]:
monad-discard return (*bind* :: ('*a*, '*m*) *bind*)
$\Longrightarrow$ *monad-discard return* (*bind* :: ('*a*, ('*r*, '*m*) *envT*) *bind*)
⟨*proof*⟩

6.7 Writer monad transformer

overloading
return-writerT ≡ *return* :: ('*a*, ('*w*, '*a*, '*m*) *writerT*) *return* (**unchecked**)
bind-writerT ≡ *bind* :: ('*a*, ('*w*, '*a*, '*m*) *writerT*) *bind* (**unchecked**)
fail-writerT ≡ *fail* :: ('*w*, '*a*, '*m*) *writerT fail*
get-writerT ≡ *get* :: ('*s*, ('*w*, '*a*, '*m*) *writerT*) *get*
put-writerT ≡ *put* :: ('*s*, ('*w*, '*a*, '*m*) *writerT*) *put*
sample-writerT ≡ *sample* :: ('*p*, ('*w*, '*a*, '*m*) *writerT*) *sample*
ask-writerT ≡ *ask* :: ('*r*, ('*w*, '*a*, '*m*) *writerT*) *ask*
alt-writerT ≡ *alt* :: ('*w*, '*a*, '*m*) *writerT alt*
altc-writerT ≡ *altc* :: ('*c*, ('*w*, '*a*, '*m*) *writerT*) *altc*
pause-writerT ≡ *pause* :: ('*o*, '*i*, ('*w*, '*a*, '*m*) *writerT*) *pause*
tell-writerT ≡ *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*) (λ(*a* :: *'a*, *ws* :: *'w list*). *bind* (*run-writer* (*f a*)) (λ(*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* (λ*r*. *run-writer* (*f r*))
⟨*proof*⟩

lemma *run-fail-writerT* [*simp*]: *run-writer fail* = *fail*
⟨*proof*⟩

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

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

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

lemma *run-alt-writerT* [*simp*]: $run\text{-}writer\ (alt\ m\ m') = alt\ (run\text{-}writer\ m)\ (run\text{-}writer\ m')$
⟨*proof*⟩

lemma *run-altc-writerT* [*simp*]: $run\text{-}writer\ (altc\ C\ f) = altc\ C\ (run\text{-}writer \circ f)$
⟨*proof*⟩

lemma *run-pause-writerT* [*simp*]: $run\text{-}writer\ (pause\ out\ c) = pause\ out\ (\lambda input.\ run\text{-}writer\ (c\ input))$
⟨*proof*⟩

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

lemma *monad-writerT'* [*locale-witness*]:
$monad\ return\ (bind :: ('a \times 'w\ list, 'm)\ bind)$
$\Longrightarrow monad\ return\ (bind :: ('a, ('w, 'a, 'm)\ writerT)\ bind)$
⟨*proof*⟩

lemma *monad-writer-writerT'* [*locale-witness*]:
$monad\ return\ (bind :: ('a \times 'w\ list, 'm)\ bind)$
$\Longrightarrow monad\text{-}writer\ return\ (bind :: ('a, ('w, 'a, 'm)\ writerT)\ bind)\ (tell :: ('w, ('w, 'a, 'm)\ writerT)\ tell)$
⟨*proof*⟩

lemma *monad-fail-writerT'* [*locale-witness*]:
$monad\text{-}fail\ return\ (bind :: ('a \times 'w\ list, 'm)\ bind)\ fail$
$\Longrightarrow monad\text{-}fail\ return\ (bind :: ('a, ('w, 'a, 'm)\ writerT)\ bind)\ fail$
⟨*proof*⟩

lemma *monad-state-writerT'* [*locale-witness*]:
$monad\text{-}state\ return\ (bind :: ('a \times 'w\ list, 'm)\ bind)\ (get :: ('s, 'm)\ get)\ put$
$\Longrightarrow monad\text{-}state\ return\ (bind :: ('a, ('w, 'a, 'm)\ writerT)\ bind)\ (get :: ('s, ('w, 'a, 'm)\ writerT)\ get)\ put$
⟨*proof*⟩

lemma *monad-prob-writerT'* [*locale-witness*]:
monad-prob return (*bind* :: ($'a \times 'w$ *list*, $'m$) *bind*) (*sample* :: ($'p$, $'m$) *sample*)
$\Longrightarrow$ *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* :: ($'a \times 'w$ *list*, $'m$) *bind*) (*get* :: ($'s$, $'m$) *get*) *put* (*sample* :: ($'p$, $'m$) *sample*)
$\Longrightarrow$ *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* :: ($'a \times 'w$ *list*, $'m$) *bind*) (*ask* :: ($'r$, $'m$) *ask*)
$\Longrightarrow$ *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* :: ($'a \times 'w$ *list*, $'m$) *bind*) (*ask* :: ($'r$, $'m$) *ask*) (*get* :: ($'s$, $'m$) *get*) *put*
$\Longrightarrow$ *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* :: ($'a \times 'w$ *list*, $'m$) *bind*) (*pause* :: ($'o$, $'i$, $'m$) *pause*)
$\Longrightarrow$ *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* :: ($'a \times 'w$ *list*, $'m$) *bind*) *alt*
$\Longrightarrow$ *monad-alt return* (*bind* :: ($'a$, ($'w$, $'a$, $'m$) *writerT*) *bind*) *alt*
⟨*proof*⟩

lemma *monad-fail-alt-writerT'* [*locale-witness*]:
monad-fail-alt return (*bind* :: ($'a \times 'w$ *list*, $'m$) *bind*) *fail alt*
$\Longrightarrow$ *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* :: ($'a \times 'w$ *list*, $'m$) *bind*) (*get* :: ($'s$, $'m$) *get*) *put alt*
$\Longrightarrow$ *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* × ′*w list*, ′*m*) *bind*) (*altc* :: (′*c*, ′*m*) *altc*)
⟹ *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* × ′*w list*, ′*m*) *bind*) (*get* :: (′*s*, ′*m*) *get*) *put* (*altc* :: (′*c*, ′*m*) *altc*)
⟹ *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 ≡ *return* :: (′*a*, (′*a*, ′*m*) *contT*) *return*
bind-contT ≡ *bind* :: (′*a*, (′*a*, ′*m*) *contT*) *bind*
fail-contT ≡ *fail* :: (′*a*, ′*m*) *contT fail*
get-contT ≡ *get* :: (′*s*, (′*a*, ′*m*) *contT*) *get*
put-contT ≡ *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*
$\Longrightarrow$ *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 σ :: $'v \Rightarrow 'w\ exp$ **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*: $eval\ E\ (subst\ \sigma\ e) = eval\ (eval\ E \circ \sigma)\ e$
$\langle proof \rangle$

end

lemma *eval-parametric* [*transfer-rule*]:
  includes *lifting-syntax* **shows**
  $(((=) ===> M) ===> (M ===> ((=) ===> M) ===> M) ===> M ===> (V ===> M) ===> rel\text{-}exp\ V ===> M)$
  $exp\text{-}base.eval\ exp\text{-}base.eval$
$\langle proof \rangle$

declare *exp-base.eval.simps* [*code*]

context *exp-base* **begin**

lemma *eval-cong*:
  assumes $\bigwedge x.\ x \in vars\ e \Longrightarrow E\ x = E'\ x$
  shows $eval\ E\ e = eval\ E'\ e$
  including *lifting-syntax*
$\langle proof \rangle$

end

7.1.2 Memoisation

lemma *case-option-apply*: $case\text{-}option\ none\ some\ x\ y = case\text{-}option\ (none\ y)\ (\lambda a.\ some\ a\ y)\ x$

$\langle proof \rangle$

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*)
$\langle proof \rangle$

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*
$\langle proof \rangle$

locale *memoization-base* = *monad-state-base return bind get put*
  for *return* :: ($'a$, $'m$) *return*
  and *bind* :: ($'a$, $'m$) *bind*
  and *get* :: ($'k \rightharpoonup 'a$, $'m$) *get*
  and *put* :: ($'k \rightharpoonup 'a$, $'m$) *put*
begin

definition *memo* :: $('k \Rightarrow 'm) \Rightarrow 'k \Rightarrow 'm$
where
  memo f x =
   get (λ*table*.
   case table x of Some y $\Rightarrow$ *return y*
   | *None* $\Rightarrow$ *bind* (*f x*) (λy. *update* (λm. $m(x \mapsto y)$) (*return y*)))

lemma *memo-cong* [*cong*, *fundef-cong*]: $[\![\ x = y;\ f\ y = g\ y\ ]\!] \Longrightarrow$ *memo f x* = *memo g y*
$\langle proof \rangle$

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* :: ($'k \rightharpoonup 'a$, $'m$) *get*
  and *put* :: ($'k \rightharpoonup 'a$, $'m$) *put*
begin

lemma *memo-idem*: *memo* (*memo f*) *x* = *memo f x*
$\langle proof \rangle$

lemma *memo-same*:
  bind (*memo f x*) (λa. *bind* (*memo f x*) (*g a*)) = *bind* (*memo f x*) (λa. *g a a*)
$\langle proof \rangle$

lemma *memo-commute*:

assumes *f-bind*: $\bigwedge m\ x\ g.\ bind\ m\ (\lambda a.\ bind\ (f\ x)\ (g\ a)) = bind\ (f\ x)\ (\lambda b.\ bind\ m\ (\lambda a.\ g\ a\ b))$
and *f-get*: $\bigwedge x\ g.\ get\ (\lambda s.\ bind\ (f\ x)\ (g\ s)) = bind\ (f\ x)\ (\lambda a.\ get\ (\lambda s.\ g\ s\ a))$
shows $bind\ (memo\ f\ x)\ (\lambda a.\ bind\ (memo\ f\ y)\ (g\ a)) = bind\ (memo\ f\ y)\ (\lambda b.\ bind\ (memo\ f\ x)\ (\lambda a.\ g\ a\ b))$
⟨*proof*⟩

end

7.1.3 Probabilistic interpreter

locale *memo-exp-base* =
exp-base return bind fail +
memoization-base return bind get put
for *return* :: (*int*, *'m*) *return*
and *bind* :: (*int*, *'m*) *bind*
and *fail* :: *'m fail*
and *get* :: (*'v* ⇀ *int*, *'m*) *get*
and *put* :: (*'v* ⇀ *int*, *'m*) *put*
begin

definition *lookup* :: *'v* ⇒ *'m*
where $lookup\ x = get\ (\lambda s.\ case\ s\ x\ of\ None \Rightarrow fail \mid Some\ y \Rightarrow return\ y)$

lemma *lookup-alt-def*: $lookup\ x = get\ (\lambda s.\ case\ apply\ s\ x\ of\ None \Rightarrow fail \mid Some\ y \Rightarrow return\ y)$
⟨*proof*⟩

end

locale *prob-exp-base* =
memo-exp-base return bind fail get put +
monad-prob-base return bind sample
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 *sample* :: (*int*, *'m*) *sample*
begin

definition *sample-var* :: (*'v* ⇒ *int pmf*) ⇒ *'v* ⇒ *'m*
where *sample-var X x* = *sample* (*X x*) *return*

definition *lazy* :: (*'v* ⇒ *int pmf*) ⇒ *'v exp* ⇒ *'m*
where *lazy X* ≡ *eval* (*memo* (*sample-var X*))

definition *sample-vars* :: (*'v* ⇒ *int pmf*) ⇒ *'v set* ⇒ *'m* ⇒ *'m*
where *sample-vars X A m* = *Finite-Set.fold* (λ*x m*. *bind* (*memo* (*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* :: $('v \rightharpoonup int, 'm)$ *get*
and *put* :: $('v \rightharpoonup int, 'm)$ *put*
and *sample* :: (*int*, $'m$) *sample*
begin

lemma *comp-fun-commute-sample-var*: *comp-fun-commute* ($\lambda x\ m$. *bind* (*memo* (*sample-var X*) *x*) (λ-. *m*))
⟨*proof*⟩

interpretation *sample-var*: *comp-fun-commute* $\lambda x\ m$. *bind* (*memo* (*sample-var* *X*) *x*) (λ-. *m*)
rewrites $\bigwedge X\ m\ A$. *Finite-Set.fold* ($\lambda x\ m$. *bind* (*memo* (*sample-var X*) *x*) (λ-. *m*)) *m A* $\equiv$ *sample-vars X A m*
for *X*
⟨*proof*⟩

lemma *comp-fun-idem-sample-var*: *comp-fun-idem* ($\lambda x\ m$. *bind* (*memo* (*sample-var* *X*) *x*) (λ-. *m*))
⟨*proof*⟩

interpretation *sample-var*: *comp-fun-idem* $\lambda x\ m$. *bind* (*memo* (*sample-var X*) *x*) (λ-. *m*)
rewrites $\bigwedge X\ m\ A$. *Finite-Set.fold* ($\lambda x\ m$. *bind* (*memo* (*sample-var X*) *x*) (λ-. *m*)) *m A* $\equiv$ *sample-vars X A m*
for *X*
⟨*proof*⟩

lemma *sample-vars-empty* [*simp*]: *sample-vars X* {} *m* = *m*
⟨*proof*⟩

lemma *sample-vars-insert*:
finite $A \Longrightarrow$ *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 $A \Longrightarrow$ *sample-vars* X (*insert* x A) m = *sample-vars* X A (*bind* (*memo* (*sample-var* X) x) (λ-. m))
⟨*proof*⟩

lemma *sample-vars-union*:
⟦ *finite* A; *finite* B ⟧ $\Longrightarrow$ *sample-vars* X ($A \cup B$) m = *sample-vars* X A (*sample-vars* X B m)
⟨*proof*⟩

lemma *memo-lookup*:
bind (*memo* f x) (λi. *bind* (*lookup* x) (g i)) = *bind* (*memo* f x) (λi. 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* (λx. *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* × ('b ⇀ *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* × ('b ⇀ *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* (λ*s*. *case s x of None* ⇒ *fail* | *Some y* ⇒ *return y*)

lemma *lookup-alt-def*:
lookup x = *ask* (λ*s*. *case apply s x of None* ⇒ *fail* | *Some y* ⇒ *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* (λ*b*. (*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* (λ*x*. *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* ⇀ *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* (λ*x*. *pmf-of-set* {*0*, *1*}) (*Div* (*Const 2*) (*Var* (*CHR* ′′*x*′′)))) *Map.empty*

value *run-state* (*SFNI-lazy* (λ*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* ⇒ ′*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* ⇒ ′*m* **where**
hanoi 0 = *return* ()

| *hanoi* (*Suc n*) = *bind* (*hanoi n*) (λ-. *tick* (*hanoi n*))

primrec *repeat* :: *nat* ⇒ ′*m* ⇒ ′*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* ^ *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* ⇒ ′*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* ⇒ ′*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