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* ⟧ ⟹ *Finite-Set.fold f x* (*A* ∪ *B*) = *Finite-Set.fold f* (*Finite-Set.fold f x A*) *B*
by(*induction A arbitrary*: *x rule*: *finite-induct*)(*simp-all add*: *fold-insert-idem2 del*: *fold-insert-idem*)

lemma (**in** *comp-fun-idem*) *ffold-set-union*: *ffold f x* (*A* |∪| *B*) = *ffold f* (*ffold f x A*) *B*
including *fset.lifting* **by**(*transfer fixing*: *f*)(*rule fold-set-union*)

lemma *relcompp-top-top* [*simp*]: *top OO top* = *top*
by(*auto simp add*: *fun-eq-iff*)

attribute-setup *locale-witness* = ‹*Scan.succeed Locale.witness-add*›

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*)
by(*simp add*: *rel-itself.simps*)
lemma *plus-multiset-parametric* [*transfer-rule*]:
(*rel-mset A* ===> *rel-mset A* ===> *rel-mset A*) (+) (+)
apply(*rule rel-funI*)+
subgoal premises *prems* **using** *prems* **by** *induction*(*auto intro*: *rel-mset-Plus*)
done

lemma *Mempty-parametric* [*transfer-rule*]: *rel-mset A* {#} {#}
by(*fact rel-mset-Zero*)

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(*rule rel-funI*)+

```
  interpret f1: comp-fun-commute f1 by fact
  interpret f2: comp-fun-commute f2 by fact

  show B (fold-mset f1 z1 X) (fold-mset f2 z2 Y)
    if rel-mset A X Y B z1 z2 for z1 z2 X Y
   using that(1) by(induction R≡A X Y)(simp-all add: that(2) 12[THEN rel-funD,
THEN rel-funD])
qed

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: ⋀X Y x y. ⟦ rel-fset A X Y; P X Y; A x y; x |∉| X ∨ y |∉| Y ⟧ ⟹
P (finsert x X) (finsert y Y)
  shows P X Y
proof −
  from XY obtain Z where X: X = fst |`| Z and Y: Y = snd |`| Z and Z:
fBall Z (λ(x, y). A x y)
    unfolding fset.in-rel by auto
  from Z show ?thesis unfolding X Y
  proof(induction Z)
    case (insert xy Z)
    obtain x y where [simp]: xy = (x, y) by(cases xy)
    show ?case using insert
      apply(cases x |∈| fst |`| Z ∧ y |∈| snd |`| Z)
       apply(simp add: finsert-absorb)
      apply(auto intro!: step simp add: fset.in-rel; blast)
      done
  qed(simp add: assms)
qed

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(rule rel-funI)+
  interpret f1: comp-fun-idem f1 by fact
  interpret f2: comp-fun-idem f2 by fact

  show B (ffold f1 z1 X) (ffold f2 z2 Y)
    if rel-fset A X Y B z1 z2 for z1 z2 X Y
    using that(1) by(induction)(simp-all add: that(2) 12[THEN rel-funD, THEN
rel-funD])
qed

end

lemma rel-set-Grp: rel-set (BNF-Def.Grp A f) = BNF-Def.Grp {X. X ⊆ A}
(image f)
```

by(*auto simp add*: *fun-eq-iff Grp-def rel-set-def*)

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

lemma *cUNION-assoc*: *cUNION* (*cUNION A f*) *g* = *cUNION A* (*λx. cUNION* (*f x*) *g*)
by *transfer auto*

lemma *cUnion-cempty* [*simp*]: *cUnion cempty* = *cempty*
by *transfer simp*

lemma *cUNION-cempty* [*simp*]: *cUNION cempty f* = *cempty*
by *simp*

lemma *cUnion-cinsert*: *cUnion* (*cinsert x A*) = *cUn x* (*cUnion A*)
by *transfer simp*

lemma *cUNION-cinsert*: *cUNION* (*cinsert x A*) *f* = *cUn* (*f x*) (*cUNION A f*)
by (*simp add*: *cUnion-cinsert*)

lemma *cUnion-csingle* [*simp*]: *cUnion* (*csingle x*) = *x*
by (*simp add*: *cUnion-cinsert*)

lemma *cUNION-csingle* [*simp*]: *cUNION* (*csingle x*) *f* = *f x*
by *simp*

lemma *cUNION-csingle2* [*simp*]: *cUNION A csingle* = *A*
by (*fact cUN-csingleton*)

lemma *cUNION-cUn*: *cUNION* (*cUn A B*) *f* = *cUn* (*cUNION A f*) (*cUNION B f*)
by *simp*

lemma *cUNION-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
(*rel-cset A* ===> (*A* ===> *rel-cset B*) ===> *rel-cset B*) *cUNION cUNION*
unfolding *rel-fun-def* **by** *transfer*(*blast intro*: *rel-set-UNION*)

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 −
  have **1** ≠ **2** ∧ **1** ≠ **3** ∧ **2** ≠ **3**
    unfolding *three*$_1$*-def three*$_2$*-def three*$_3$*-def threes-def split-def*
    by(*rule someI-ex*)(*use ex-three* **in** *auto*)
  then show **1** ≠ **2** **1** ≠ **3** **2** ≠ **3** **by** *simp-all*
qed

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

inductive *rel-12-23* :: ′*a* ⇒ ′*a* ⇒ *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*
  by(*auto simp add*: *bi-unique-def rel-12-23.simps*)

inductive *rel-12-21* :: ′*a* ⇒ ′*a* ⇒ *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*
  by(*auto simp add*: *bi-unique-def rel-12-21.simps*)

end

lemma *bernoulli-pmf-0*: *bernoulli-pmf 0* = *return-pmf False*
  by(*rule pmf-eqI*)(*simp split*: *split-indicator*)

lemma *bernoulli-pmf-1*: *bernoulli-pmf 1* = *return-pmf True*
  by(*rule pmf-eqI*)(*simp split*: *split-indicator*)

lemma *bernoulli-Not*: *map-pmf Not* (*bernoulli-pmf r*) = *bernoulli-pmf* (*1* − *r*)
  apply(*rule pmf-eqI*)
  apply(*rewrite* **in** *pmf* - ⌑ = - *not-not*[*symmetric*])
  apply(*subst pmf-map-inj*′)
  apply(*simp-all add*: *inj-on-def bernoulli-pmf.rep-eq min-def max-def*)
  done

lemma *pmf-eqI-avoid*: *p* = *q* **if** ⋀*i*. *i* ≠ *x* ⟹ *pmf p i* = *pmf q i*
proof(*rule pmf-eqI*)
  show *pmf p i* = *pmf q i* **for** *i*
  proof(*cases i* = *x*)
    case [*simp*]: *True*
    have *pmf p i* = *measure-pmf.prob p* {*i*} **by**(*simp add*: *measure-pmf-single*)
    also have ... = *1* − *measure-pmf.prob p* (*UNIV* − {*i*})
      by(*subst measure-pmf.prob-compl*[*unfolded space-measure-pmf*]) *simp-all*
    also have *measure-pmf.prob p* (*UNIV* − {*i*}) = *measure-pmf.prob q* (*UNIV*

$- \{i\})$
unfolding *integral-pmf*[*symmetric*] **by**(*rule Bochner-Integration.integral-cong*)(*auto intro*: *that*)
also have *1* − ... = *measure-pmf.prob q* {*i*}
by(*subst measure-pmf.prob-compl*[*unfolded space-measure-pmf*]) *simp-all*
also have ... = *pmf q i* **by**(*simp add*: *measure-pmf-single*)
finally show *?thesis* **.**
next
case *False*
then show *?thesis* **by**(*rule that*)
qed
qed

2 Locales for monomorphic monads

2.1 Plain monad

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

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

primrec *sequence* :: *'m list* ⇒ (*'a list* ⇒ *'m*) ⇒ *'m*
where
sequence [] *f* = *f* []
| *sequence* (*x* # *xs*) *f* = *bind x* (λ*a*. *sequence xs* (*f* ∘ (#) *a*))

definition *lift* :: (*'a* ⇒ *'a*) ⇒ *'m* ⇒ *'m*
where *lift f x* = *bind x* (λ*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*
unfolding *monad-base.sequence-def*[*abs-def*] **by** *transfer-prover*

lemma *lift-parametric* [*transfer-rule*]:
((*A* ===> *M*) ===> (*M* ===> (*A* ===> *M*) ===> *M*) ===> (*A* ===> *A*) ===> *M* ===> *M*) *monad-base.lift monad-base.lift*

unfolding *monad-base.lift-def* **by** *transfer-prover*

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*]: $bind\ (lift\ f\ x)\ g = bind\ x\ (g \circ f)$
by(*simp add*: *lift-def bind-assoc return-bind o-def*)

lemma *lift-bind* [*simp*]: $lift\ f\ (bind\ m\ g) = bind\ m\ (\lambda x.\ lift\ f\ (g\ x))$
by(*simp add*: *lift-def bind-assoc*)

end

2.2 State

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

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

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

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
unfolding *monad-state-base.update-def* **by** *transfer-prover*

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

for *return* :: ($'a$, $'m$) *return*
and *bind* :: ($'a$, $'m$) *bind*
and *get* :: ($'s$, $'m$) *get*
and *put* :: ($'s$, $'m$) *put*
+
assumes *put-get*: $\bigwedge f$. *put s* (*get f*) = *put s* (*f s*)
and *get-get*: $\bigwedge f$. *get* (λs. *get* (*f s*)) = *get* (λs. *f s s*)
and *put-put*: *put s* (*put s' m*) = *put s' m*
and *get-put*: *get* (λs. *put s m*) = *m*
and *get-const*: $\bigwedge m$. *get* (λ-. *m*) = *m*
and *bind-get*: $\bigwedge f\ g$. *bind* (*get f*) *g* = *get* (λs. *bind* (*f s*) *g*)
and *bind-put*: $\bigwedge f$. *bind* (*put s m*) *f* = *put s* (*bind m f*)
begin

lemma *put-update*: *put s* (*update f m*) = *put* (*f s*) *m*
by(*simp add*: *update-def put-get put-put*)

lemma *update-put*: *update f* (*put s m*) = *put s m*
by(*simp add*: *update-def put-put get-const*)

lemma *bind-update*: *bind* (*update f m*) *g* = *update f* (*bind m g*)
by(*simp add*: *update-def bind-get bind-put*)

lemma *update-get*: *update f* (*get g*) = *get* (*update f* ∘ *g* ∘ *f*)
by(*simp add*: *update-def put-get get-get o-def*)

lemma *update-const*: *update* (λ-. *s*) *m* = *put s m*
by(*simp add*: *update-def get-const*)

lemma *update-update*: *update f* (*update g m*) = *update* (*g* ∘ *f*) *m*
by(*simp add*: *update-def put-get put-put*)

lemma *update-id*: *update id m* = *m*
by(*simp add*: *update-def get-put*)

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* (*λx. 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
unfolding *monad-fail-base.assert-def* **by** *transfer-prover*

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*
by(*simp add*: *assert-def fail-bind*)

end

2.4 Exception

type-synonym *'m catch* = *'m* ⇒ *'m* ⇒ *'m*

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

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

locale *monad-catch-state* = *monad-catch return bind fail catch* + *monad-state re-*

turn bind get put
for *return* :: (′*a*, ′*m*) *return*
and *bind* :: (′*a*, ′*m*) *bind*
and *fail* :: ′*m fail*
and *catch* :: ′*m catch*
and *get* :: (′*s*, ′*m*) *get*
and *put* :: (′*s*, ′*m*) *put*
+
assumes *catch-get*: *catch* (*get f*) *m* = *get* (λ*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*′)
by(*simp add*: *update-def catch-get catch-put*)

end

2.5 Reader

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

type-synonym (′*r*, ′*m*) *ask* = (′*r* ⇒ ′*m*) ⇒ ′*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*: ⋀*f*. *ask* (λ*r*. *ask* (*f r*)) = *ask* (λ*r*. *f r r*)
and *ask-const*: *ask* (λ-. *m*) = *m*
and *bind-ask*: ⋀*f g*. *bind* (*ask f*) *g* = *ask* (λ*r*. *bind* (*f r*) *g*)
and *bind-ask2*: ⋀*f*. *bind m* (λ*x*. *ask* (*f x*)) = *ask* (λ*r*. *bind m* (λ*x*. *f x r*))
begin

lemma *ask-bind*: *ask* (λ*r*. *bind* (*f r*) (*g r*)) = *bind* (*ask f*) (λ*x*. *ask* (λ*r*. *g r x*))
by(*simp add*: *bind-ask bind-ask2 ask-ask*)

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** $p = q$
by(*rule sample-parametric*[**where** *R=eq-onp* $(\lambda x.\ x \in set\text{-}pmf\ p)$, *THEN rel-funD*, *THEN rel-funD*])
(*simp-all add*: *bi-unique-def eq-onp-def rel-fun-def pmf.rel-refl-strong that*)

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* ⇒ '*m* ⇒ '*m* ⇒ '*m* **where**
pchoose r m m' = *sample* (*map-pmf* (λ*b*. *if b then* **1** *else* **2**) (*bernoulli-pmf r*)) (λ*x*. *if x* = **1** *then m else m'*)

abbreviation *pchoose-syntax* :: '*m* ⇒ *real* ⇒ '*m* ⇒ '*m* (‹- ◁ - ▷ -› [*100*, *0*, *100*] *99*) **where**
m ◁ *r* ▷ *m'* ≡ *pchoose r m m'*

lemma *pchoose-0*: *m* ◁ *0* ▷ *m'* = *m'*
by(*simp add*: *pchoose-def bernoulli-pmf-0 sample-return-pmf*)

lemma *pchoose-1*: *m* ◁ *1* ▷ *m'* = *m*
by(*simp add*: *pchoose-def bernoulli-pmf-1 sample-return-pmf*)

lemma *pchoose-idemp*: *m* ◁ *r* ▷ *m* = *m*
by(*simp add*: *pchoose-def sample-const*)

lemma *pchoose-bind1*: *bind* (*m* ◁ *r* ▷ *m'*) *f* = *bind m f* ◁ *r* ▷ *bind m' f*
by(*simp add*: *pchoose-def bind-sample1 if-distrib*[**where** *f*=λ*m*. *bind m* -])

lemma *pchoose-bind2*: *bind m* (λ*x*. *f x* ◁ *p* ▷ *g x*) = *bind m f* ◁ *p* ▷ *bind m g*
by(*auto simp add*: *pchoose-def bind-sample2 intro*!: *arg-cong2*[**where** *f*=*sample*])

lemma *pchoose-commute*: *m* ◁ *1* − *r* ▷ *m'* = *m'* ◁ *r* ▷ *m*
apply(*simp add*: *pchoose-def bernoulli-Not*[*symmetric*] *pmf.map-comp o-def*)
apply(*rule sample-parametric*[**where** *R*=*rel-12-21*, *THEN rel-funD*, *THEN rel-funD*])
subgoal by(*simp*)
subgoal by(*rule pmf.map-transfer*[**where** *Rb*=(=), *THEN rel-funD*, *THEN rel-funD*])
(*simp-all add*: *rel-fun-def rel-12-21.simps pmf.rel-eq*)
subgoal by(*simp add*: *rel-fun-def rel-12-21.simps*)
done

lemma *pchoose-assoc*: *m* ◁ *p* ▷ (*m'* ◁ *q* ▷ *m''*) = (*m* ◁ *r* ▷ *m'*) ◁ *s* ▷ *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 −
let *?f* = (λ*x*. *if x* = **1** *then m else if x* = **2** *then m' else m''*)
let *?p* = *bind-pmf* (*map-pmf* (λ*b*. *if b then* **1** *else* **2**) (*bernoulli-pmf p*))
(λ*x*. *if x* = **1** *then return-pmf* **1** *else map-pmf* (λ*b*. *if b then* **2** *else* **3**) (*bernoulli-pmf q*))
let *?q* = *bind-pmf* (*map-pmf* (λ*b*. *if b then* **1** *else* **2**) (*bernoulli-pmf s*))
(λ*x*. *if x* = **1** *then map-pmf* (λ*b*. *if b then* **1** *else* **2**) (*bernoulli-pmf r*) *else return-pmf* **3**)

have [*simp*]: {*x*. ¬ *x*} = {*False*} {*x*. *x*} = {*True*} **by** *auto*

have *?lhs* = *sample ?p ?f*
by(*auto simp add*: *pchoose-def sample-bind-pmf if-distrib*[**where** *f*=λ*x*. *sample x* -] *sample-return-pmf rel-fun-def rel-12-23*.*simps pmf*.*rel-eq cong*: *if-cong intro*!: *sample-cong*[*OF refl*] *sample-parametric*[**where** *R*=*rel-12-23*, *THEN rel-funD*, *THEN rel-funD*] *pmf*.*map-transfer*[**where** *Rb*=(=), *THEN rel-funD*, *THEN rel-funD*])
also have *?p* = *?q*
proof(*rule pmf-eqI-avoid*)
fix *i* :: ′*p*
assume *i* ≠ **2**
then consider (*one*) *i* = **1** | (*three*) *i* = **3** | (*other*) *i* ≠ **1** *i* ≠ **2** *i* ≠ **3** **by** *metis*
then show *pmf ?p i* = *pmf ?q i*
proof *cases*
case [*simp*]: *one*
have *pmf ?p i* = *measure-pmf*.*expectation* (*map-pmf* (λ*b*. *if b then* **1** *else* **2**) (*bernoulli-pmf p*)) (*indicator* {**1**})
unfolding *pmf-bind*
by(*rule arg-cong2*[**where** *f*=*measure-pmf*.*expectation*, *OF refl*])(*auto simp add*: *fun-eq-iff pmf-eq-0-set-pmf*)
also have . . . = *min 1* (*max 0 p*)
by(*simp add*: *vimage-def*)(*simp add*: *measure-pmf-single bernoulli-pmf*.*rep-eq*)
also have . . . = *min 1* (*max 0 s*) ∗ *min 1* (*max 0 r*) **using** *that*(*1*) **by** *simp*
also have . . . = *measure-pmf*.*expectation* (*bernoulli-pmf s*)
(λ*x*. *indicator* {*True*} *x* ∗ *pmf* (*map-pmf* (λ*b*. *if b then* **1** *else* **2**) (*bernoulli-pmf r*)) **1**)
by(*simp add*: *pmf-map vimage-def measure-pmf-single*)(*simp add*: *bernoulli-pmf*.*rep-eq*)
also have . . . = *pmf ?q i*
unfolding *pmf-bind integral-map-pmf*
by(*rule arg-cong2*[**where** *f*=*measure-pmf*.*expectation*, *OF refl*])(*auto simp add*: *fun-eq-iff pmf-eq-0-set-pmf*)
finally show *?thesis* **.**
next
case [*simp*]: *three*
have *pmf ?p i* = *measure-pmf*.*expectation* (*bernoulli-pmf p*)
(λ*x*. *indicator* {*False*} *x* ∗ *pmf* (*map-pmf* (λ*b*. *if b then* **2** *else* **3**) (*bernoulli-pmf q*)) **3**)
unfolding *pmf-bind integral-map-pmf*
by(*rule arg-cong2*[**where** *f*=*measure-pmf*.*expectation*, *OF refl*])(*auto simp add*: *fun-eq-iff pmf-eq-0-set-pmf*)
also have . . . = (*1* − *min 1* (*max 0 p*)) ∗ (*1* − *min 1* (*max 0 q*))
by(*simp add*: *pmf-map vimage-def measure-pmf-single*)(*simp add*: *bernoulli-pmf*.*rep-eq*)
also have . . . = *1* − *min 1* (*max 0 s*) **using** *that*(*2*) **by** *simp*
also have . . . = *measure-pmf*.*expectation* (*map-pmf* (λ*b*. *if b then* **1** *else* **2**) (*bernoulli-pmf s*)) (*indicator* {**2**})
by(*simp add*: *vimage-def*)(*simp add*: *measure-pmf-single bernoulli-pmf*.*rep-eq*)
also have . . . = *pmf ?q i*
unfolding *pmf-bind*
by(*rule Bochner-Integration*.*integral-cong-AE*)(*auto simp add*: *fun-eq-iff*

pmf-eq-0-set-pmf AE-measure-pmf-iff)
      finally show *?thesis* **.**
    next
      case *other*
      then have *pmf ?p i = 0 pmf ?q i = 0* **by**(*auto simp add*: *pmf-eq-0-set-pmf*)
      then show *?thesis* **by** *simp*
    qed
  qed
  also have *sample ?q ?f = ?rhs*
    by(*auto simp add*: *pchoose-def sample-bind-pmf if-distrib*[**where** *f*=λ*x*. *sample x* -] *sample-return-pmf cong*: *if-cong intro*!: *sample-cong*[*OF refl*])
  finally show *?thesis* **.**
qed

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 $0 \leq p$ $p \leq 1$ $0 \leq q$ $q \leq 1$ $0 \leq r$ $r \leq 1$ $0 \leq s$ $s \leq 1$
  by(*rule pchoose-assoc*; *use that* **in** ‹*simp add*: *min-def max-def*›)

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* (λ*x*. *get* (*f x*)) = *get* (λ*s*. *sample p* (λ*x*. *f x s*))
begin

lemma *sample-put*: *sample p* (λ*x*. *put s* (*m x*)) = *put s* (*sample p m*)
proof −
  fix *UU*
  have *sample p* (λ*x*. *put s* (*m x*)) = *sample p* (λ*x*. *bind* (*put s* (*return UU*)) (λ-. *m x*))
    by(*simp add*: *bind-put return-bind*)
  also have ... = *bind* (*put s* (*return UU*)) (λ-. *sample p m*)
    by(*simp add*: *bind-sample2*)
  also have ... = *put s* (*sample p m*)
    by(*simp add*: *bind-put return-bind*)
  finally show *?thesis* **.**
qed

lemma *sample-update*: *sample p* (λ*x*. *update f* (*m x*)) = *update f* (*sample p m*)
by(*simp add*: *update-def sample-get sample-put*)

end

2.7 Nondeterministic choice

2.7.1 Binary choice

type-synonym *'m alt* = *'m* ⇒ *'m* ⇒ *'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'*)
by(*simp add*: *assert-def bind-alt1*)

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* (λ*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'*)

by(*simp add*: *update-def alt-get alt-put*)

end

2.7.2 Countable choice

type-synonym ('*c*, '*m*) *altc* = '*c cset* ⇒ ('*c* ⇒ '*m*) ⇒ '*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*: ⋀*C g f*. *bind* (*altc C g*) *f* = *altc C* (λ*c*. *bind* (*g c*) *f*)
  and *altc-single*: ⋀*x f*. *altc* (*csingle x*) *f* = *f x*
  and *altc-cUNION*: ⋀*C f g*. *altc* (*cUNION C f*) *g* = *altc C* (λ*x*. *altc* (*f x*) *g*)
  — We do not assume *altc-const* like for *sample* because the choice set might be empty
  and *altc-parametric*: ⋀*R*. *bi-unique R* ⟹ *rel-fun* (*rel-cset R*) (*rel-fun* (*rel-fun R* (=)) (=)) *altc altc*
begin

lemma *altc-cong*: *cBall C* (λ*x*. *f x* = *g x*) ⟹ *altc C f* = *altc C g*
  apply(*rule altc-parametric*[**where** *R*=*eq-onp* (λ*x*. *cin x C*), *THEN rel-funD*, *THEN rel-funD*])
  subgoal by(*simp add*: *bi-unique-def eq-onp-def*)
  subgoal by(*simp add*: *cset.rel-eq-onp eq-onp-same-args pred-cset-def cin-def*)
  subgoal by(*simp add*: *rel-fun-def eq-onp-def cBall-def cin-def*)
  done

lemma *monad-fail* [*locale-witness*]: *monad-fail return bind fail*
proof
  show *bind fail f* = *fail* **for** *f*
    by(*simp add*: *fail-def bind-altc1 cong*: *altc-cong*)
qed

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

lemma *monad-alt*: *monad-alt return bind alt*
proof
  show *bind* (*alt m m'*) *f* = *alt* (*bind m f*) (*bind m' f*) **for** *m m' f*
    by(*simp add*: *alt-def bind-altc1 if-distrib*[**where** *f*=*λm. bind m -*])

  fix *m1 m2 m3* :: *'m*
  let *?C* = *cUNION* (*cinsert* **1** (*csingle* **2**)) (*λc. if c* = **1** *then cinsert* **1** (*csingle* **2**) *else csingle* **3**)
  let *?D* = *cUNION* (*cinsert* **1** (*csingle* **2**)) (*λc. if c* = **1** *then csingle* **1** *else cinsert* **2** (*csingle* **3**))
  let *?f* = *λc. if c* = **1** *then m1 else if c* = **2** *then m2 else m3*
  have *alt* (*alt m1 m2*) *m3* = *altc ?C ?f*
   by (*simp only*: *altc-cUNION*) (*auto simp add*: *alt-def altc-single intro*!: *altc-cong*)
   also have *?C* = *?D* **including** *cset.lifting* **by** *transfer*(*auto simp add*: *insert-commute*)
  also have *altc ?D ?f* = *alt m1* (*alt m2 m3*)
    apply (*simp only*: *altc-cUNION*)
    apply (*clarsimp simp add*: *alt-def altc-single intro*!: *altc-cong*)
    apply (*rule altc-parametric* [**where** *R*=*conversep rel-12-23*, *THEN rel-funD*, *THEN rel-funD*])
    subgoal by *simp*
    subgoal including *cset.lifting* **by** *transfer*
      (*simp add*: *rel-set-def rel-12-23.simps*)
    subgoal by (*simp add*: *rel-fun-def rel-12-23.simps*)
    done
  finally show *alt* (*alt m1 m2*) *m3* = *alt m1* (*alt m2 m3*) **.**
qed

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

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*: $bind\ (pause\ out\ c)\ f = pause\ out\ (\lambda i.\ bind\ (c\ i)\ f)$

2.10 Commutative monad

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

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* (λ*x*. *bind m* (*f x*)) = *bind m* (λ*x*. *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*′ ⟷ (∃ *x*′. *m*′ = *return-id x*′ ∧ *A x x*′)
  rel-id A m (*return-id x*′) ⟷ (∃ *x*. *m* = *return-id x* ∧ *A x x*′)
  subgoal by(*cases m*′; *simp*)
  subgoal by(*cases m*; *simp*)
  done

lemma *rel-id-expand*: *M* (*extract m*) (*extract m*′) ⟹ *rel-id M m m*′
  by(*cases m*; *cases m*′; *simp*)

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*))
by(*cases x*) *simp*

lemma *bind-id-parametric* [*transfer-rule*]: **includes** *lifting-syntax* **shows**
  (*rel-id A* ===> (*A* ===> *rel-id A*) ===> *rel-id A*) *bind-id bind-id*
unfolding *bind-id-def* **by** *transfer-prover*

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

show *bind-id* (*bind-id x f*) *g* = *bind-id x* (*λx. bind-id* (*f x*) *g*)
for *x* :: *'a id* **and** *f* :: *'a* ⇒ *'a id* **and** *g* :: *'a* ⇒ *'a id*
by(*rule id.expand*) *simp*
show *bind-id* (*return-id x*) *f* = *f x* **for** *f* :: *'a* ⇒ *'a id* **and** *x*
by(*rule id.expand*) *simp*
show *bind-id x return-id* = *x* **for** *x* :: *'a id*
by(*rule id.expand*) *simp*
qed

lemma *monad-commute-id* [*locale-witness*]: *monad-commute return-id bind-id*
proof
show *bind-id m* (*λx. bind-id m'* (*f x*)) = *bind-id m'* (*λy. bind-id m* (*λx. f x y*))
for *m m'* :: *'a id* **and** *f*
by(*rule id.expand*) *simp*
qed

lemma *monad-discard-id* [*locale-witness*]: *monad-discard return-id bind-id*
proof
show *bind-id m* (*λ-. m'*) = *m'* **for** *m m'* :: *'a id* **by**(*rule id.expand*) *simp*
qed

lemma *monad-duplicate-id* [*locale-witness*]: *monad-duplicate return-id bind-id*
proof
show *bind-id m* (*λx. bind-id m* (*f x*)) = *bind-id m* (*λx. f x x*) **for** *m* :: *'a id* **and** *f*
by(*rule id.expand*) *simp*
qed

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*
by *unfold-locales*(*simp-all add*: *bind-assoc-pmf bind-return-pmf bind-return-pmf'*)

lemma *monad-prob-prob* [*locale-witness*]: *monad-prob return-pmf bind-pmf bind-pmf*
including *lifting-syntax*
proof
show *bind-pmf p* (*λ-. m*) = *m* **for** *p* :: *'b pmf* **and** *m* :: *'a prob*
by(*rule bind-pmf-const*)
show *bind-pmf* (*return-pmf x*) *f* = *f x* **for** *f* :: *'b* ⇒ *'a prob* **and** *x* **by**(*rule bind-return-pmf*)
show *bind-pmf* (*bind-pmf p f*) *g* = *bind-pmf p* (*λx. bind-pmf* (*f x*) *g*)
for *p* :: *'b pmf* **and** *f* :: *'b* ⇒ *'b pmf* **and** *g* :: *'b* ⇒ *'a prob*
by(*rule bind-assoc-pmf*)
show *bind-pmf p* (*λx. bind-pmf q* (*f x*)) = *bind-pmf q* (*λy. bind-pmf p* (*λx. f x y*))

for *p q* :: *'b pmf* **and** *f* :: *'b* ⇒ *'b* ⇒ *'a prob* **by**(*rule bind-commute-pmf*)
show *bind-pmf* (*bind-pmf p f*) *g* = *bind-pmf p* (λ*x*. *bind-pmf* (*f x*) *g*)
for *p* :: *'b pmf* **and** *f* :: *'b* ⇒ *'a prob* **and** *g* :: *'a* ⇒ *'a prob*
by(*simp add*: *bind-assoc-pmf*)
show *bind-pmf m* (λ*y*. *bind-pmf p* (*f y*)) = *bind-pmf p* (λ*x*. *bind-pmf m* (λ*y*. *f y x*))
for *m* :: *'a prob* **and** *p* :: *'b pmf* **and** *f* :: *'a* ⇒ *'b* ⇒ *'a prob*
by(*rule bind-commute-pmf*)
show (*rel-pmf R* ===> (*R* ===> (=)) ===> (=)) *bind-pmf bind-pmf* **for** *R* :: *'b* ⇒ *'b* ⇒ *bool*
by *transfer-prover*
qed

lemma *monad-commute-prob* [*locale-witness*]: *monad-commute return-pmf bind-pmf*
proof
show *bind-pmf m* (λ*x*. *bind-pmf m'* (*f x*)) = *bind-pmf m'* (λ*y*. *bind-pmf m* (λ*x*. *f x y*))
for *m m'* :: *'a prob* **and** *f* :: *'a* ⇒ *'a* ⇒ *'a prob*
by(*rule bind-commute-pmf*)
qed

lemma *monad-discard-prob* [*locale-witness*]: *monad-discard return-pmf bind-pmf*
proof
show *bind-pmf m* (λ-. *m'*) = *m'* **for** *m m'* :: *'a pmf* **by**(*simp*)
qed

3.3 Resumption

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

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

3.3.1 Plain monad

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

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

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

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

by(*simp add*: *return-resumption-def*)

lemma *result-return-resumption* [*simp*]: *result* (*return-resumption x*) = *x*
by(*simp add*: *return-resumption-def*)

lemma *monad-resumption* [*locale-witness*]: *monad return-resumption bind-resumption*
proof
show *bind-resumption* (*bind-resumption x f*) *g* = *bind-resumption x* (λ*y*. *bind-resumption* (*f y*) *g*)
for *x* :: ('*o*, '*i*, '*a*) *resumption* **and** *f g*
by(*coinduction arbitrary*: *x f g rule*: *resumption.coinduct-strong*) *auto*
show *bind-resumption* (*return-resumption x*) *f* = *f x* **for** *x* **and** *f* :: '*a* ⇒ ('*o*, '*i*, '*a*) *resumption*
by(*rule resumption.expand*)(*simp-all add*: *return-resumption-def*)
show *bind-resumption x return-resumption* = *x* **for** *x* :: ('*o*, '*i*, '*a*) *resumption*
by(*coinduction arbitrary*: *x rule*: *resumption.coinduct-strong*) *auto*
qed

lemma *monad-resumption-resumption* [*locale-witness*]:
monad-resumption return-resumption bind-resumption pause-resumption
proof
show *bind-resumption* (*pause-resumption out c*) *f* = *pause-resumption out* (λ*i*. *bind-resumption* (*c i*) *f*)
for *out c* **and** *f* :: '*a* ⇒ ('*o*, '*i*, '*a*) *resumption*
by(*rule resumption.expand*)(*simp-all add*: *pause-resumption-def*)
qed

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* = {}
by(*cases x*) *simp*

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

lemma *rel-optionT'-phantom*: *rel-optionT' A* = *rel-optionT' top*
by(*auto 4 4 intro*: *optionT.rel-mono antisym optionT.rel-mono-strong*)

lemma *map-optionT'-phantom*: *map-optionT' f* = *map-optionT' undefined*

by(*auto 4 4 intro*: *optionT.map-cong*)

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

definition *rel-optionT* :: $('m \Rightarrow 'm' \Rightarrow bool) \Rightarrow ('a, 'm)$ *optionT* $\Rightarrow ('b, 'm')$ *optionT* $\Rightarrow$ *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*
using *assms* **by**(*cases m*; *cases m'*; *simp add*: *rel-optionT-def*)

lemma *rel-optionT-simps* [*simp*]: *rel-optionT M* (*OptionT m*) (*OptionT m'*) $\longleftrightarrow$ *M m m'*
by(*simp add*: *rel-optionT-def*)

lemma *rel-optionT-eq* [*relator-eq*]: *rel-optionT* (=) = (=)
by(*auto simp add*: *fun-eq-iff rel-optionT-def intro*: *optionT.rel-refl-strong elim*: *optionT.rel-cases*)

lemma *rel-optionT-mono* [*relator-mono*]: *rel-optionT A* $\leq$ *rel-optionT B* **if** $A \leq B$
by(*simp add*: *rel-optionT-def optionT.rel-mono that*)

lemma *rel-optionT-distr* [*relator-distr*]: *rel-optionT A OO rel-optionT B* = *rel-optionT* (*A OO B*)
by(*simp add*: *rel-optionT-def optionT.rel-compp*[*symmetric*])

lemma *rel-optionT-Grp*: *rel-optionT* (*BNF-Def.Grp A f*) = *BNF-Def.Grp* {*x*. *set-optionT x* $\subseteq$ *A*} (*map-optionT f*)
by(*simp add*: *rel-optionT-def rel-optionT'-phantom*[*of BNF-Def.Grp UNIV undefined*, *symmetric*] *optionT.rel-Grp map-optionT-def*)

lemma *OptionT-parametric* [*transfer-rule*]: (*M* ===> *rel-optionT M*) *OptionT OptionT*
by(*simp add*: *rel-fun-def rel-optionT-def*)

lemma *run-option-parametric* [*transfer-rule*]: (*rel-optionT M* ===> *M*) *run-option run-option*
by(*auto simp add*: *rel-fun-def rel-optionT-def elim*: *optionT.rel-cases*)

lemma *case-optionT-parametric* [*transfer-rule*]:
  ((*M* ===> *X*) ===> *rel-optionT M* ===> *X*) *case-optionT case-optionT*
by(*auto simp add*: *rel-fun-def rel-optionT-def split*: *optionT.split*)

lemma *rec-optionT-parametric* [*transfer-rule*]:
  ((*M* ===> *X*) ===> *rel-optionT M* ===> *X*) *rec-optionT rec-optionT*
by(*auto simp add*: *rel-fun-def elim*: *rel-optionTE*)

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* ⇒ *return* (*None* :: *'a option*) | *Some y* ⇒ *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* ⇒ *return None* | *Some y* ⇒ *run-option* (*f y*))
by(*cases x*) *simp*

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

lemma *run-fail-option* [*simp*]: *run-option fail-option* = *return None*
by(*simp add*: *fail-option-def*)

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*)
by(*simp add*: *catch-option-def*)

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

interpretation *monad return bind* **by**(*rule monad*)

lemma *monad-optionT* [*locale-witness*]: *monad return-option bind-option* (**is** *monad*

?return ?bind)
proof
  show *?bind* (*?bind x f*) *g* = *?bind x* (*λx. ?bind* (*f x*) *g*) **for** *x f g*
   by(*rule optionT.expand*)(*auto simp add*: *bind-assoc run-bind-option return-bind intro*!: *arg-cong2*[**where** *f*=*bind*] *split*: *option.split*)
  show *?bind* (*?return x*) *f* = *f x* **for** *f x*
   by(*rule optionT.expand*)(*simp add*: *run-bind-option return-bind return-option-def*)
  show *?bind x ?return* = *x* **for** *x*
   by(*rule optionT.expand*)(*simp add*: *run-bind-option option.case-distrib*[*symmetric*] *case-option-id bind-return cong del*: *option.case-cong*)
qed

lemma *monad-fail-optionT* [*locale-witness*]:
  monad-fail return-option bind-option fail-option
proof
  show *bind-option fail-option f* = *fail-option* **for** *f*
   by(*rule optionT.expand*)(*simp add*: *run-bind-option return-bind*)
qed

lemma *monad-catch-optionT* [*locale-witness*]:
  monad-catch return-option bind-option fail-option catch-option
proof
  show *catch-option* (*return-option x*) *m* = *return-option x* **for** *x m*
   by(*rule optionT.expand*)(*simp add*: *return-bind*)
  show *catch-option fail-option m* = *m* **for** *m*
   by(*rule optionT.expand*)(*simp add*: *return-bind*)
  show *catch-option m fail-option* = *m* **for** *m*
   by(*rule optionT.expand*)(*simp add*: *bind-return if-distrib*[**where** *f*=*return*, *symmetric*] *cong del*: *if-weak-cong*)
  show *catch-option* (*catch-option m m′*) *m″* = *catch-option m* (*catch-option m′ m″*) **for** *m m′ m″*
   by(*rule optionT.expand*)(*auto simp add*: *bind-assoc fun-eq-iff return-bind intro*!: *arg-cong2*[**where** *f*=*bind*])
qed

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*))

by(*simp add*: *ask-option-def*)

lemma *monad-reader-optionT* [*locale-witness*]:
  assumes *monad-reader return bind ask*
  shows *monad-reader return-option bind-option ask-option*
proof −
  interpret *monad-reader return bind ask* **by**(*fact assms*)
  show *?thesis*
  proof
    show *ask-option* (*λr. ask-option* (*f r*)) = *ask-option* (*λr. f r r*) **for** *f*
      by(*rule optionT.expand*)(*simp add*: *ask-ask*)
    show *ask-option* (*λ-. x*) = *x* **for** *x*
      by(*rule optionT.expand*)(*simp add*: *ask-const*)
    show *bind-option* (*ask-option f*) *g* = *ask-option* (*λr. bind-option* (*f r*) *g*) **for** *f g*
      by(*rule optionT.expand*)(*simp add*: *bind-ask run-bind-option*)
    show *bind-option m* (*λx. ask-option* (*f x*)) = *ask-option* (*λr. bind-option m* (*λx. f x r*)) **for** *m f*
    by(*rule optionT.expand*)(*auto simp add*: *bind-ask2*[*symmetric*] *run-bind-option ask-const del*: *ext intro*!: *arg-cong2*[**where** *f*=*bind*] *ext split*: *option.split*)
  qed
qed

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*))
by(*simp add*: *get-option-def*)

lemma *run-put-option* [*simp*]:
  run-option (*put-option s m*) = *put s* (*run-option m*)
by(*cases m*)(*simp*)

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

interpretation *monad-state return bind get put* **by**(*fact state*)

lemma *monad-state-optionT* [*locale-witness*]:
monad-state return-option bind-option get-option put-option
proof
show *put-option s* (*get-option f*) = *put-option s* (*f s*) **for** *s f*
by(*rule optionT.expand*)(*simp add*: *put-get*)
show *get-option* (λ*s. get-option* (*f s*)) = *get-option* (λ*s. f s s*) **for** *f*
by(*rule optionT.expand*)(*simp add*: *get-get*)
show *put-option s* (*put-option s′ m*) = *put-option s′ m* **for** *s s′ m*
by(*rule optionT.expand*)(*simp add*: *put-put*)
show *get-option* (λ*s. put-option s m*) = *m* **for** *m*
by(*rule optionT.expand*)(*simp add*: *get-put*)
show *get-option* (λ-. *m*) = *m* **for** *m*
by(*rule optionT.expand*)(*simp add*: *get-const*)
show *bind-option* (*get-option f*) *g* = *get-option* (λ*s. bind-option* (*f s*) *g*) **for** *f g*
by(*rule optionT.expand*)(*simp add*: *bind-get run-bind-option*)
show *bind-option* (*put-option s m*) *f* = *put-option s* (*bind-option m f*) **for** *s m f*
by(*rule optionT.expand*)(*simp add*: *bind-put run-bind-option*)
qed

lemma *monad-catch-state-optionT* [*locale-witness*]:
monad-catch-state return-option bind-option fail-option catch-option get-option put-option
proof
show *catch-option* (*get-option f*) *m* = *get-option* (λ*s. catch-option* (*f s*) *m*) **for** *f m*
by(*rule optionT.expand*)(*simp add*: *bind-get*)
show *catch-option* (*put-option s m*) *m′* = *put-option s* (*catch-option m m′*) **for** *s m m′*
by(*rule optionT.expand*)(*simp add*: *bind-put*)
qed

end

3.4.4 Probability

definition *altc-sample-option* :: (′*x* ⇒ (′*b* ⇒ ′*m*) ⇒ ′*m*) ⇒ ′*x* ⇒ (′*b* ⇒ (′*a*, ′*m*) *optionT*) ⇒ (′*a*, ′*m*) *optionT*
where *altc-sample-option altc-sample p f* = *OptionT* (*altc-sample p* (λ*x. run-option* (*f x*)))

lemma *run-altc-sample-option* [*simp*]: *run-option* (*altc-sample-option altc-sample p f*) = *altc-sample p* (λ*x. run-option* (*f x*))
by(*simp add*: *altc-sample-option-def*)

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 −
  interpret monad-prob return bind sample by(fact assms)
  note sample-parametric[transfer-rule]
  show ?thesis including lifting-syntax
  proof
    show sample-option p (λ-. x) = x for p x
      by(rule optionT.expand)(simp add: sample-const)
    show sample-option (return-pmf x) f = f x for f x
      by(rule optionT.expand)(simp add: sample-return-pmf)
    show sample-option (bind-pmf p f) g = sample-option p (λx. sample-option (f
x) g) for p f g
      by(rule optionT.expand)(simp add: sample-bind-pmf)
     show sample-option p (λx. sample-option q (f x)) = sample-option q (λy.
sample-option p (λx. f x y)) for p q f
      by(rule optionT.expand)(auto intro!: sample-commute)
    show bind-option (sample-option p f) g = sample-option p (λx. bind-option (f
x) g) for p f g
      by(rule optionT.expand)(auto simp add: bind-sample1 run-bind-option)
      show bind-option m (λy. sample-option p (f y)) = sample-option p (λx.
bind-option m (λy. f y x)) for m p f
    by(rule optionT.expand)(auto simp add: bind-sample2[symmetric] run-bind-option
sample-const del: ext intro!: arg-cong2[where f=bind] ext split: option.split)
     show (rel-pmf R ===> (R ===> (=)) ===> (=)) sample-option sam-
ple-option
      if [transfer-rule]: bi-unique R for R
      unfolding altc-sample-option-def by transfer-prover
  qed
qed

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 sam-
ple-option
proof −
  interpret monad-state-prob return bind get put sample by fact
  show ?thesis
  proof
    show sample-option p (λx. get-option (f x)) = get-option (λs. sample-option p
(λx. f x s)) for p f
      by(rule optionT.expand)(simp add: sample-get)
  qed
```

qed

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*)
by(*simp add*: *tell-option-def*)

lemma *monad-writer-optionT* [*locale-witness*]:
  assumes *monad-writer return bind tell*
  shows *monad-writer return-option bind-option tell-option*
proof −
  interpret *monad-writer return bind tell* **by** *fact*
  show *?thesis*
  proof
    show *bind-option* (*tell-option w m*) *f* = *tell-option w* (*bind-option m f*) **for** *w m f*
      by(*rule optionT.expand*)(*simp add*: *run-bind-option bind-tell*)
  qed
qed

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*)
by(*simp add*: *alt-option-def*)

lemma *monad-alt-optionT* [*locale-witness*]:
  assumes *monad-alt return bind alt*
  shows *monad-alt return-option bind-option alt-option*
proof −
  interpret *monad-alt return bind alt* **by** *fact*

show *?thesis*
proof
show *alt-option* (*alt-option m1 m2*) *m3* = *alt-option m1* (*alt-option m2 m3*) **for** *m1 m2 m3*
by(*rule optionT.expand*)(*simp add*: *alt-assoc*)
show *bind-option* (*alt-option m m′*) *f* = *alt-option* (*bind-option m f*) (*bind-option m′ f*) **for** *m m′ f*
by(*rule optionT.expand*)(*simp add*: *bind-alt1 run-bind-option*)
qed
qed

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 −
interpret *monad-state-alt return bind get put alt* **by** *fact*
show *?thesis*
proof
show *alt-option* (*get-option f*) (*get-option g*) = *get-option* (*λx. alt-option* (*f x*) (*g x*))
for *f g* **by**(*rule optionT.expand*)(*simp add*: *alt-get*)
show *alt-option* (*put-option s m*) (*put-option s m′*) = *put-option s* (*alt-option m m′*)
for *s m m′* **by**(*rule optionT.expand*)(*simp add*: *alt-put*)
qed
qed

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 −
interpret *monad-altc return bind altc* **by** *fact*
note *altc-parametric*[*transfer-rule*]
show *?thesis* **including** *lifting-syntax*
proof
show *bind-option* (*altc-option C g*) *f* = *altc-option C* (*λc. bind-option* (*g c*) *f*)

for *C g f*
by(*rule optionT.expand*)(*simp add*: *run-bind-option bind-altc1 o-def*)
show *altc-option* (*csingle x*) *f* = *f x* **for** *x f*
by(*rule optionT.expand*)(*simp add*: *bind-altc1 altc-single*)
show *altc-option* (*cUNION C f*) *g* = *altc-option C* (λ*x*. *altc-option* (*f x*) *g*) **for** *C f g*
by(*rule optionT.expand*)(*simp add*: *bind-altc1 altc-cUNION o-def*)
show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) *altc-option altc-option*
if [*transfer-rule*]: *bi-unique R* **for** *R*
unfolding *altc-sample-option-def* **by** *transfer-prover*
qed
qed

lemma *monad-altc3-optionT* [*locale-witness*]:
assumes *monad-altc3 return bind altc*
shows *monad-altc3 return-option bind-option altc-option*
proof −
interpret *monad-altc3 return bind altc* **by** *fact*
show *?thesis* **..**
qed

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 −
interpret *monad-state-altc return bind get put altc* **by** *fact*
show *?thesis*
proof
show *altc-option C* (λ*c*. *get-option* (*f c*)) = *get-option* (λ*s*. *altc-option C* (λ*c*. *f c s*))
for *C f* **by**(*rule optionT.expand*)(*simp add*: *o-def altc-get*)
show *altc-option C* (λ*c*. *put-option s* (*f c*)) = *put-option s* (*altc-option C f*)
for *C s f* **by**(*rule optionT.expand*)(*simp add*: *o-def altc-put*)
qed
qed

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*))
by(*simp add*: *pause-option-def*)

lemma *monad-resumption-optionT* [*locale-witness*]:
  assumes *monad-resumption return bind pause*
  shows *monad-resumption return-option bind-option pause-option*
proof −
  interpret *monad-resumption return bind pause* **by** *fact*
  show *?thesis*
  proof
    show *bind-option* (*pause-option out c*) *f* = *pause-option out* (λ*i*. *bind-option* (*c i*) *f*) **for** *out c f*
      by(*rule optionT.expand*)(*simp add*: *bind-pause run-bind-option*)
  qed
qed

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 −
  interpret *monad-commute return bind* **by** *fact*
  interpret *monad-discard return bind* **by** *fact*
  show *?thesis*
  proof
    fix *m m′ f*
    have *run-option* (*bind-option m* (λ*x*. *bind-option m′* (*f x*))) =
      bind (*run-option m*) (λ*x*. *bind* (*run-option m′*) (λ*y*. *case* (*x*, *y*) *of* (*Some x′*, *Some y′*) ⇒ *run-option* (*f x′ y′*) | - ⇒ *return None*))
      by(*auto simp add*: *run-bind-option bind-const cong del*: *option.case-cong del*: *ext intro*!: *arg-cong2*[**where** *f*=*bind*] *ext split*: *option.split*)
    also have ... = *bind* (*run-option m′*) (λ*y*. *bind* (*run-option m*) (λ*x*. *case* (*x*, *y*) *of* (*Some x′*, *Some y′*) ⇒ *run-option* (*f x′ y′*) | - ⇒ *return None*))
      by(*rule bind-commute*)
    also have ... = *run-option* (*bind-option m′* (λ*y*. *bind-option m* (λ*x*. *f x y*)))
      by(*auto simp add*: *run-bind-option bind-const case-option-collapse cong del*: *option.case-cong del*: *ext intro*!: *arg-cong2*[**where** *f*=*bind*] *ext split*: *option.split*)
    finally show *bind-option m* (λ*x*. *bind-option m′* (*f x*)) = *bind-option m′* (λ*y*. *bind-option m* (λ*x*. *f x y*))
      by(*rule optionT.expand*)
  qed
qed

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 −
 interpret *monad-duplicate return bind* **by** *fact*
 interpret *monad-discard return bind* **by** *fact*
 show *?thesis*
 proof
  fix *m f*
  have *run-option* (*bind-option m* (λ*x*. *bind-option m* (*f x*))) =
   bind (*run-option m*) (λ*x*. *bind* (*run-option m*) (λ*y*. *case x of None* ⇒ *return None* | *Some x*′ ⇒ (*case y of None* ⇒ *return None* | *Some y*′ ⇒ *run-option* (*f x*′ *y*′))))
   by(*auto intro*!: *arg-cong2*[**where** *f*=*bind*] *simp add*: *fun-eq-iff bind-const run-bind-option split*: *option.split*)
  also have ... = *run-option* (*bind-option m* (λ*x*. *f x x*))
   by(*simp add*: *bind-duplicate run-bind-option cong*: *option.case-cong*)
  finally show *bind-option m* (λ*x*. *bind-option m* (*f x*)) = *bind-option m* (λ*x*. *f x x*)
   by(*rule optionT.expand*)
 qed
qed

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*
unfolding *return-option-def* **by** *transfer-prover*

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
unfolding *bind-option-def* **by** *transfer-prover*

lemma *fail-option-parametric* [*transfer-rule*]:
 ((*rel-option A* ===> *M*) ===> *rel-optionT M*) *fail-option fail-option*
unfolding *fail-option-def* **by** *transfer-prover*

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*
unfolding *catch-option-def* *Option.is-none-def*[*symmetric*] **by** *transfer-prover*

lemma *ask-option-parametric* [*transfer-rule*]:
(((*R* ===> *M*) ===> *M*) ===> (*R* ===> *rel-optionT* *M*) ===> *rel-optionT* *M*) *ask-option* *ask-option*
unfolding *ask-option-def* **by** *transfer-prover*

lemma *get-option-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *M*) ===> (*S* ===> *rel-optionT* *M*) ===> *rel-optionT* *M*) *get-option* *get-option*
unfolding *get-option-def* **by** *transfer-prover*

lemma *put-option-parametric* [*transfer-rule*]:
((*S* ===> *M* ===> *M*) ===> *S* ===> *rel-optionT* *M* ===> *rel-optionT* *M*) *put-option* *put-option*
unfolding *put-option-def* **by** *transfer-prover*

lemma *altc-sample-option-parametric* [*transfer-rule*]:
((*A* ===> (*P* ===> *M*) ===> *M*) ===> *A* ===> (*P* ===> *rel-optionT* *M*) ===> *rel-optionT* *M*)
altc-sample-option *altc-sample-option*
unfolding *altc-sample-option-def* **by** *transfer-prover*

lemma *alt-option-parametric* [*transfer-rule*]:
((*M* ===> *M* ===> *M*) ===> *rel-optionT* *M* ===> *rel-optionT* *M* ===> *rel-optionT* *M*) *alt-option* *alt-option*
unfolding *alt-option-def* **by** *transfer-prover*

lemma *tell-option-parametric* [*transfer-rule*]:
((*W* ===> *M* ===> *M*) ===> *W* ===> *rel-optionT* *M* ===> *rel-optionT* *M*) *tell-option* *tell-option*
unfolding *tell-option-def* **by** *transfer-prover*

lemma *pause-option-parametric* [*transfer-rule*]:
((*Out* ===> (*In* ===> *M*) ===> *M*) ===> *Out* ===> (*In* ===> *rel-optionT* *M*) ===> *rel-optionT* *M*)
pause-option *pause-option*
unfolding *pause-option-def* **by** *transfer-prover*

end

3.5 Reader monad transformer

datatype ('*r*, '*m*) *envT* = *EnvT* (*run-env*: '*r* ⇒ '*m*)

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

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

lemma *rel-envTI* [*intro*!]: (*R* ===> *M*) *f g* $\Longrightarrow$ *rel-envT R M* (*EnvT f*) (*EnvT g*)
by(*simp add*: *rel-envT-def BNF-Def.vimage2p-def*)

lemma *rel-envT-simps*: *rel-envT R M* (*EnvT f*) (*EnvT g*) $\longleftrightarrow$ (*R* ===> *M*) *f g*
by(*simp add*: *rel-envT-def BNF-Def.vimage2p-def*)

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*
using *assms* **by**(*cases m*; *cases m'*; *auto simp add*: *rel-envT-simps*)

lemma *rel-envT-eq* [*relator-eq*]: *rel-envT* (=) (=) = (=)
by(*auto simp add*: *rel-envT-def rel-fun-eq BNF-Def.vimage2p-def fun-eq-iff intro*: *envT.expand*)

lemma *rel-envT-mono* [*relator-mono*]: $[\![\ R \le R';\ M \le M'\ ]\!] \Longrightarrow$ *rel-envT R' M* $\le$ *rel-envT R M'*
by(*simp add*: *rel-envT-def predicate2I vimage2p-mono fun-mono*)

lemma *EnvT-parametric* [*transfer-rule*]: ((*R* ===> *M*) ===> *rel-envT R M*) *EnvT EnvT*
by(*simp add*: *rel-funI rel-envT-simps*)

lemma *run-env-parametric* [*transfer-rule*]: (*rel-envT R M* ===> *R* ===> *M*) *run-env run-env*
by(*auto elim*!: *rel-envTE*)

lemma *rec-envT-parametric* [*transfer-rule*]:
  (((*R* ===> *M*) ===> *X*) ===> *rel-envT R M* ===> *X*) *rec-envT rec-envT*
by(*auto 4 4 elim*!: *rel-envTE dest*: *rel-funD*)

lemma *case-envT-parametric* [*transfer-rule*]:
  (((*R* ===> *M*) ===> *X*) ===> *rel-envT R M* ===> *X*) *case-envT case-envT*
by(*auto 4 4 elim*!: *rel-envTE dest*: *rel-funD*)

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*)
by(*cases x*) *simp*

lemma *run-return-env* [*simp*]: *run-env* (*return-env x*) *r* = *return x*
by(*simp add*: *return-env-def*)

lemma *run-ask-env* [*simp*]: *run-env* (*ask-env f*) *r* = *run-env* (*f r*) *r*
by(*simp add*: *ask-env-def*)

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

interpretation *monad return bind* :: ('*a*, '*m*) *bind* **by**(*fact monad*)

lemma *monad-envT* [*locale-witness*]: *monad return-env bind-env*
proof
  show *bind-env* (*bind-env x f*) *g* = *bind-env x* (λ*x*. *bind-env* (*f x*) *g*)
    for *x* :: ('*r*, '*m*) *envT* **and** *f* :: '*a* ⇒ ('*r*, '*m*) *envT* **and** *g* :: '*a* ⇒ ('*r*, '*m*) *envT*
    by(*rule envT.expand*)(*auto simp add*: *bind-assoc return-bind*)
  show *bind-env* (*return-env x*) *f* = *f x* **for** *f* :: '*a* ⇒ ('*r*, '*m*) *envT* **and** *x*
    by(*rule envT.expand*)(*simp add*: *return-bind return-env-def*)
  show *bind-env x* (*return-env* :: ('*a*, ('*r*, '*m*) *envT*) *return*) = *x* **for** *x* :: ('*r*, '*m*) *envT*
    by(*rule envT.expand*)(*simp add*: *bind-return fun-eq-iff*)
qed

lemma *monad-reader-envT* [*locale-witness*]:
  monad-reader return-env bind-env ask-env
proof
  show *ask-env* (λ*r*. *ask-env* (*f r*)) = *ask-env* (λ*r*. *f r r*) **for** *f* :: '*r* ⇒ '*r* ⇒ ('*r*, '*m*) *envT*
    by(*rule envT.expand*)(*auto simp add*: *fun-eq-iff*)
  show *ask-env* (λ-. *x*) = *x* **for** *x* :: ('*r*, '*m*) *envT*
    by(*rule envT.expand*)(*auto simp add*: *fun-eq-iff*)
  show *bind-env* (*ask-env f*) *g* = *ask-env* (λ*r*. *bind-env* (*f r*) *g*) **for** *f* :: '*r* ⇒ ('*r*, '*m*) *envT* **and** *g*

by(*rule envT.expand*)(*auto simp add*: *fun-eq-iff*)
show *bind-env m* (λ*x*. *ask-env* (*f x*)) = *ask-env* (λ*r*. *bind-env m* (λ*x*. *f x r*)) **for** *m* :: (′*r*, ′*m*) *envT* **and** *f*
by(*rule envT.expand*)(*auto simp add*: *fun-eq-iff*)
qed

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*
by(*simp add*: *fail-env-def*)

lemma *monad-fail-envT* [*locale-witness*]:
assumes *monad-fail return bind fail*
shows *monad-fail return-env bind-env fail-env*
proof −
interpret *monad-fail return bind fail* **by**(*fact assms*)
have *bind-env fail-env f* = *fail-env* **for** *f* :: ′*a* ⇒ (′*r*, ′*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff fail-bind*)
then show *?thesis* **by** *unfold-locales*
qed

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*)
by(*simp add*: *catch-env-def*)

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 −
interpret *monad-catch return bind fail catch* **by** *fact*
show *?thesis*
proof
show *catch-env* (*return-env x*) *m* = *return-env x* **for** *x* **and** *m* :: (′*r*, ′*m*) *envT*

by(*rule envT.expand*)(*simp add*: *fun-eq-iff catch-return*)
show *catch-env fail-env m* = *m* **for** *m* :: ('*r*, '*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff catch-fail*)
show *catch-env m fail-env* = *m* **for** *m* :: ('*r*, '*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff catch-fail2*)
show *catch-env* (*catch-env m m*') *m*'' = *catch-env m* (*catch-env m*' *m*'')
for *m m*' *m*'' :: ('*r*, '*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff catch-assoc*)
qed
qed

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*)
by(*simp add*: *get-env-def*)

lemma *run-put-env* [*simp*]: *run-env* (*put-env s m*) *r* = *put s* (*run-env m r*)
by(*simp add*: *put-env-def*)

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 −
interpret *monad-state return bind get put* **by**(*fact assms*)
show *?thesis*
proof
show *put-env s* (*get-env f*) = *put-env s* (*f s*) **for** *s* :: '*s* **and** *f* :: '*s* ⇒ ('*r*, '*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff put-get*)
show *get-env* (λ*s*. *get-env* (*f s*)) = *get-env* (λ*s*. *f s s*) **for** *f* :: '*s* ⇒ '*s* ⇒ ('*r*, '*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff get-get*)
show *put-env s* (*put-env s*' *m*) = *put-env s*' *m* **for** *s s*' :: '*s* **and** *m* :: ('*r*, '*m*) *envT*

by(*rule envT.expand*)(*simp add*: *fun-eq-iff put-put*)
show *get-env* (*λs. put-env s m*) = *m* **for** *m* :: (*'r*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff get-put*)
show *get-env* (*λ-. m*) = *m* **for** *m* :: (*'r*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff get-const*)
show *bind-env* (*get-env f*) *g* = *get-env* (*λs. bind-env* (*f s*) *g*) **for** *f* :: *'s* ⇒ (*'r*, *'m*) *envT* **and** *g*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-get*)
show *bind-env* (*put-env s m*) *f* = *put-env s* (*bind-env m f*) **for** *s* **and** *m* :: (*'r*, *'m*) *envT* **and** *f*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-put*)
qed
qed

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* (*λr. sample p* (*λx. run-env* (*f x*) *r*))

lemma *run-sample-env* [*simp*]: *run-env* (*sample-env p f*) *r* = *sample p* (*λx. run-env* (*f x*) *r*)
by(*simp add*: *sample-env-def*)

lemma *monad-prob-envT* [*locale-witness*]:
assumes *monad-prob return bind sample*
shows *monad-prob return-env bind-env sample-env*
proof −
interpret *monad-prob return bind sample* **by**(*fact assms*)
note *sample-parametric*[*transfer-rule*]
show *?thesis* **including** *lifting-syntax*
proof
show *sample-env p* (*λ-. x*) = *x* **for** *p* :: *'p pmf* **and** *x* :: (*'r*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff sample-const*)
show *sample-env* (*return-pmf x*) *f* = *f x* **for** *f* :: *'p* ⇒ (*'r*, *'m*) *envT* **and** *x*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff sample-return-pmf*)
show *sample-env* (*bind-pmf p f*) *g* = *sample-env p* (*λx. sample-env* (*f x*) *g*) **for** *f* **and** *g* :: *'p* ⇒ (*'r*, *'m*) *envT* **and** *p*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff sample-bind-pmf*)
show *sample-env p* (*λx. sample-env q* (*f x*)) = *sample-env q* (*λy. sample-env p* (*λx. f x y*))
for *p q* :: *'p pmf* **and** *f* :: *'p* ⇒ *'p* ⇒ (*'r*, *'m*) *envT*
by(*rule envT.expand*)(*auto simp add*: *fun-eq-iff intro*: *sample-commute*)
show *bind-env* (*sample-env p f*) *g* = *sample-env p* (*λx. bind-env* (*f x*) *g*)
for *p* **and** *f* :: *'p* ⇒ (*'r*, *'m*) *envT* **and** *g*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-sample1*)

show *bind-env m* (*λy. sample-env p* (*f y*)) = *sample-env p* (*λx. bind-env m* (*λy. f y x*))
for *m p* **and** *f* :: *'a* ⇒ *'p* ⇒ (*'r*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-sample2*)
show (*rel-pmf R* ===> (*R* ===> (=)) ===> (=)) *sample-env sample-env*
if [*transfer-rule*]: *bi-unique R* **for** *R* **unfolding** *sample-env-def* **by** *transfer-prover*
qed
qed

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 −
interpret *monad-state-prob return bind get put sample* **by** *fact*
show *?thesis*
proof
show *sample-env p* (*λx. get-env* (*f x*)) = *get-env* (*λs. sample-env p* (*λx. f x s*))
for *p* **and** *f* :: *'p* ⇒ *'s* ⇒ (*'r*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff sample-get*)
qed
qed

end

3.5.5 Binary Non-determinism

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

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

lemma *run-alt-env* [*simp*]: *run-env* (*alt-env m1 m2*) *r* = *alt* (*run-env m1 r*) (*run-env m2 r*)
by(*simp add*: *alt-env-def*)

lemma *monad-alt-envT* [*locale-witness*]:
assumes *monad-alt return bind alt*
shows *monad-alt return-env bind-env alt-env*
proof −
interpret *monad-alt return bind alt* **by** *fact*
show *?thesis*
proof
show *alt-env* (*alt-env m1 m2*) *m3* = *alt-env m1* (*alt-env m2 m3*) **for** *m1 m2 m3* :: (*'r*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff alt-assoc*)
show *bind-env* (*alt-env m m'*) *f* = *alt-env* (*bind-env m f*) (*bind-env m' f*) **for**

m m′ :: (*′r*, *′m*) *envT* **and** *f*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-alt1*)
qed
qed

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 −
interpret *monad-fail-alt return bind fail alt* **by** *fact*
show *?thesis*
proof
show *alt-env* (*fail-env fail*) *m* = *m* **for** *m* :: (*′r*, *′m*) *envT*
by(*rule envT.expand*)(*simp add*: *alt-fail1 fun-eq-iff*)
show *alt-env m* (*fail-env fail*) = *m* **for** *m* :: (*′r*, *′m*) *envT*
by(*rule envT.expand*)(*simp add*: *alt-fail2 fun-eq-iff*)
qed
qed

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 −
interpret *monad-state-alt return bind get put alt* **by** *fact*
show *?thesis*
proof
show *alt-env* (*get-env f*) (*get-env g*) = *get-env* (*λx. alt-env* (*f x*) (*g x*))
for *f g* :: *′s* ⇒ (*′b*, *′m*) *envT* **by**(*rule envT.expand*)(*simp add*: *fun-eq-iff alt-get*)
show *alt-env* (*put-env s m*) (*put-env s m′*) = *put-env s* (*alt-env m m′*)
for *s* **and** *m m′* :: (*′b*, *′m*) *envT* **by**(*rule envT.expand*)(*simp add*: *fun-eq-iff alt-put*)
qed
qed

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* (*λr. altc C* (*λc. run-env* (*f c*) *r*))

lemma *run-altc-env* [*simp*]: *run-env* (*altc-env C f*) *r* = *altc C* (*λc. run-env* (*f c*) *r*)
by(*simp add*: *altc-env-def*)

lemma *monad-altc-envT* [*locale-witness*]:
assumes *monad-altc return bind altc*
shows *monad-altc return-env bind-env altc-env*
proof −
interpret *monad-altc return bind altc* **by** *fact*
note *altc-parametric*[*transfer-rule*]
show *?thesis* **including** *lifting-syntax*
proof
show *bind-env* (*altc-env C g*) *f* = *altc-env C* (λ*c. bind-env* (*g c*) *f*) **for** *C g* **and** *f* :: ′*a* ⇒ (′*b*, ′*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-altc1*)
show *altc-env* (*csingle x*) *f* = *f x* **for** *x* **and** *f* :: ′*c* ⇒ (′*b*, ′*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff altc-single*)
show *altc-env* (*cUNION C f*) *g* = *altc-env C* (λ*x. altc-env* (*f x*) *g*) **for** *C f* **and** *g* :: ′*c* ⇒ (′*b*, ′*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff altc-cUNION*)
show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) *altc-env altc-env* **if** [*transfer-rule*]: *bi-unique R* **for** *R*
unfolding *altc-env-def* **by** *transfer-prover*
qed
qed

lemma *monad-altc3-envT* [*locale-witness*]:
assumes *monad-altc3 return bind altc*
shows *monad-altc3 return-env bind-env altc-env*
proof −
interpret *monad-altc3 return bind altc* **by** *fact*
show *?thesis* **..**
qed

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 −
interpret *monad-state-altc return bind get put altc* **by** *fact*
show *?thesis*
proof
show *altc-env C* (λ*c. get-env* (*f c*)) = *get-env* (λ*s. altc-env C* (λ*c. f c s*))
for *C* **and** *f* :: ′*c* ⇒ ′*s* ⇒ (′*b*, ′*m*) *envT* **by**(*rule envT.expand*)(*simp add*: *fun-eq-iff altc-get*)
show *altc-env C* (λ*c. put-env s* (*f c*)) = *put-env s* (*altc-env C f*)
for *C s* **and** *f* :: ′*c* ⇒ (′*b*, ′*m*) *envT* **by**(*rule envT.expand*)(*simp add*: *fun-eq-iff altc-put*)
qed
qed

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

lemma *run-pause-env* [*simp*]:
  run-env (*pause-env out c*) *r* = *pause out* (*λi. run-env* (*c i*) *r*)
by(*simp add*: *pause-env-def*)

lemma *monad-resumption-envT* [*locale-witness*]:
  assumes *monad-resumption return bind pause*
  shows *monad-resumption return-env bind-env pause-env*
proof −
  interpret *monad-resumption return bind pause* **by** *fact*
  show *?thesis*
  proof
    show *bind-env* (*pause-env out c*) *f* = *pause-env out* (*λi. bind-env* (*c i*) *f*) **for** *out f* **and** *c* :: *'i* ⇒ (*'r*, *'m*) *envT*
      by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-pause*)
  qed
qed

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*)
by(*simp add*: *tell-env-def*)

lemma *monad-writer-envT* [*locale-witness*]:
  assumes *monad-writer return bind tell*
  shows *monad-writer return-env bind-env tell-env*
proof −
  interpret *monad-writer return bind tell* **by** *fact*
  show *?thesis*
  proof

show *bind-env* (*tell-env w m*) *f* = *tell-env w* (*bind-env m f*) **for** *w* **and** *m* :: ('*r*, '*m*) *envT* **and** *f*
by(*rule envT.expand*)(*simp add*: *bind-tell fun-eq-iff*)
qed
qed

end

3.5.9 Commutativity

lemma *monad-commute-envT* [*locale-witness*]:
assumes *monad-commute return bind*
shows *monad-commute return-env bind-env*
proof −
interpret *monad-commute return bind* **by** *fact*
show *?thesis*
proof
show *bind-env m* (λx. *bind-env m'* (*f x*)) = *bind-env m'* (λy. *bind-env m* (λx. *f x y*))
for *f* **and** *m m'* :: ('*r*, '*m*) *envT*
by(*rule envT.expand*)(*auto simp add*: *fun-eq-iff intro*: *bind-commute*)
qed
qed

3.5.10 Discardability

lemma *monad-discard-envT* [*locale-witness*]:
assumes *monad-discard return bind*
shows *monad-discard return-env bind-env*
proof −
interpret *monad-discard return bind* **by** *fact*
show *?thesis*
proof
show *bind-env m* (λ-. *m'*) = *m'* **for** *m m'* :: ('*r*, '*m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-const*)
qed
qed

3.5.11 Duplicability

lemma *monad-duplicate-envT* [*locale-witness*]:
assumes *monad-duplicate return bind*
shows *monad-duplicate return-env bind-env*
proof −
interpret *monad-duplicate return bind* **by** *fact*
show *?thesis*
proof
show *bind-env m* (λx. *bind-env m* (*f x*)) = *bind-env m* (λx. *f x x*) **for** *m* :: ('*b*, '*m*) *envT* **and** *f*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff bind-duplicate*)

qed
qed

end

3.5.12 Parametricity

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

lemma *return-env-parametric* [*transfer-rule*]:
((*A* ===> *M*) ===> *A* ===> *rel-envT R M*) *return-env return-env*
unfolding *return-env-def* **by** *transfer-prover*

lemma *bind-env-parametric* [*transfer-rule*]:
((*M* ===> (*A* ===> *M*) ===> *M*) ===> *rel-envT R M* ===> (*A* ===> *rel-envT R M*) ===> *rel-envT R M*)
bind-env bind-env
unfolding *bind-env-def* **by** *transfer-prover*

lemma *ask-env-parametric* [*transfer-rule*]: ((*R* ===> *rel-envT R M*) ===> *rel-envT R M*) *ask-env ask-env*
unfolding *ask-env-def* **by** *transfer-prover*

lemma *fail-env-parametric* [*transfer-rule*]: (*M* ===> *rel-envT R M*) *fail-env fail-env*
unfolding *fail-env-def* **by** *transfer-prover*

lemma *catch-env-parametric* [*transfer-rule*]:
((*M* ===> *M* ===> *M*) ===> *rel-envT R M* ===> *rel-envT R M* ===> *rel-envT R M*) *catch-env catch-env*
unfolding *catch-env-def* **by** *transfer-prover*

lemma *get-env-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *M*) ===> (*S* ===> *rel-envT R M*) ===> *rel-envT R M*) *get-env get-env*
unfolding *get-env-def* **by** *transfer-prover*

lemma *put-env-parametric* [*transfer-rule*]:
((*S* ===> *M* ===> *M*) ===> *S* ===> *rel-envT R M* ===> *rel-envT R M*) *put-env put-env*
unfolding *put-env-def* **by** *transfer-prover*

lemma *sample-env-parametric* [*transfer-rule*]:
((*rel-pmf P* ===> (*P* ===> *M*) ===> *M*) ===> *rel-pmf P* ===> (*P* ===> *rel-envT R M*) ===> *rel-envT R M*)
sample-env sample-env
unfolding *sample-env-def* **by** *transfer-prover*

lemma *alt-env-parametric* [*transfer-rule*]:
((*M* ===> *M* ===> *M*) ===> *rel-envT R M* ===> *rel-envT R M* ===>

rel-envT R M) *alt-env alt-env*
unfolding *alt-env-def* **by** *transfer-prover*

lemma *altc-env-parametric* [*transfer-rule*]:
((*rel-cset C* ===> (*C* ===> *M*) ===> *M*) ===> *rel-cset C* ===> (*C* ===> *rel-envT R M*) ===> *rel-envT R M*)
altc-env altc-env
unfolding *altc-env-def* **by** *transfer-prover*

lemma *pause-env-parametric* [*transfer-rule*]:
((*Out* ===> (*In* ===> *M*) ===> *M*) ===> *Out* ===> (*In* ===> *rel-envT R M*) ===> *rel-envT R M*)
pause-env pause-env
unfolding *pause-env-def* **by** *transfer-prover*

lemma *tell-env-parametric* [*transfer-rule*]:
((*W* ===> *M* ===> *M*) ===> *W* ===> *rel-envT R M* ===> *rel-envT R M*) *tell-env tell-env*
unfolding *tell-env-def* **by** *transfer-prover*

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* ≡ *λA 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* ≡ *λf*. ⋃ (*f* ‘ *rcset C*)

lemma *monad-set* [*locale-witness*]: *monad return-set bind-set*
by *unfold-locales auto*

lemma *monad-fail-set* [*locale-witness*]: *monad-fail return-set bind-set fail-set*
by *unfold-locales auto*

lemma *monad-lift-set* [*simp*]: *monad-base.lift return-set bind-set* = *image*
by(*auto simp add*: *monad-base.lift-def o-def fun-eq-iff*)

lemma *monad-alt-set* [*locale-witness*]: *monad-alt return-set bind-set alt-set*
by *unfold-locales auto*

lemma *monad-altc-set* [*locale-witness*]: *monad-altc return-set bind-set altc-set*
including *cset.lifting* **and** *lifting-syntax*
proof
show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) (*λC f*. ⋃ (*f* ‘ *rcset C*)) (*λC f*. ⋃ (*f* ‘ *rcset C*)) **for** *R*

by *transfer-prover*
qed(*transfer*; *auto*; *fail*)+

lemma *monad-altc3-set* [*locale-witness*]:
monad-altc3 return-set bind-set (*altc-set* :: (′*c*, ′*a set*) *altc*)
if [*locale-witness*]: *three TYPE*(′*c*)
..

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* = {}
by(*cases x*) *simp*

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

lemma *rel-nondetT*′*-phantom*: *rel-nondetT*′ *A* = *rel-nondetT*′ *top*
by(*auto 4 4 intro*: *nondetT.rel-mono antisym nondetT.rel-mono-strong*)

lemma *map-nondetT*′*-phantom*: *map-nondetT*′ *f* = *map-nondetT*′ *undefined*
by(*auto 4 4 intro*: *nondetT.map-cong*)

definition *map-nondetT* :: (′*m* ⇒ ′*m*′) ⇒ (′*a*, ′*m*) *nondetT* ⇒ (′*b*, ′*m*′) *nondetT*
where *map-nondetT* = *map-nondetT*′ *undefined*

definition *rel-nondetT* :: (′*m* ⇒ ′*m*′ ⇒ *bool*) ⇒ (′*a*, ′*m*) *nondetT* ⇒ (′*b*, ′*m*′) *nondetT* ⇒ *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*
using *assms* **by**(*cases m*; *cases m*′; *simp add*: *rel-nondetT-def*)

lemma *rel-nondetT-simps* [*simp*]: *rel-nondetT M* (*NondetT m*) (*NondetT m*′) ⟷ *M m m*′
by(*simp add*: *rel-nondetT-def*)

lemma *rel-nondetT-unfold*:
⋀*m m*′. *rel-nondetT M* (*NondetT m*) *m*′ ⟷ (∃ *m*″. *m*′ = *NondetT m*″ ∧ *M m m*″)
⋀*m m*′. *rel-nondetT M m* (*NondetT m*′) ⟷ (∃ *m*″. *m* = *NondetT m*″ ∧ *M m*″ *m*′)

subgoal for *m m′* **by**(*cases m′*; *simp*)
subgoal for *m m′* **by**(*cases m*; *simp*)
done

lemma *rel-nondetT-expand*: *M* (*run-nondet m*) (*run-nondet m′*) ⟹ *rel-nondetT M m m′*
by(*cases m*; *cases m′*; *simp*)

lemma *rel-nondetT-eq* [*relator-eq*]: *rel-nondetT* (=) = (=)
by(*auto simp add*: *fun-eq-iff rel-nondetT-def intro*: *nondetT.rel-refl-strong elim*: *nondetT.rel-cases*)

lemma *rel-nondetT-mono* [*relator-mono*]: *rel-nondetT A* ≤ *rel-nondetT B* **if** *A* ≤ *B*
by(*simp add*: *rel-nondetT-def nondetT.rel-mono that*)

lemma *rel-nondetT-distr* [*relator-distr*]: *rel-nondetT A OO rel-nondetT B* = *rel-nondetT* (*A OO B*)
by(*simp add*: *rel-nondetT-def nondetT.rel-compp*[*symmetric*])

lemma *rel-nondetT-Grp*: *rel-nondetT* (*BNF-Def.Grp A f*) = *BNF-Def.Grp* {*x. set-nondetT x* ⊆ *A*} (*map-nondetT f*)
by(*simp add*: *rel-nondetT-def rel-nondetT′-phantom*[*of BNF-Def.Grp UNIV undefined*, *symmetric*] *nondetT.rel-Grp map-nondetT-def*)

lemma *NondetT-parametric* [*transfer-rule*]: (*M* ===> *rel-nondetT M*) *NondetT NondetT*
by(*simp add*: *rel-fun-def rel-nondetT-def*)

lemma *run-nondet-parametric* [*transfer-rule*]: (*rel-nondetT M* ===> *M*) *run-nondet run-nondet*
by(*auto simp add*: *rel-fun-def rel-nondetT-def elim*: *nondetT.rel-cases*)

lemma *case-nondetT-parametric* [*transfer-rule*]:
((*M* ===> *X*) ===> *rel-nondetT M* ===> *X*) *case-nondetT case-nondetT*
by(*auto simp add*: *rel-fun-def rel-nondetT-def split*: *nondetT.split*)

lemma *rec-nondetT-parametric* [*transfer-rule*]:
((*M* ===> *X*) ===> *rel-nondetT M* ===> *X*) *rec-nondetT rec-nondetT*
by(*auto simp add*: *rel-fun-def elim*: *rel-nondetTE*)

end

3.7.1 Generic implementation

type-synonym (′*a*, ′*m*, ′*s*) *merge* = ′*s* ⇒ (′*a* ⇒ ′*m*) ⇒ ′*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* ⇒ *'s*
and *union* :: *'s* ⇒ *'s* ⇒ *'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*) (λ*A*. *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*) (λ*A*. *bind* (*run-nondet m2*) (λ*B*. *return* (*A* ∪ *B*))))

definition *get-nondet* :: (*'state*, *'m*) *get* ⇒ (*'state*, (*'a*, *'m*) *nondetT*) *get*
where *get-nondet get f* = *NondetT* (*get* (λ*s*. *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* (λ*r*. *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*)
by(*simp add*: *return-nondet-def*)

lemma *run-bind-nondet* [*simp*]:
run-nondet (*bind-nondet m f*) = *bind* (*run-nondet m*) (λ*A*. *merge A* (*run-nondet* ∘ *f*))
by(*simp add*: *bind-nondet-def*)

lemma *run-fail-nondet* [*simp*]: *run-nondet fail-nondet* = *return empty*
by(*simp add*: *fail-nondet-def*)

lemma *run-alt-nondet* [*simp*]:
run-nondet (*alt-nondet m1 m2*) = *bind* (*run-nondet m1*) (λ*A*. *bind* (*run-nondet*

m2) (*λB*. *return* (*A* ∪ *B*)))
by(*simp add*: *alt-nondet-def*)

lemma *run-get-nondet* [*simp*]: *run-nondet* (*get-nondet get f*) = *get* (*λs*. *run-nondet* (*f s*)) **for** *get*
by(*simp add*: *get-nondet-def*)

lemma *run-put-nondet* [*simp*]: *run-nondet* (*put-nondet put s m*) = *put s* (*run-nondet* *m*) **for** *put*
by(*simp add*: *put-nondet-def*)

lemma *run-ask-nondet* [*simp*]: *run-nondet* (*ask-nondet ask f*) = *ask* (*λr*. *run-nondet* (*f r*)) **for** *ask*
by(*simp add*: *ask-nondet-def*)

end

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

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*:
⋀*y f g*. *bind* (*merge y f*) (*λA*. *merge A g*) = *merge y* (*λx*. *bind* (*f x*) (*λA*. *merge* *A g*))
and *merge-empty*: ⋀*f*. *merge empty f* = *return empty*
and *merge-single*: ⋀*x f*. *merge* (*single x*) *f* = *f x*
and *merge-single2*: ⋀*A*. *merge A* (*λx*. *return* (*single x*)) = *return A*
and *merge-union*: ⋀*A B f*. *merge* (*A* ∪ *B*) *f* = *bind* (*merge A f*) (*λA'*. *bind* (*merge B f*) (*λB'*. *return* (*A'* ∪ *B'*)))
and *union-assoc*: ⋀*A B C*. (*A* ∪ *B*) ∪ *C* = *A* ∪ (*B* ∪ *C*)

and *empty-union*: $\bigwedge A.$ *empty* ∪ $A = A$
and *union-empty*: $\bigwedge A.$ A ∪ *empty* $= A$
begin

lemma *monad-nondetT* [*locale-witness*]: *monad return-nondet bind-nondet*
proof
show *bind-nondet* (*bind-nondet x f*) *g* = *bind-nondet x* (λ*y*. *bind-nondet* (*f y*) *g*) **for** *x f g*
by(*rule nondetT.expand*)(*simp add*: *bind-assoc bind-merge-merge o-def*)
show *bind-nondet* (*return-nondet x*) *f* = *f x* **for** *x f*
by(*rule nondetT.expand*)(*simp add*: *return-bind merge-single*)
show *bind-nondet x return-nondet* = *x* **for** *x*
by(*rule nondetT.expand*)(*simp add*: *bind-return o-def merge-single2*)
qed

lemma *monad-fail-nondetT* [*locale-witness*]: *monad-fail return-nondet bind-nondet fail-nondet*
proof
show *bind-nondet fail-nondet f* = *fail-nondet* **for** *f*
by(*rule nondetT.expand*)(*simp add*: *return-bind merge-empty*)
qed

lemma *monad-alt-nondetT* [*locale-witness*]: *monad-alt return-nondet bind-nondet alt-nondet*
proof
show *alt-nondet* (*alt-nondet m1 m2*) *m3* = *alt-nondet m1* (*alt-nondet m2 m3*) **for** *m1 m2 m3*
by(*rule nondetT.expand*)(*simp add*: *bind-assoc return-bind union-assoc*)
show *bind-nondet* (*alt-nondet m m′*) *f* = *alt-nondet* (*bind-nondet m f*) (*bind-nondet m′ f*) **for** *m m′ f*
apply(*rule nondetT.expand*)
apply(*simp add*: *bind-assoc return-bind*)
apply(*subst* (*2*) *bind-commute*)
apply(*simp add*: *merge-union*)
done
qed

lemma *monad-fail-alt-nondetT* [*locale-witness*]:
monad-fail-alt return-nondet bind-nondet fail-nondet alt-nondet
proof
show *alt-nondet fail-nondet m* = *m* **for** *m*
by(*rule nondetT.expand*)(*simp add*: *return-bind bind-return empty-union*)
show *alt-nondet m fail-nondet* = *m* **for** *m*
by(*rule nondetT.expand*)(*simp add*: *return-bind bind-return union-empty*)
qed

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 −
interpret *monad-state return bind get put* **by** *fact*
show *?thesis*
proof
show *put-nondet put s* (*get-nondet get f*) = *put-nondet put s* (*f s*) **for** *s f*
by(*rule nondetT.expand*)(*simp add*: *put-get*)
show *get-nondet get* (λ*s*. *get-nondet get* (*f s*)) = *get-nondet get* (λ*s*. *f s s*) **for** *f*
by(*rule nondetT.expand*)(*simp add*: *get-get*)
show *put-nondet put s* (*put-nondet put s′ m*) = *put-nondet put s′ m* **for** *s s′ m*
by(*rule nondetT.expand*)(*simp add*: *put-put*)
show *get-nondet get* (λ*s*. *put-nondet put s m*) = *m* **for** *m*
by(*rule nondetT.expand*)(*simp add*: *get-put*)
show *get-nondet get* (λ-. *m*) = *m* **for** *m*
by(*rule nondetT.expand*)(*simp add*: *get-const*)
show *bind-nondet* (*get-nondet get f*) *g* = *get-nondet get* (λ*s*. *bind-nondet* (*f s*) *g*) **for** *f g*
by(*rule nondetT.expand*)(*simp add*: *bind-get*)
show *bind-nondet* (*put-nondet put s m*) *f* = *put-nondet put s* (*bind-nondet m f*) **for** *s m f*
by(*rule nondetT.expand*)(*simp add*: *bind-put*)
qed
qed

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 −
interpret *monad-state return bind get put* **by** *fact*
show *?thesis*
proof
show *alt-nondet* (*get-nondet get f*) (*get-nondet get g*) = *get-nondet get* (λ*x*. *alt-nondet* (*f x*) (*g x*))
for *f g*
apply(*rule nondetT.expand*; *simp*)
apply(*subst bind-get*)
apply(*subst* (*1 2*) *bind-commute*)
apply(*simp add*: *bind-get get-get*)
done
show *alt-nondet* (*put-nondet put s m*) (*put-nondet put s m′*) = *put-nondet put s* (*alt-nondet m m′*)
for *s m m′*
apply(*rule nondetT.expand*; *simp*)
apply(*subst bind-put*)
apply(*subst* (*1 2*) *bind-commute*)

apply(*simp add*: *bind-put put-put*)
done
qed
qed

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* ⇒ '*s*
and *union* :: '*s* ⇒ '*s* ⇒ '*s* (**infixl** ‹∪› *65*)
+
assumes *monad-reader*: *monad-reader return bind ask*
assumes *merge-ask*:
⋀*A* (*f* :: '*a* ⇒ '*r* ⇒ ('*a*, '*m*) *nondetT*). *merge A* (λ*x*. *ask* (λ*r*. *run-nondet* (*f x r*))) =
ask (λ*r*. *merge A* (λ*x*. *run-nondet* (*f x r*)))
begin

interpretation *monad-reader return bind ask* **by**(*fact monad-reader*)

lemma *monad-reader-nondetT*: *monad-reader return-nondet bind-nondet* (*ask-nondet ask*)
proof
show *ask-nondet ask* (λ*r*. *ask-nondet ask* (*f r*)) = *ask-nondet ask* (λ*r*. *f r r*) **for** *f*
by(*rule nondetT.expand*)(*simp add*: *ask-ask*)
show *ask-nondet ask* (λ-. *m*) = *m* **for** *m*
by(*rule nondetT.expand*)(*simp add*: *ask-const*)
show *bind-nondet* (*ask-nondet ask f*) *g* = *ask-nondet ask* (λ*r*. *bind-nondet* (*f r*) *g*) **for** *f g*
by(*rule nondetT.expand*)(*simp add*: *bind-ask*)
show *bind-nondet m* (λ*x*. *ask-nondet ask* (*f x*)) = *ask-nondet ask* (λ*r*. *bind-nondet m* (λ*x*. *f x r*)) **for** *f m*
by(*rule nondetT.expand*)(*simp add*: *bind-ask2*[*symmetric*] *o-def merge-ask*)
qed

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
unfolding *nondetM-base.return-nondet-def* **by** *transfer-prover*

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
unfolding *nondetM-base.bind-nondet-def* **by** *transfer-prover*

lemma *fail-nondet-parametric* [*transfer-rule*]:
((*S* ===> *M*) ===> *S* ===> *rel-nondetT M*) *nondetM-base.fail-nondet nondetM-base.fail-nondet*
unfolding *nondetM-base.fail-nondet-def* **by** *transfer-prover*

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
unfolding *nondetM-base.alt-nondet-def* **by** *transfer-prover*

lemma *get-nondet-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *M*) ===> (*S* ===> *rel-nondetT M*) ===> *rel-nondetT M*)
nondetM-base.get-nondet nondetM-base.get-nondet
unfolding *nondetM-base.get-nondet-def* **by** *transfer-prover*

lemma *put-nondet-parametric* [*transfer-rule*]:
((*S* ===> *M* ===> *M*) ===> *S* ===> *rel-nondetT M* ===> *rel-nondetT M*)
nondetM-base.put-nondet nondetM-base.put-nondet
unfolding *nondetM-base.put-nondet-def* **by** *transfer-prover*

lemma *ask-nondet-parametric* [*transfer-rule*]:
(((*R* ===> *M*) ===> *M*) ===> (*R* ===> *rel-nondetT M*) ===> *rel-nondetT M*)
nondetM-base.ask-nondet nondetM-base.ask-nondet
unfolding *nondetM-base.ask-nondet-def* **by** *transfer-prover*

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* ≡ *bind m1* (λ*A*. *bind m2* (λ*B*. *return* (*A* @ *B*)))
    and *lUnionM ms* ≡ *foldr lunionM ms* (*return* [])
begin

definition *lmerge* :: *'a list* ⇒ (*'a* ⇒ *'m*) ⇒ *'m* **where**
  lmerge A f = *lUnionM* (*map f A*)

context
  assumes *monad-commute return bind*
begin

interpretation *monad-commute return bind* **by** *fact*
interpretation *nondetM-base return bind lmerge* [] λ*x*. [*x*] (@) **.**

lemma *lUnionM-empty* [*simp*]: *lUnionM* [] = *return* [] **by**(*simp add*: *lUnionM-def*)
lemma *lUnionM-Cons* [*simp*]: *lUnionM* (*x* # *M*) = *lunionM x* (*lUnionM M*) **for** *x M*
  by(*simp add*: *lUnionM-def*)
lemma *lunionM-return-empty1* [*simp*]: *lunionM* (*return* []) *x* = *x* **for** *x*
  by(*simp add*: *lunionM-def return-bind bind-return*)
lemma *lunionM-return-empty2* [*simp*]: *lunionM x* (*return* []) = *x* **for** *x*
  by(*simp add*: *lunionM-def return-bind bind-return*)
lemma *lunionM-return-return* [*simp*]: *lunionM* (*return A*) (*return B*) = *return* (*A* @ *B*) **for** *A B*
  by(*simp add*: *lunionM-def return-bind*)
lemma *lunionM-assoc*: *lunionM* (*lunionM x y*) *z* = *lunionM x* (*lunionM y z*) **for** *x y z*
  by(*simp add*: *lunionM-def bind-assoc return-bind*)
lemma *lunionM-lUnionM1*: *lunionM* (*lUnionM A*) *x* = *foldr lunionM A x* **for** *A x*
  by(*induction A arbitrary*: *x*)(*simp-all add*: *lunionM-assoc*)
lemma *lUnionM-append* [*simp*]: *lUnionM* (*A* @ *B*) = *lunionM* (*lUnionM A*) (*lUnionM B*) **for** *A B*
  by(*subst lunionM-lUnionM1*)(*simp add*: *lUnionM-def*)
lemma *lUnionM-return* [*simp*]: *lUnionM* (*map* (λ*x*. *return* [*x*]) *A*) = *return A* **for** *A*
  by(*induction A*) *simp-all*
lemma *bind-lunionM*: *bind* (*lunionM m m'*) *f* = *lunionM* (*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*

apply(*simp add*: *bind-assoc return-bind lunionM-def that*)
apply(*subst* (*2*) *bind-commute*)
apply *simp*
done

lemma *list-nondetM*: *nondetM return bind lmerge* [] (λx. [*x*]) (@)
proof
show *bind* (*lmerge y f*) (λA. *lmerge A g*) = *lmerge y* (λx. *bind* (*f x*) (λA. *lmerge A g*)) **for** *y f g*
apply(*induction y*)
apply(*simp-all add*: *lmerge-def return-bind*)
apply(*subst bind-lunionM*; *simp add*: *lunionM-def o-def*)
done
show *lmerge* [] *f* = *return* [] **for** *f* **by**(*simp add*: *lmerge-def*)
show *lmerge* [*x*] *f* = *f x* **for** *x f* **by**(*simp add*: *lmerge-def*)
show *lmerge A* (λx. *return* [*x*]) = *return A* **for** *A* **by**(*simp add*: *lmerge-def*)
show *lmerge* (*A* @ *B*) *f* = *bind* (*lmerge A f*) ($\lambda A'$. *bind* (*lmerge B f*) ($\lambda B'$. *return* (*A'* @ *B'*)))
for *f A B* **by**(*simp add*: *lmerge-def lunionM-def*)
qed *simp-all*

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
interpret *monad-reader return bind ask* **by** *fact*
show *lmerge A* (λx. *ask* (λr. *run-nondet* (*f x r*))) = *ask* (λr. *lmerge A* (λx. *run-nondet* (*f x r*)))
for *A* **and** *f* :: *'a* ⇒ *'b* ⇒ (*'a*, *'m*) *nondetT* **unfolding** *lmerge-def*
by(*induction A*)(*simp-all add*: *ask-const lunionM-def bind-ask bind-ask2 ask-ask*)
qed

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
unfolding *lmerge-def* **by** *transfer-prover*

3.7.4 Implementation using multisets

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

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

context
  assumes *monad-commute return bind*
begin

interpretation *monad-commute return bind* **by** *fact*
interpretation *nondetM-base return bind mmerge* {#} λ*x*. {#*x*#} (+) **.**

lemma *munionM-comp-fun-commute*: *comp-fun-commute munionM*
  apply(*unfold-locales*)
  apply(*simp add*: *fun-eq-iff bind-assoc return-bind munionM-def*)
  apply(*subst bind-commute*)
  apply(*simp add*: *union-ac*)
  done

interpretation *comp-fun-commute munionM* **by**(*rule munionM-comp-fun-commute*)

lemma *mUnionM-empty* [*simp*]: *mUnionM* {#} = *return* {#} **by**(*simp add*: *mUnionM-def*)
lemma *mUnionM-add-mset* [*simp*]: *mUnionM* (*add-mset x M*) = *munionM x* (*mUnionM M*) **for** *x M*
  by(*simp add*: *mUnionM-def*)
lemma *munionM-return-empty1* [*simp*]: *munionM* (*return* {#}) *x* = *x* **for** *x*
  by(*simp add*: *munionM-def return-bind bind-return*)
lemma *munionM-return-empty2* [*simp*]: *munionM x* (*return* {#}) = *x* **for** *x*
  by(*simp add*: *munionM-def return-bind bind-return*)
lemma *munionM-return-return* [*simp*]: *munionM* (*return A*) (*return B*) = *return* (*A* + *B*) **for** *A B*
  by(*simp add*: *munionM-def return-bind*)
lemma *munionM-assoc*: *munionM* (*munionM x y*) *z* = *munionM x* (*munionM y z*) **for** *x y z*
  by(*simp add*: *munionM-def bind-assoc return-bind add.assoc*)
lemma *munionM-commute*: *munionM x y* = *munionM y x* **for** *x y*
  unfolding *munionM-def* **by**(*subst bind-commute*)(*simp add*: *add.commute*)
lemma *munionM-mUnionM1*: *munionM* (*mUnionM A*) *x* = *fold-mset munionM x A* **for** *A x*
  by(*induction A arbitrary*: *x*)(*simp-all add*: *munionM-assoc*)
lemma *munionM-mUnionM2*: *munionM x* (*mUnionM A*) = *fold-mset munionM*

x A **for** *x A*
  by(*subst munionM-commute*)(*rule munionM-mUnionM1*)
lemma *mUnionM-add* [*simp*]: *mUnionM* (*A* + *B*) = *munionM* (*mUnionM A*) (*mUnionM B*) **for** *A B*
  by(*subst munionM-mUnionM2*)(*simp add*: *mUnionM-def*)
lemma *mUnionM-return* [*simp*]: *mUnionM* (*image-mset* (λ*x*. *return* {#*x*#}) *A*) = *return A* **for** *A*
  by(*induction A*) *simp-all*
lemma *bind-munionM*: *bind* (*munionM m m′*) *f* = *munionM* (*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*
  apply(*simp add*: *bind-assoc return-bind munionM-def that*)
  apply(*subst* (*2*) *bind-commute*)
  apply *simp*
  done

lemma *mset-nondetM*: *nondetM return bind mmerge* {#} (λ*x*. {#*x*#}) (+)
proof
  show *bind* (*mmerge y f*) (λ*A*. *mmerge A g*) = *mmerge y* (λ*x*. *bind* (*f x*) (λ*A*. *mmerge A g*)) **for** *y f g*
    apply(*induction y*)
    apply(*simp-all add*: *return-bind mmerge-def*)
    apply(*subst bind-munionM*; *simp add*: *munionM-def o-def*)
    done
  show *mmerge* {#} *f* = *return* {#} **for** *f* **by**(*simp add*: *mmerge-def*)
  show *mmerge* {#*x*#} *f* = *f x* **for** *x f* **by**(*simp add*: *mmerge-def*)
  show *mmerge A* (λ*x*. *return* {#*x*#}) = *return A* **for** *A* **by**(*simp add*: *mmerge-def*)
  show *mmerge* (*A* + *B*) *f* = *bind* (*mmerge A f*) (λ*A′*. *bind* (*mmerge B f*) (λ*B′*. *return* (*A′* + *B′*)))
    for *f A B* **by**(*simp add*: *mmerge-def munionM-def*)
qed *simp-all*

lemma *mset-nondetM-ask*:
  notes *mset-nondetM*[*locale-witness*]
  assumes [*locale-witness*]: *monad-reader return bind ask*
  shows *nondetM-ask return bind ask mmerge* {#} (λ*x*. {#*x*#}) (+)
proof
  interpret *monad-reader return bind ask* **by** *fact*
  show *mmerge A* (λ*x*. *ask* (λ*r*. *run-nondet* (*f x r*))) = *ask* (λ*r*. *mmerge A* (λ*x*. *run-nondet* (*f x r*)))
    for *A* **and** *f* :: ′*a* ⇒ ′*b* ⇒ (′*a*, ′*m*) *nondetT* **unfolding** *mmerge-def*
   by(*induction A*)(*simp-all add*: *ask-const munionM-def bind-ask bind-ask2 ask-ask*)
qed

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*)
unfolding *mmerge-def*
apply(*rule rel-funI*)+
apply(*drule* (*1*) *multiset.map-transfer*[*THEN rel-funD*, *THEN rel-funD*])
apply(*rule fold-mset-parametric*[*OF* - *munionM-comp-fun-commute*[*OF comm1*] *munionM-comp-fun-commute*[*OF comm2*], *THEN rel-funD*, *THEN rel-funD*, *rotated* −*1*], *assumption*)
subgoal premises [*transfer-rule*] **by** *transfer-prover*
subgoal premises by *transfer-prover*
done

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* **by** *fact*
interpretation *monad-duplicate return bind* **by** *fact*
interpretation *nondetM-base return bind fmerge* {||} λ*x*. {|*x*|} (|∪|) **.**

lemma *funionM-comp-fun-commute*: *comp-fun-commute funionM*
apply(*unfold-locales*)
apply(*simp add*: *fun-eq-iff bind-assoc return-bind funionM-def*)
apply(*subst bind-commute*)

apply(*simp add*: *funion-ac*)
done

interpretation *comp-fun-commute funionM* **by**(*rule funionM-comp-fun-commute*)

lemma *funionM-comp-fun-idem*: *comp-fun-idem funionM*
by(*unfold-locales*)(*simp add*: *fun-eq-iff funionM-def bind-assoc bind-duplicate return-bind*)

interpretation *comp-fun-idem funionM* **by**(*rule funionM-comp-fun-idem*)

lemma *fUnionM-empty* [*simp*]: *fUnionM* {||} = *return* {||} **by**(*simp add*: *fUnionM-def*)
lemma *fUnionM-finset* [*simp*]: *fUnionM* (*finsert x M*) = *funionM x* (*fUnionM M*) **for** *x M*
by(*simp add*: *fUnionM-def*)
lemma *funionM-return-empty1* [*simp*]: *funionM* (*return* {||}) *x* = *x* **for** *x*
by(*simp add*: *funionM-def return-bind bind-return*)
lemma *funionM-return-empty2* [*simp*]: *funionM x* (*return* {||}) = *x* **for** *x*
by(*simp add*: *funionM-def return-bind bind-return*)
lemma *funionM-return-return* [*simp*]: *funionM* (*return A*) (*return B*) = *return* (*A* |∪| *B*) **for** *A B*
by(*simp add*: *funionM-def return-bind*)
lemma *funionM-assoc*: *funionM* (*funionM x y*) *z* = *funionM x* (*funionM y z*) **for** *x y z*
by(*simp add*: *funionM-def bind-assoc return-bind funion-assoc*)
lemma *funionM-commute*: *funionM x y* = *funionM y x* **for** *x y*
unfolding *funionM-def* **by**(*subst bind-commute*)(*simp add*: *funion-commute*)
lemma *funionM-fUnionM1*: *funionM* (*fUnionM A*) *x* = *ffold funionM x A* **for** *A x*
by(*induction A arbitrary*: *x*)(*simp-all add*: *funionM-assoc*)
lemma *funionM-fUnionM2*: *funionM x* (*fUnionM A*) = *ffold funionM x A* **for** *x A*
by(*subst funionM-commute*)(*rule funionM-fUnionM1*)
lemma *fUnionM-funion* [*simp*]: *fUnionM* (*A* |∪| *B*) = *funionM* (*fUnionM A*) (*fUnionM B*) **for** *A B*
by(*subst funionM-fUnionM2*)(*simp add*: *fUnionM-def ffold-set-union*)
lemma *fUnionM-return* [*simp*]: *fUnionM* (*fimage* (*λx. return* {|*x*|}) *A*) = *return A* **for** *A*
by(*induction A*) *simp-all*
lemma *bind-funionM*: *bind* (*funionM m m′*) *f* = *funionM* (*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*
apply(*simp add*: *bind-assoc return-bind funionM-def that*)
apply(*subst* (*2*) *bind-commute*)
apply *simp*
done
lemma *fUnionM-return-fempty* [*simp*]: *fUnionM* (*fimage* (*λx. return* {||}) *A*) = *return* {||} **for** *A*

by(*induction A*) *simp-all*
lemma *funionM-bind*: *funionM* (*bind m f*) (*bind m g*) = *bind m* (*λx. funionM* (*f x*) (*g x*)) **for** *m f g*
unfolding *funionM-def bind-assoc* **by**(*subst bind-commute*)(*simp add*: *bind-duplicate*)
lemma *fUnionM-funionM*:
fUnionM ((*λy. funionM* (*f y*) (*g y*)) |'| *A*) = *funionM* (*fUnionM* (*f* |'| *A*)) (*fUnionM* (*g* |'| *A*)) **for** *f g A*
by(*induction A*)(*simp-all add*: *funionM-assoc funionM-commute fun-left-comm*)

lemma *fset-nondetM*: *nondetM return bind fmerge* {||} (*λx.* {|*x*|}) (|∪|)
proof
show *bind* (*fmerge y f*) (*λA. fmerge A g*) = *fmerge y* (*λx. bind* (*f x*) (*λA. fmerge A g*)) **for** *y f g*
apply(*induction y*)
apply(*simp-all add*: *return-bind fmerge-def*)
apply(*subst bind-funionM*; *simp add*: *funionM-def o-def fimage-funion*)
done

show *fmerge* {||} *f* = *return* {||} **for** *f* **by**(*simp add*: *fmerge-def*)
show *fmerge* {|*x*|} *f* = *f x* **for** *x f* **by**(*simp add*: *fmerge-def*)
show *fmerge A* (*λx. return* {|*x*|}) = *return A* **for** *A* **by**(*simp add*: *fmerge-def*)
show *fmerge* (*A* |∪| *B*) *f* = *bind* (*fmerge A f*) (*λA′. bind* (*fmerge B f*) (*λB′. return* (*A′* |∪| *B′*)))
for *f A B* **by**(*simp add*: *fmerge-def funionM-def fimage-funion*)
qed *auto*

lemma *fset-nondetM-ask*:
notes *fset-nondetM*[*locale-witness*]
assumes [*locale-witness*]: *monad-reader return bind ask*
shows *nondetM-ask return bind ask fmerge* {||} (*λx.* {|*x*|}) (|∪|)
proof
interpret *monad-reader return bind ask* **by** *fact*
show *fmerge A* (*λx. ask* (*λr. run-nondet* (*f x r*))) = *ask* (*λr. fmerge A* (*λx. run-nondet* (*f x r*)))
for *A* **and** *f* :: *′a* ⇒ *′b* ⇒ (*′a*, *′m*) *nondetT* **unfolding** *fmerge-def*
by(*induction A*)(*simp-all add*: *ask-const funionM-def bind-ask bind-ask2 ask-ask*)
qed

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* **by** *fact*

lemma *fmerge-bind*:
fmerge A (λx. *bind m′* ($\lambda A'$. *fmerge A′* (*f x*))) = *bind m′* ($\lambda A'$. *fmerge A* (λx. *fmerge A′* (*f x*)))
by(*induction A*)(*simp-all add*: *fmerge-def bind-const funionM-bind*)

lemma *fmerge-commute*: *fmerge A* (λx. *fmerge B* (*f x*)) = *fmerge B* (λy. *fmerge A* (λx. *f x y*))
by(*induction A*)(*simp-all add*: *fmerge-def fUnionM-funionM*)

lemma *monad-commute-nondetT-fset* [*locale-witness*]:
monad-commute return-nondet bind-nondet
proof
show *bind-nondet m* (λx. *bind-nondet m′* (*f x*)) = *bind-nondet m′* (λy. *bind-nondet m* (λx. *f x y*)) **for** *m m′ f*
apply(*rule nondetT.expand*)
apply(*simp add*: *o-def*)
apply(*subst fmerge-bind*)
apply(*subst bind-commute*)
apply(*subst fmerge-commute*)
apply(*subst fmerge-bind*[*symmetric*])
apply(*rule refl*)
done
qed

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*)
unfolding *fmerge-def*
apply(*rule rel-funI*)+
apply(*drule* (*1*) *fset.map-transfer*[*THEN rel-funD*, *THEN rel-funD*])
apply(*rule ffold-parametric*[*OF - funionM-comp-fun-idem*[*OF comm1*] *funionM-comp-fun-idem*[*OF comm2*], *THEN rel-funD*, *THEN rel-funD*, *rotated −1*], *assumption*)
subgoal premises [*transfer-rule*] **by** *transfer-prover*
subgoal premises by *transfer-prover*
done

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* ∘ *f*))

lemma *run-altc-nondet* [*simp*]: *run-nondet* (*altc-nondet A f*) = *mergec A* (*run-nondet* ∘ *f*)
  by(*simp add*: *altc-nondet-def*)

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*:
    ⋀*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*: ⋀*f*. *merge cempty f* = *return cempty*
  and *merge-single*: ⋀*x f*. *merge* (*csingle x*) *f* = *f x*
  and *merge-single2*: ⋀*A*. *merge A* (λ*x*. *return* (*csingle x*)) = *return A*
  and *merge-union*: ⋀*A B f*. *merge* (*cUn A B*) *f* = *bind* (*merge A f*) (λ*A'*. *bind* (*merge B f*) (λ*B'*. *return* (*cUn A' B'*)))
  and *bind-mergec-merge*:
    ⋀*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*: ⋀*x f*. *mergec* (*csingle x*) *f* = *f x*

and *mergec-UNION*: $\bigwedge C\, f\, g.$ *mergec* (*cUNION C f*) *g* = *mergec C* ($\lambda x.$ *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*
by(*unfold-locales*; (*rule bind-merge-merge merge-empty merge-single merge-single2 merge-union* | *simp add*: *cUn-assoc*)?)

sublocale *nondet*: *monad-altc return-nondet bind-nondet altc-nondet*
including *lifting-syntax*
proof
show *bind-nondet* (*altc-nondet C g*) *f* = *altc-nondet C* ($\lambda c.$ *bind-nondet* (*g c*) *f*) **for** *C g f*
by(*rule nondetT.expand*)(*simp add*: *bind-mergec-merge o-def*)
show *altc-nondet* (*csingle x*) *f* = *f x* **for** *x f*
by(*rule nondetT.expand*)(*simp add*: *mergec-single*)
show *altc-nondet* (*cUNION C f*) *g* = *altc-nondet C* ($\lambda x.$ *altc-nondet* (*f x*) *g*) **for** *C f g*
by(*rule nondetT.expand*)(*simp add*: *o-def mergec-UNION*)
show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) *altc-nondet altc-nondet*
if [*transfer-rule*]: *bi-unique R* **for** *R*
unfolding *altc-nondet-def* **by**(*transfer-prover*)
qed

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* **..**

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*)
by(*simp add*: *merge-id-def*)

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*
unfolding *merge-id-def* **by** *transfer-prover*

lemma *cset-nondetM-id* [*locale-witness*]: *cset-nondetM return-id bind-id merge-id merge-id*
including *lifting-syntax*
proof(*unfold-locales*)
show *bind-id* (*merge-id y f*) (*λA. merge-id A g*) = *merge-id y* (*λx. bind-id* (*f x*) (*λA. merge-id A g*))
for *y* **and** *f* :: *'c ⇒ 'd cset id* **and** *g* **by**(*rule id.expand*)(*simp add*: *o-def cUNION-assoc*)
then show *bind-id* (*merge-id y f*) (*λA. merge-id A g*) = *merge-id y* (*λx. bind-id* (*f x*) (*λA. merge-id A g*))
for *y* **and** *f* :: *'c ⇒ 'd cset id* **and** *g* **by** *this*
show *merge-id cempty f* = *return-id cempty* **for** *f* :: *'a ⇒ 'a cset id* **by**(*rule id.expand*) *simp*
show *merge-id* (*csingle x*) *f* = *f x* **for** *x* **and** *f* :: *'c ⇒ 'a cset id* **by**(*rule id.expand*) *simp*
then show *merge-id* (*csingle x*) *f* = *f x* **for** *x* **and** *f* :: *'c ⇒ 'a cset id* **by** *this*
show *merge-id A* (*λx. return-id* (*csingle x*)) = *return-id A* **for** *A* :: *'a cset*
by(*rule id.expand*)(*simp add*: *o-def*)
show *merge-id* (*cUn A B*) *f* = *bind-id* (*merge-id A f*) (*λA'. bind-id* (*merge-id B f*) (*λB'. return-id* (*cUn A' B'*)))
for *A B* **and** *f* :: *'a ⇒ 'a cset id* **by**(*rule id.expand*)(*simp add*: *cUNION-cUn*)

show *merge-id* (*cUNION C f*) *g* = *merge-id C* (*λx. merge-id* (*f x*) *g*)
for *C* **and** *f* :: *'b ⇒ 'b cset* **and** *g* :: *'b ⇒ 'a cset id*
by(*rule id.expand*)(*simp add*: *o-def cUNION-assoc*)
show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) *merge-id merge-id*
if *bi-unique R* **for** *R* :: *'b ⇒ 'b ⇒ bool*
unfolding *merge-id-def* **by** *transfer-prover*
qed

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* (*λr. merge A* (*λa. run-env* (*f a*) *r*)) **for** *merge*

lemma *run-merge-env* [*simp*]: *run-env* (*merge-env merge A f*) *r* = *merge A* (*λa. run-env* (*f a*) *r*) **for** *merge*
by(*simp add*: *merge-env-def*)

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
unfolding *merge-env-def* **by** *transfer-prover*

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 −
interpret *cset-nondetM return bind merge* **by** *fact*
show *?thesis* **including** *lifting-syntax*
proof
show *bind-env bind* (*merge-env merge y f*) (*λA. merge-env merge A g*) = *merge-env merge y* (*λx. bind-env bind* (*f x*) (*λA. merge-env merge A g*))
for *y* **and** *f* :: *'a* ⇒ (*'b*, *'m*) *envT* **and** *g*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff cUNION-assoc bind-merge-merge*)
show *merge-env merge cempty f* = *return-env return cempty* **for** *f* :: *'a* ⇒ (*'b*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff merge-empty*)
show *merge-env merge* (*csingle x*) *f* = *f x* **for** *f* :: *'a* ⇒ (*'b*, *'m*) *envT* **and** *x*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff merge-single*)
show *merge-env merge A* (*λx. return-env return* (*csingle x*)) = *return-env return A* **for** *A*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff merge-single2*)
show *merge-env merge* (*cUn A B*) *f* = *bind-env bind* (*merge-env merge A f*) (*λA'. bind-env bind* (*merge-env merge B f*) (*λB'. return-env return* (*cUn A' B'*)))
for *A B* **and** *f* :: *'a* ⇒ (*'b*, *'m*) *envT* **by**(*rule envT.expand*)(*simp add*: *fun-eq-iff merge-union*)
show *bind-env bind* (*merge-env mergec y f*) (*λA. merge-env merge A g*) = *merge-env mergec y* (*λx. bind-env bind* (*f x*) (*λA. merge-env merge A g*))
for *y* **and** *f* :: *'c* ⇒ (*'b*, *'m*) *envT* **and** *g*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff cUNION-assoc bind-mergec-merge*)
show *merge-env mergec* (*csingle x*) *f* = *f x* **for** *f* :: *'c* ⇒ (*'b*, *'m*) *envT* **and** *x*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff mergec-single*)
show *merge-env mergec* (*cUNION C f*) *g* = *merge-env mergec C* (*λx. merge-env mergec* (*f x*) *g*)
for *C f* **and** *g* :: *'c* ⇒ (*'b*, *'m*) *envT*
by(*rule envT.expand*)(*simp add*: *fun-eq-iff mergec-UNION*)
show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) (*merge-env mergec*) (*merge-env mergec*)
if [*transfer-rule*]: *bi-unique R* **for** *R*
unfolding *merge-env-def* **by** *transfer-prover*
qed
qed

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* :: $('s \Rightarrow 's' \Rightarrow bool) \Rightarrow ('m \Rightarrow 'm' \Rightarrow bool) \Rightarrow ('s, 'm)$ *stateT* $\Rightarrow ('s', 'm')$ *stateT* $\Rightarrow$ *bool*
where *rel-stateT S M m m'* $\longleftrightarrow$ (*S* ===> *M*) (*run-state m*) (*run-state m'*)

lemma *rel-stateT-eq* [*relator-eq*]: *rel-stateT* (=) (=) = (=)
by(*auto simp add*: *rel-stateT-def fun-eq-iff rel-fun-eq intro*: *stateT.expand*)

lemma *rel-stateT-mono* [*relator-mono*]: $[\![S' \leq S; M \leq M']\!] \Longrightarrow$ *rel-stateT S M* $\leq$ *rel-stateT S' M'*
by(*rule predicate2I*)(*simp add*: *rel-stateT-def fun-mono*[*THEN predicate2D*])

lemma *StateT-parametric* [*transfer-rule*]: ((*S* ===> *M*) ===> *rel-stateT S M*) *StateT StateT*
by(*auto simp add*: *rel-stateT-def*)

lemma *run-state-parametric* [*transfer-rule*]: (*rel-stateT S M* ===> *S* ===> *M*) *run-state run-state*
by(*auto simp add*: *rel-stateT-def*)

lemma *case-stateT-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *A*) ===> *rel-stateT S M* ===> *A*) *case-stateT case-stateT*
by(*auto 4 3 split*: *stateT.split simp add*: *rel-stateT-def del*: *rel-funI intro*!: *rel-funI dest*: *rel-funD*)

lemma *rec-stateT-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *A*) ===> *rel-stateT S M* ===> *A*) *rec-stateT rec-stateT*
apply(*rule rel-funI*)+
subgoal for ... *m m'* **by**(*cases m*; *cases m'*)(*auto 4 3 simp add*: *rel-stateT-def del*: *rel-funI intro*!: *rel-funI dest*: *rel-funD*)
done

lemma *rel-stateT-Grp*: *rel-stateT* (=) (*BNF-Def.Grp UNIV f*) = *BNF-Def.Grp UNIV* (*map-stateT f*)
by(*auto simp add*: *fun-eq-iff Grp-def rel-stateT-def rel-fun-def stateT.map-sel intro*: *stateT.expand*)

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* (λ*s*. *bind* (*x s*) (λ(*a*, *s'*). *run-state* (*f a*) *s'*))

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

definition *get-state* :: ('*s*, ('*s*, '*m*) *stateT*) *get*
where *get-state f* = *StateT* (λ*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*
by(*cases m*) *simp*

lemma *run-get-state* [*simp*]: *run-state* (*get-state f*) *s* = *run-state* (*f s*) *s*
by(*simp add*: *get-state-def*)

lemma *run-bind-state* [*simp*]:
  run-state (*bind-state x f*) *s* = *bind* (*run-state x s*) (λ(*a*, *s'*). *run-state* (*f a*) *s'*)
by(*cases x*)(*simp*)

lemma *run-return-state* [*simp*]:
  run-state (*return-state x*) *s* = *return* (*x*, *s*)
by(*simp add*: *return-state-def*)

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

interpretation *monad return bind* **by**(*fact monad*)

lemma *monad-stateT* [*locale-witness*]: *monad return-state bind-state* (**is** *monad ?return ?bind*)
proof
  show *?bind* (*?bind x f*) *g* = *?bind x* (λ*x*. *?bind* (*f x*) *g*) **for** *x* **and** *f g* :: '*a* ⇒ ('*s*, '*m*) *stateT*
    by(*rule stateT.expand ext*)+(*simp add*: *bind-assoc split-def*)
  show *?bind* (*?return x*) *f* = *f x* **for** *f* :: '*a* ⇒ ('*s*, '*m*) *stateT* **and** *x*
    by(*rule stateT.expand ext*)+(*simp add*: *return-bind*)
  show *?bind x ?return* = *x* **for** *x*
    by(*rule stateT.expand ext*)+(*simp add*: *bind-return*)
qed

lemma *monad-state-stateT* [*locale-witness*]:
  monad-state return-state bind-state get-state put-state

proof
show *put-state s (get-state f) = put-state s (f s)* **for** *f :: 's ⇒ ('s, 'm) stateT* **and** *s :: 's*
by(*rule stateT.expand*)(*simp add*: *get-state-def fun-eq-iff*)
show *get-state (λs. get-state (f s)) = get-state (λs. f s s)* **for** *f :: 's ⇒ 's ⇒ ('s, 'm) stateT*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *put-state s (put-state s' m) = put-state s' m* **for** *s s' :: 's* **and** *m :: ('s, 'm) stateT*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *get-state (λs :: 's. put-state s m) = m* **for** *m :: ('s, 'm) stateT*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *get-state (λ-. m) = m* **for** *m :: ('s, 'm) stateT*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *bind-state (get-state f) g = get-state (λs. bind-state (f s) g)* **for** *f g*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *bind-state (put-state s m) f = put-state s (bind-state m f)* **for** *s :: 's* **and** *m f*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
qed

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 (λs. fail)*

lemma *run-fail-state* [*simp*]: *run-state fail-state s = fail*
by(*simp add*: *fail-state-def*)

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 −

interpret *monad-fail return bind fail* **by**(*fact assms*)
have *?bind ?fail f* = *?fail* **for** *f* **by**(*rule stateT.expand*)(*simp add*: *fun-eq-iff fail-bind*)
then show *?thesis* **by** *unfold-locales*
qed

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.

fix *catch* :: *'m catch*
assume *monad-catch return bind fail catch*
then interpret *monad-catch return bind fail catch* **.**

define *catch-state* :: (*'s*, *'m*) *stateT catch* **where**
catch-state m1 m2 = *StateT* (λ*s*. *catch* (*run-state m1 s*) (*run-state m2 s*)) **for** *m1 m2*
have *monad-catch return-state bind-state fail-state catch-state*
by(*unfold-locales*; *rule stateT.expand*; *simp add*: *fun-eq-iff catch-state-def catch-return catch-fail catch-fail2 catch-assoc*)
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* (λ*r*. *run-state* (*f r*) *s*)
by(*simp add*: *ask-state-def*)

lemma *monad-reader-stateT* [*locale-witness*]:
assumes *monad-reader return bind ask*
shows *monad-reader return-state bind-state ask-state*
proof −
interpret *monad-reader return bind ask* **by**(*fact assms*)
show *?thesis*
proof
show *ask-state* (λ*r*. *ask-state* (*f r*)) = *ask-state* (λ*r*. *f r r*) **for** *f* :: *'r* ⇒ *'r* ⇒ (*'s*, *'m*) *stateT*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff ask-ask*)
show *ask-state* (λ-. *x*) = *x* **for** *x*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff ask-const*)

show *bind-state* (*ask-state f*) *g* = *ask-state* (λ*r*. *bind-state* (*f r*) *g*) **for** *f g*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff bind-ask*)
show *bind-state m* (λ*x*. *ask-state* (*f x*)) = *ask-state* (λ*r*. *bind-state m* (λ*x*. *f x r*)) **for** *m f*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff bind-ask2 split-def*)
qed
qed

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 −
interpret *monad-reader return bind ask* **by**(*fact assms*)
show *?thesis*
proof
show *ask-state* (λ*r*. *get-state* (*f r*)) = *get-state* (λ*s*. *ask-state* (λ*r*. *f r s*)) **for** *f*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *put-state m* (*ask-state f*) = *ask-state* (λ*r*. *put-state m* (*f r*)) **for** *m f*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
qed
qed

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* (λ*s*. *altc-sample p* (λ*x*. *run-state* (*f x*) *s*))

lemma *run-altc-sample-state* [*simp*]:
run-state (*altc-sample-state altc-sample p f*) *s* = *altc-sample p* (λ*x*. *run-state* (*f x*) *s*)
by(*simp add*: *altc-sample-state-def*)

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* **by**(*fact*)

lemma *monad-prob-stateT* [*locale-witness*]: *monad-prob return-state bind-state sample-state*
including *lifting-syntax*
proof
note *sample-parametric*[*transfer-rule*]
show *sample-state p* (λ-. *x*) = *x* **for** *p x*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff sample-const*)
show *sample-state* (*return-pmf x*) *f* = *f x* **for** *f x*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff sample-return-pmf*)
show *sample-state* (*bind-pmf p f*) *g* = *sample-state p* (λ*x*. *sample-state* (*f x*) *g*) **for** *p f g*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff sample-bind-pmf*)
show *sample-state p* (λ*x*. *sample-state q* (*f x*)) = *sample-state q* (λ*y*. *sample-state p* (λ*x*. *f x y*)) **for** *p q f*
by(*rule stateT.expand*)(*auto simp add*: *fun-eq-iff intro*: *sample-commute*)
show *bind-state* (*sample-state p f*) *g* = *sample-state p* (λ*x*. *bind-state* (*f x*) *g*) **for** *p f g*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff bind-sample1*)
show *bind-state m* (λ*y*. *sample-state p* (*f y*)) = *sample-state p* (λ*x*. *bind-state m* (λ*y*. *f y x*)) **for** *m p f*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff bind-sample2 split-def*)
show (*rel-pmf R* ===> (*R* ===> (=)) ===> (=)) *sample-state sample-state*
if [*transfer-rule*]: *bi-unique R* **for** *R* **unfolding** *altc-sample-state-def* **by** *transfer-prover*
qed

lemma *monad-state-prob-stateT* [*locale-witness*]:
monad-state-prob return-state bind-state get-state put-state sample-state
proof
show *sample-state p* (λ*x*. *get-state* (*f x*)) = *get-state* (λ*s*. *sample-state p* (λ*x*. *f x s*)) **for** *p f*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
qed

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*)
by(*simp add*: *tell-state-def*)

lemma *monad-writer-stateT* [*locale-witness*]:
assumes *monad-writer return bind tell*
shows *monad-writer return-state bind-state tell-state*
proof −
interpret *monad-writer return bind tell* **by**(*fact assms*)
show *?thesis*
proof
show *bind-state* (*tell-state w m*) *f* = *tell-state w* (*bind-state m f*) **for** *w m f*
by(*rule stateT.expand*)(*simp add*: *bind-tell fun-eq-iff*)
qed
qed

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* (*λs. 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*)
by(*simp add*: *alt-state-def*)

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

interpretation *monad-alt return bind alt* **by** *fact*

lemma *monad-alt-stateT* [*locale-witness*]: *monad-alt return-state bind-state alt-state*
proof
show *alt-state* (*alt-state m1 m2*) *m3* = *alt-state m1* (*alt-state m2 m3*) **for** *m1 m2 m3*
by(*rule stateT.expand*)(*simp add*: *alt-assoc fun-eq-iff*)
show *bind-state* (*alt-state m m'*) *f* = *alt-state* (*bind-state m f*) (*bind-state m' f*) **for** *m m' f*
by(*rule stateT.expand*)(*simp add*: *bind-alt1 fun-eq-iff*)
qed

lemma *monad-state-alt-stateT* [*locale-witness*]:
monad-state-alt return-state bind-state get-state put-state alt-state
proof
show *alt-state* (*get-state f*) (*get-state g*) = *get-state* (*λx. alt-state* (*f x*) (*g x*))
for *f g* **by**(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *alt-state* (*put-state s m*) (*put-state s m'*) = *put-state s* (*alt-state m m'*)
for *s m m'* **by**(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)

qed

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 −
  interpret *monad-fail-alt return bind fail alt* **by** *fact*
  show *?thesis*
  proof
    show *alt-state* (*fail-state fail*) *m* = *m* **for** *m*
      by(*rule stateT.expand*)(*simp add*: *fun-eq-iff alt-fail1*)
    show *alt-state m* (*fail-state fail*) = *m* **for** *m*
      by(*rule stateT.expand*)(*simp add*: *fun-eq-iff alt-fail2*)
  qed
qed

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* **by** *fact*

lemma *monad-altc-stateT* [*locale-witness*]: *monad-altc return-state bind-state altc-state*
proof
  note *altc-parametric*[*transfer-rule*]
  show *bind-state* (*altc-state C g*) *f* = *altc-state C* (λ*c*. *bind-state* (*g c*) *f*) **for** *C g f*
    by(*rule stateT.expand*)(*simp add*: *fun-eq-iff bind-altc1*)
  show *altc-state* (*csingle x*) *f* = *f x* **for** *x f*
    by(*rule stateT.expand*)(*simp add*: *fun-eq-iff altc-single*)
  show *altc-state* (*cUNION C f*) *g* = *altc-state C* (λ*x*. *altc-state* (*f x*) *g*) **for** *C f g*
    by(*rule stateT.expand*)(*simp add*: *fun-eq-iff altc-cUNION*)
  show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) *altc-state altc-state* **if** [*transfer-rule*]: *bi-unique R* **for** *R*

unfolding *altc-sample-state-def* **by** *transfer-prover*
qed

lemma *monad-state-altc-stateT* [*locale-witness*]:
monad-state-altc return-state bind-state get-state put-state altc-state
proof
show *altc-state C* (λ*c. get-state* (*f c*)) = *get-state* (λ*s. altc-state C* (λ*c. f c s*))
for *C* **and** *f* :: *'c* ⇒ *'s* ⇒ (*'s*, *'m*) *stateT* **by**(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
show *altc-state C* (λ*c. put-state s* (*f c*)) = *put-state s* (*altc-state C f*)
for *C s* **and** *f* :: *'c* ⇒ (*'s*, *'m*) *stateT* **by**(*rule stateT.expand*)(*simp add*: *fun-eq-iff*)
qed

end

lemma *monad-altc3-stateT* [*locale-witness*]:
assumes *monad-altc3 return bind altc*
shows *monad-altc3 return-state bind-state altc-state*
proof −
interpret *monad-altc3 return bind altc* **by** *fact*
show *?thesis* **..**
qed

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* (λ*s. pause out* (λ*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*)
by(*simp add*: *pause-state-def*)

lemma *monad-resumption-stateT* [*locale-witness*]:
assumes *monad-resumption return bind pause*
shows *monad-resumption return-state bind-state pause-state*
proof −
interpret *monad-resumption return bind pause* **by** *fact*
show *?thesis*
proof
show *bind-state* (*pause-state out c*) *f* = *pause-state out* (λ*i. bind-state* (*c i*) *f*)
for *out c f*
by(*rule stateT.expand*)(*simp add*: *fun-eq-iff bind-pause*)

qed
qed

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*
unfolding *return-state-def* **by** *transfer-prover*

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
unfolding *bind-state-def* **by** *transfer-prover*

lemma *get-state-parametric* [*transfer-rule*]:
((*S* ===> *rel-stateT S M*) ===> *rel-stateT S M*) *get-state get-state*
unfolding *get-state-def* **by** *transfer-prover*

lemma *put-state-parametric* [*transfer-rule*]:
(*S* ===> *rel-stateT S M* ===> *rel-stateT S M*) *put-state put-state*
unfolding *put-state-def* **by** *transfer-prover*

lemma *fail-state-parametric* [*transfer-rule*]: (*M* ===> *rel-stateT S M*) *fail-state fail-state*
unfolding *fail-state-def* **by** *transfer-prover*

lemma *ask-state-parametric* [*transfer-rule*]:
(((*R* ===> *M*) ===> *M*) ===> (*R* ===> *rel-stateT S M*) ===> *rel-stateT S M*) *ask-state ask-state*
unfolding *ask-state-def* **by** *transfer-prover*

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
unfolding *altc-sample-state-def* **by** *transfer-prover*

lemma *tell-state-parametric* [*transfer-rule*]:
((*W* ===> *M* ===> *M*) ===> *W* ===> *rel-stateT S M* ===> *rel-stateT S M*)
tell-state tell-state

unfolding *tell-state-def* **by** *transfer-prover*

lemma *alt-state-parametric* [*transfer-rule*]:
((*M* ===> *M* ===> *M*) ===> *rel-stateT S M* ===> *rel-stateT S M* ===> *rel-stateT S M*)
alt-state alt-state
unfolding *alt-state-def* **by** *transfer-prover*

lemma *pause-state-parametric* [*transfer-rule*]:
((*Out* ===> (*In* ===> *M*) ===> *M*) ===> *Out* ===> (*In* ===> *rel-stateT S M*) ===> *rel-stateT S M*)
pause-state pause-state
unfolding *pause-state-def* **by** *transfer-prover*

end

3.9 Writer monad transformer

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

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

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

definition *return-writer* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *return*
where *return-writer x* = *WriterT* (*return* (*x*, []))

definition *bind-writer* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *bind*
where *bind-writer m f* = *WriterT* (*bind* (*run-writer m*) (λ(*a*, *ws*). *bind* (*run-writer* (*f a*)) (λ(*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*) (λ(*a*, *ws*). *return* (*a*, *w* # *ws*)))

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

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'*)))
by(*simp add*: *bind-writer-def*)

lemma *run-tell-writer* [*simp*]:
run-writer (*tell-writer w m*) = *bind* (*run-writer m*) (λ(*a*, *ws*). *return* (*a*, *w* # *ws*))
by(*simp add*: *tell-writer-def*)

context
assumes *monad return bind*
begin

interpretation *monad return bind* **by** *fact*

lemma *monad-writerT* [*locale-witness*]: *monad return-writer bind-writer*
proof
show *bind-writer* (*bind-writer x f*) *g* = *bind-writer x* (λ*y*. *bind-writer* (*f y*) *g*) **for** *x f g*
by(*rule writerT.expand*)(*simp add*: *bind-assoc split-def return-bind*)
show *bind-writer* (*return-writer x*) *f* = *f x* **for** *x f*
by(*rule writerT.expand*)(*simp add*: *bind-return return-bind*)
show *bind-writer x return-writer* = *x* **for** *x*
by(*rule writerT.expand*)(*simp add*: *bind-return return-bind*)
qed

lemma *monad-writer-writerT* [*locale-witness*]: *monad-writer return-writer bind-writer tell-writer*
proof
show *bind-writer* (*tell-writer w m*) *f* = *tell-writer w* (*bind-writer m f*) **for** *w m f*
by(*rule writerT.expand*)(*simp add*: *bind-assoc split-def return-bind*)
qed

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*
by(*simp add*: *fail-writer-def*)

lemma *monad-fail-writerT* [*locale-witness*]:
assumes *monad-fail return bind fail*
shows *monad-fail return-writer bind-writer fail-writer*
proof −
interpret *monad-fail return bind fail* **by** *fact*
show *?thesis*

proof
  show *bind-writer fail-writer f* = *fail-writer* **for** *f*
    by(*rule writerT.expand*)(*simp add*: *fail-bind*)
  qed
qed

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* (λ*s*. *run-writer* (*f s*)))

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

lemma *run-get-writer* [*simp*]: *run-writer* (*get-writer f*) = *get* (λ*s*. *run-writer* (*f s*))
by(*simp add*: *get-writer-def*)

lemma *run-put-writer* [*simp*]: *run-writer* (*put-writer s m*) = *put s* (*run-writer m*)
by(*simp add*: *put-writer-def*)

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 −
  interpret *monad-state return bind get put* **by** *fact*
  show *?thesis*
  proof
    show *put-writer s* (*get-writer f*) = *put-writer s* (*f s*) **for** *s f*
      by(*rule writerT.expand*)(*simp add*: *put-get*)
    show *get-writer* (λ*s*. *get-writer* (*f s*)) = *get-writer* (λ*s*. *f s s*) **for** *f*
      by(*rule writerT.expand*)(*simp add*: *get-get*)
    show *put-writer s* (*put-writer s' m*) = *put-writer s' m* **for** *s s' m*
      by(*rule writerT.expand*)(*simp add*: *put-put*)
    show *get-writer* (λ*s*. *put-writer s m*) = *m* **for** *m*
      by(*rule writerT.expand*)(*simp add*: *get-put*)
    show *get-writer* (λ-. *m*) = *m* **for** *m*
      by(*rule writerT.expand*)(*simp add*: *get-const*)
    show *bind-writer* (*get-writer f*) *g* = *get-writer* (λ*s*. *bind-writer* (*f s*) *g*) **for** *f g*
      by(*rule writerT.expand*)(*simp add*: *bind-get*)
    show *bind-writer* (*put-writer s m*) *f* = *put-writer s* (*bind-writer m f*) **for** *s m f*

by(*rule writerT.expand*)(*simp add*: *bind-put*)
qed
qed

3.9.3 Probability

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

lemma *run-altc-sample-writer* [*simp*]:
run-writer (*altc-sample-writer altc-sample p f*) = *altc-sample p* (λ*p*. *run-writer* (*f p*))
by(*simp add*: *altc-sample-writer-def*)

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 −
interpret *monad-prob return bind sample* **by** *fact*
note *sample-parametric*[*transfer-rule*]
show *?thesis* **including** *lifting-syntax*
proof
show *sample-writer p* (λ-. *m*) = *m* **for** *p m*
by(*rule writerT.expand*)(*simp add*: *sample-const*)
show *sample-writer* (*return-pmf x*) *f* = *f x* **for** *x f*
by(*rule writerT.expand*)(*simp add*: *sample-return-pmf*)
show *sample-writer* (*p* ⤜ *f*) *g* = *sample-writer p* (λ*x*. *sample-writer* (*f x*) *g*) **for** *p f g*
by(*rule writerT.expand*)(*simp add*: *sample-bind-pmf*)
show *sample-writer p* (λ*x*. *sample-writer q* (*f x*)) = *sample-writer q* (λ*y*. *sample-writer p* (λ*x*. *f x y*))
for *p q f* **by**(*rule writerT.expand*)(*auto intro*: *sample-commute*)
show *bind-writer* (*sample-writer p f*) *g* = *sample-writer p* (λ*x*. *bind-writer* (*f x*) *g*) **for** *p f g*
by(*rule writerT.expand*)(*simp add*: *bind-sample1*)
show *bind-writer m* (λ*y*. *sample-writer p* (*f y*)) = *sample-writer p* (λ*x*. *bind-writer m* (λ*y*. *f y x*))
for *m p f* **by**(*rule writerT.expand*)(*simp add*: *bind-sample2*[*symmetric*] *bind-sample1 split-def*)

show (*rel-pmf R* ===> (*R* ===> (=)) ===> (=)) *sample-writer sample-writer*
if [*transfer-rule*]: *bi-unique R* **for** *R* **unfolding** *altc-sample-writer-def* **by** *transfer-prover*
qed
qed

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 −
interpret *monad-state-prob return bind get put sample* **by** *fact*
show *?thesis*
proof
show *sample-writer p* (λ*x*. *get-writer* (*f x*)) = *get-writer* (λ*s*. *sample-writer p* (λ*x*. *f x s*)) **for** *p f*
by(*rule writerT.expand*)(*simp add*: *sample-get*)
qed
qed

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* (λ*r*. *run-writer* (*f r*)))

lemma *run-ask-writer* [*simp*]: *run-writer* (*ask-writer f*) = *ask* (λ*r*. *run-writer* (*f r*))
by(*simp add*: *ask-writer-def*)

lemma *monad-reader-writerT* [*locale-witness*]:
assumes *monad-reader return bind ask*
shows *monad-reader return-writer bind-writer ask-writer*
proof −
interpret *monad-reader return bind ask* **by** *fact*
show *?thesis*
proof
show *ask-writer* (λ*r*. *ask-writer* (*f r*)) = *ask-writer* (λ*r*. *f r r*) **for** *f*
by(*rule writerT.expand*)(*simp add*: *ask-ask*)
show *ask-writer* (λ-. *m*) = *m* **for** *m*
by(*rule writerT.expand*)(*simp add*: *ask-const*)
show *bind-writer* (*ask-writer f*) *g* = *ask-writer* (λ*r*. *bind-writer* (*f r*) *g*) **for** *f g*
by(*rule writerT.expand*)(*simp add*: *bind-ask*)
show *bind-writer m* (λ*x*. *ask-writer* (*f x*)) = *ask-writer* (λ*r*. *bind-writer m* (λ*x*.

f x r))
for *m f* **by**(*rule writerT.expand*)(*simp add*: *split-def bind-ask2*[*symmetric*] *bind-ask*)
qed
qed

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 −
interpret *monad-reader-state return bind ask get put* **by** *fact*
show *?thesis*
proof
show *ask-writer* (*λr. get-writer* (*f r*)) = *get-writer* (*λs. ask-writer* (*λr. f r s*))
for *f* **by**(*rule writerT.expand*)(*simp add*: *ask-get*)
show *put-writer s* (*ask-writer f*) = *ask-writer* (*λr. put-writer s* (*f r*)) **for** *s f*
by(*rule writerT.expand*)(*simp add*: *put-ask*)
qed
qed

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* (*λinput. run-writer* (*c input*)))

lemma *run-pause-writer* [*simp*]:
run-writer (*pause-writer out c*) = *pause out* (*λinput. run-writer* (*c input*))
by(*simp add*: *pause-writer-def*)

lemma *monad-resumption-writerT* [*locale-witness*]:
assumes *monad-resumption return bind pause*
shows *monad-resumption return-writer bind-writer pause-writer*
proof −
interpret *monad-resumption return bind pause* **by** *fact*
show *?thesis*
proof
show *bind-writer* (*pause-writer out c*) *f* = *pause-writer out* (*λi. bind-writer* (*c i*) *f*) **for** *out c f*
by(*rule writerT.expand*)(*simp add*: *bind-pause*)
qed
qed

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'*)
by(*simp add*: *alt-writer-def*)

lemma *monad-alt-writerT* [*locale-witness*]:
  assumes *monad-alt return bind alt*
  shows *monad-alt return-writer bind-writer alt-writer*
proof −
  interpret *monad-alt return bind alt* **by** *fact*
  show *?thesis*
  proof
    show *alt-writer* (*alt-writer m1 m2*) *m3* = *alt-writer m1* (*alt-writer m2 m3*)
      for *m1 m2 m3* **by**(*rule writerT.expand*)(*simp add*: *alt-assoc*)
    show *bind-writer* (*alt-writer m m'*) *f* = *alt-writer* (*bind-writer m f*) (*bind-writer m' f*)
      for *m m' f* **by**(*rule writerT.expand*)(*simp add*: *bind-alt1*)
  qed
qed

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 −
  interpret *monad-fail-alt return bind fail alt* **by** *fact*
  show *?thesis*
  proof
    show *alt-writer fail-writer m* = *m* **for** *m*
      by(*rule writerT.expand*)(*simp add*: *alt-fail1*)
    show *alt-writer m fail-writer* = *m* **for** *m*
      by(*rule writerT.expand*)(*simp add*: *alt-fail2*)
  qed
qed

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 −
  interpret *monad-state-alt return bind get put alt* **by** *fact*
  show *?thesis*
  proof
    show *alt-writer* (*get-writer f*) (*get-writer g*) = *get-writer* (*λx. alt-writer* (*f x*)

(*g x*))
for *f g* **by**(*rule writerT.expand*)(*simp add*: *alt-get*)
show *alt-writer* (*put-writer s m*) (*put-writer s m'*) = *put-writer s* (*alt-writer m m'*)
for *s m m'* **by**(*rule writerT.expand*)(*simp add*: *alt-put*)
qed
qed

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 −
interpret *monad-altc return bind altc* **by** *fact*
note *altc-parametric*[*transfer-rule*]
show *?thesis* **including** *lifting-syntax*
proof
show *bind-writer* (*altc-writer C g*) *f* = *altc-writer C* (λ*c*. *bind-writer* (*g c*) *f*) **for** *C g f*
by(*rule writerT.expand*)(*simp add*: *bind-altc1 o-def*)
show *altc-writer* (*csingle x*) *f* = *f x* **for** *x f*
by(*rule writerT.expand*)(*simp add*: *altc-single*)
show *altc-writer* (*cUNION C f*) *g* = *altc-writer C* (λ*x*. *altc-writer* (*f x*) *g*) **for** *C f g*
by(*rule writerT.expand*)(*simp add*: *altc-cUNION o-def*)
show (*rel-cset R* ===> (*R* ===> (=)) ===> (=)) *altc-writer altc-writer*
if [*transfer-rule*]: *bi-unique R* **for** *R* **unfolding** *altc-sample-writer-def* **by** *transfer-prover*
qed
qed

lemma *monad-altc3-writerT* [*locale-witness*]:
assumes *monad-altc3 return bind altc*
shows *monad-altc3 return-writer bind-writer altc-writer*
proof −
interpret *monad-altc3 return bind altc* **by** *fact*
show *?thesis* **..**
qed

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 −
interpret *monad-state-altc return bind get put altc* **by** *fact*
show *?thesis*
proof
show *altc-writer C* (λ*c. get-writer* (*f c*)) = *get-writer* (λ*s. altc-writer C* (λ*c. f c s*))
for *C* **and** *f* :: *'c* ⇒ *'s* ⇒ (*'w*, *'a*, *'m*) *writerT* **by**(*rule writerT.expand*)(*simp add*: *o-def altc-get*)
show *altc-writer C* (λ*c. put-writer s* (*f c*)) = *put-writer s* (*altc-writer C f*)
for *C s* **and** *f* :: *'c* ⇒ (*'w*, *'a*, *'m*) *writerT* **by**(*rule writerT.expand*)(*simp add*: *o-def altc-put*)
qed
qed

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*
unfolding *return-writer-def* **by** *transfer-prover*

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
unfolding *bind-writer-def* **by** *transfer-prover*

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
unfolding *tell-writer-def* **by** *transfer-prover*

lemma *ask-writer-parametric* [*transfer-rule*]:
(((*R* ===> *M*) ===> *M*) ===> (*R* ===> *rel-writerT W A M*) ===> *rel-writerT W A M*) *ask-writer ask-writer*
unfolding *ask-writer-def* **by** *transfer-prover*

lemma *fail-writer-parametric* [*transfer-rule*]:
(*M* ===> *rel-writerT W A M*) *fail-writer fail-writer*
unfolding *fail-writer-def* **by** *transfer-prover*

lemma *get-writer-parametric* [*transfer-rule*]:
(((*S* ===> *M*) ===> *M*) ===> (*S* ===> *rel-writerT W A M*) ===> *rel-writerT W A M*) *get-writer get-writer*
unfolding *get-writer-def* **by** *transfer-prover*

lemma *put-writer-parametric* [*transfer-rule*]:
((*S* ===> *M* ===> *M*) ===> *S* ===> *rel-writerT W A M* ===> *rel-writerT W A M*) *put-writer put-writer*
unfolding *put-writer-def* **by** *transfer-prover*

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
unfolding *altc-sample-writer-def* **by** *transfer-prover*

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
unfolding *alt-writer-def* **by** *transfer-prover*

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
unfolding *pause-writer-def* **by** *transfer-prover*

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*
by(*simp add*: *callcc-cont-def*)

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*
by(*simp add*: *return-cont-def*)

lemma *run-bind-cont* [*simp*]: *run-cont* (*bind-cont m f*) *k* = *run-cont m* (λ*x*. *run-cont* (*f x*) *k*)
by(*simp add*: *bind-cont-def*)

lemma *monad-contT* [*locale-witness*]: *monad return-cont bind-cont* (**is** *monad ?return ?bind*)
proof
  show *?bind* (*?bind x f*) *g* = *?bind x* (λ*x*. *?bind* (*f x*) *g*) **for** *x f g*
    by(*rule contT.expand*)(*simp add*: *fun-eq-iff*)
  show *?bind* (*?return x*) *f* = *f x* **for** *f x*
    by(*rule contT.expand*)(*simp add*: *fun-eq-iff*)
  show *?bind x ?return* = *x* **for** *x*
    by(*rule contT.expand*)(*simp add*: *fun-eq-iff*)
qed

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*
by(*simp add*: *fail-cont-def*)

lemma *monad-fail-contT* [*locale-witness*]: *monad-fail return-cont bind-cont fail-cont*
proof
  show *bind-cont fail-cont f* = *fail-cont* **for** *f* :: '*a* ⇒ ('*a*, '*m*) *contT*
    by(*rule contT.expand*)(*simp add*: *fun-eq-iff*)
qed

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*)
by(*simp add*: *get-cont-def*)

lemma *run-put-cont* [*simp*]: *run-cont* (*put-cont s m*) *k* = *put s* (*run-cont m k*)
by(*simp add*: *put-cont-def*)

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 −
  interpret *monad-state return' bind' get put* **by**(*fact assms*)
  show *?thesis*
  proof
    show *put-cont s* (*get-cont f*) = *put-cont s* (*f s*) **for** *s* :: *'s* **and** *f* :: *'s* ⇒ (*'a*, *'m*) *contT*
      by(*rule contT.expand*)(*simp add*: *put-get fun-eq-iff*)
    show *get-cont* (λ*s*. *get-cont* (*f s*)) = *get-cont* (λ*s*. *f s s*) **for** *f* :: *'s* ⇒ *'s* ⇒ (*'a*, *'m*) *contT*
      by(*rule contT.expand*)(*simp add*: *get-get fun-eq-iff*)
    show *put-cont s* (*put-cont s' m*) = *put-cont s' m* **for** *s s'* **and** *m* :: (*'a*, *'m*) *contT*
      by(*rule contT.expand*)(*simp add*: *put-put fun-eq-iff*)
    show *get-cont* (λ*s*. *put-cont s m*) = *m* **for** *m* :: (*'a*, *'m*) *contT*
      by(*rule contT.expand*)(*simp add*: *get-put fun-eq-iff*)
    show *get-cont* (λ-. *m*) = *m* **for** *m* :: (*'a*, *'m*) *contT*
      by(*rule contT.expand*)(*simp add*: *get-const fun-eq-iff*)
    show *bind-cont* (*get-cont f*) *g* = *get-cont* (λ*s*. *bind-cont* (*f s*) *g*)
      for *f* :: *'s* ⇒ (*'a*, *'m*) *contT* **and** *g*
      by(*rule contT.expand*)(*simp add*: *fun-eq-iff*)
    show *bind-cont* (*put-cont s m*) *f* = *put-cont s* (*bind-cont m f*) **for** *s* **and** *m* :: (*'a*, *'m*) *contT* **and** *f*
      by(*rule contT.expand*)(*simp add*: *fun-eq-iff*)
  qed
qed

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*)
by(*simp add*: *m1.lift-def m2.lift-def hom-bind hom-return o-def*)

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* ∘ *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* ⇒ *'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
show *prob-of-id* (*return-id x*) = *return-pmf x* **for** *x* :: *'a*
by(*simp add*: *prob-of-id-def*)
show *prob-of-id* (*bind-id x f*) = *prob-of-id x* ⤜ *prob-of-id* ∘ *f* **for** *x* :: *'a id* **and** *f*
by(*simp add*: *prob-of-id-def bind-return-pmf*)
qed

inductive *cr-id-prob* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a id* ⇒ *'b prob* ⇒ *bool* **for** *A*
where *A x y* ⟹ *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*
by(*simp add*: *rel-fun-def*)

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*
by(*auto simp add*: *rel-fun-def bind-return-pmf elim*!: *cr-id-prob.cases*)

lemma *cr-id-prob-Grp*: *cr-id-prob* (*BNF-Def.Grp A f*) = *BNF-Def.Grp* {*x. set-id x* ⊆ *A*} (*return-pmf* ∘ *f* ∘ *extract*)
apply(*auto simp add*: *Grp-def fun-eq-iff simp add*: *cr-id-prob.simps intro*: *id.expand*)
subgoal for *x* **by**(*cases x*) *auto*
done

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* $c'\ A\ a\ (b,\ c) \longleftrightarrow A\ a\ b \wedge c' = c$
by(*simp add*: *cr-prod1-def*)

lemma *cr-prod1I*: $A\ a\ b \Longrightarrow$ *cr-prod1* $c'\ A\ a\ (b,\ c')$ **by** *simp*

lemma *cr-prod1-Pair-transfer* [*cr-envT-stateT-transfer*]: (*A* ===> *eq-onp* ((=) *c*) ===> *cr-prod1 c A*) (λa -. *a*) *Pair*
by(*auto simp add*: *rel-fun-def eq-onp-def*)

lemma *cr-prod1-fst-transfer* [*cr-envT-stateT-transfer*]: (*cr-prod1 c A* ===> *A*) (λa. *a*) *fst*
by(*auto simp add*: *rel-fun-def*)

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*
by(*simp add*: *rel-fun-def eq-onp-def*)

lemma *cr-prod1-Grp*: *cr-prod1 c* (*BNF-Def.Grp A f*) = *BNF-Def.Grp A* (λb. (*f b*, *c*))
by(*auto simp add*: *Grp-def fun-eq-iff*)

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*)
by(*simp add*: *cr-envT-stateT-def rel-fun-def eq-onp-def*)

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*
using *assms* **by**(*cases m1*; *cases m2*; *auto simp add*: *eq-onp-def*)

lemma *cr-envT-stateTD*: *cr-envT-stateT s M m1 m2* $\Longrightarrow$ *M* (*run-env m1 s*) (*run-state m2 s*)
by(*auto elim*!: *cr-envT-stateTE dest*: *rel-funD simp add*: *eq-onp-def*)

lemma *cr-envT-stateT-run* [*cr-envT-stateT-transfer*]:
(*cr-envT-stateT s M* ===> *eq-onp* ((=) *s*) ===> *M*) *run-env run-state*
by(*rule rel-funI*)(*auto elim*!: *cr-envT-stateTE*)

lemma *cr-envT-stateT-StateT-EnvT* [*cr-envT-stateT-transfer*]:
((*eq-onp* ((=) *s*) ===> *M*) ===> *cr-envT-stateT s M*) *EnvT StateT*
by(*auto 4 3 dest*: *rel-funD simp add*: *eq-onp-def*)

lemma *cr-envT-stateT-rec* [*cr-envT-stateT-transfer*]:
(((*eq-onp* ((=) *s*) ===> *M*) ===> *C*) ===> *cr-envT-stateT s M* ===> *C*)
rec-envT rec-stateT
by(*auto simp add*: *rel-fun-def elim*!: *cr-envT-stateTE*)

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
unfolding *return-env-def return-state-def* **by** *transfer-prover*

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
apply(*rule rel-funI*)+
apply(*erule cr-envT-stateTE*)
apply(*clarsimp simp add*: *split-def*)
apply(*drule rel-funD*)
apply(*erule rel-funD*)
apply(*simp add*: *eq-onp-def*)
apply(*erule rel-funD*)
apply(*rule rel-funI*)
apply *clarsimp*
apply(*rule cr-envT-stateT-run*[*THEN rel-funD*, *THEN rel-funD*, **where** *B*=*M*])
apply(*erule* (*1*) *rel-funD*)
apply(*simp add*: *eq-onp-def*)
done

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
unfolding *ask-env-def get-state-def*
apply(*rule rel-funI*)+
apply *simp*
apply(*rule cr-envT-stateT-run*[*THEN rel-funD*, *THEN rel-funD*])
apply(*erule rel-funD*)
apply(*simp-all add*: *eq-onp-def*)
done

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*
unfolding *fail-env-def fail-state-def* **by** *transfer-prover*

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* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*a option prob*) *optionT* ⇒ '*b spmf* ⇒ *bool*
where *cr-spmf-prob-optionT A p q* ⟷ *rel-spmf A* (*run-option p*) *q*

lemma *cr-spmf-prob-optionTI*: *rel-spmf A* (*run-option p*) *q* ⟹ *cr-spmf-prob-optionT A p q*
by(*simp add*: *cr-spmf-prob-optionT-def*)

lemma *cr-spmf-prob-optionTD*: *cr-spmf-prob-optionT A p q* ⟹ *rel-spmf A* (*run-option p*) *q*
by(*simp add*: *cr-spmf-prob-optionT-def*)

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*
by(*simp add*: *rel-fun-def cr-spmf-prob-optionTI*)

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*
by(*rule rel-funI cr-spmf-prob-optionTI*)+
(*auto 4 4 simp add*: *run-bind-option bind-spmf-def dest!*: *cr-spmf-prob-optionTD dest*: *rel-funD intro*: *rel-pmf-bindI split*: *option.split*)

lemma *cr-spmf-prob-optionT-fail-transfer*:
cr-spmf-prob-optionT A (*fail-option return-pmf*) (*return-pmf None*)
by(*rule cr-spmf-prob-optionTI*) *simp*

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* (λ*x*. *x*)

by(*auto simp add*: *rel-fun-def dest*: *cr-spmf-prob-optionTD*)

lemma *prob-optionT-of-spmf-transfer*: (*rel-spmf A* ===> *cr-spmf-prob-optionT A*) *prob-optionT-of-spmf* (λ*x*. *x*)
by(*auto simp add*: *rel-fun-def intro*: *cr-spmf-prob-optionTI*)

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* **..**

lift-definition *cset-pmf* :: *'a pmf* ⇒ *'a cset* **is** *set-pmf* **by** *simp*

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*
  by(*simp add*: *fun-eq-iff cr-pmf-cset.simps Grp-def*)

lemma *cr-pmf-cset-return-pmf* [*cr-prob-ndi-transfer*]:
  ((=) ===> *cr-pmf-cset*) *return-pmf csingle*
  by(*simp add*: *cr-pmf-cset.simps rel-fun-def*)(*transfer*; *simp*)

inductive *cr-prob-ndi* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a prob* ⇒ (*'b*, *'b cset id*) *nondetT* ⇒ *bool*
  for *A p B* **where**
  cr-prob-ndi A p B **if** *rel-set A* (*set-pmf p*) (*rcset* (*extract* (*run-nondet B*)))

lemma *cr-prob-ndi-Grp*: *cr-prob-ndi* (*BNF-Def.Grp UNIV f*) = *BNF-Def.Grp UNIV* (*NondetT* ∘ *return-id* ∘ *cimage f* ∘ *cset-pmf*)
  by(*simp add*: *fun-eq-iff cr-prob-ndi.simps rel-set-Grp*)
    (*auto simp add*: *Grp-def cimage.rep-eq cset-pmf.rep-eq cin.rep-eq intro*!: *nondetT.expand id.expand*)

lemma *cr-ndi-prob-return* [*cr-prob-ndi-transfer*]:
  (*A* ===> *cr-prob-ndi A*) *return-pmf return-nondet*
  by(*simp add*: *rel-fun-def cr-prob-ndi.simps*)(*transfer*; *simp add*: *rel-set-def*)

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*
  apply (*clarsimp simp add*: *cr-prob-ndi.simps cUnion.rep-eq cimage.rep-eq intro*!: *rel-funI*)
  apply(*rule Union-transfer*[*THEN rel-funD*])
  apply(*rule image-transfer*[*THEN rel-funD*, *THEN rel-funD*])
   apply(*rule rel-funI*)

apply(*drule* (*1*) *rel-funD*)
apply(*erule cr-prob-ndi.cases*)
apply *assumption*+
done

lemma *cr-ndi-prob-sample* [*cr-prob-ndi-transfer*]:
(*cr-pmf-cset* ===> ((=) ===> *cr-prob-ndi A*) ===> *cr-prob-ndi A*) *bind-pmf altc-nondet*
apply(*clarsimp intro*!: *rel-funI simp add*: *cr-pmf-cset.simps cr-prob-ndi.simps cUnion.rep-eq cimage.rep-eq cset-pmf.rep-eq*)
apply(*rule Union-transfer*[*THEN rel-funD*])
apply(*rule image-transfer*[*THEN rel-funD*, *THEN rel-funD*])
apply(*rule rel-funI*)
apply(*drule* (*1*) *rel-funD*)
apply(*erule cr-prob-ndi.cases*)
apply *assumption*
apply(*simp add*: *rel-set-eq*)
done

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*))
by(*simp add*: *bind-id′-def*)

lemma *extract-return* [*simp*]: *extract* (*return x*) = *x*
by(*simp add*: *return-id-def*)

lemma *monad-id′* [*locale-witness*]: *monad return* (*bind* :: (*′a*, *′a id*) *bind*)
unfolding *bind-id′-def return-id-def* **by**(*rule monad-id*)

lemma *monad-commute-id′* [*locale-witness*]: *monad-commute return* (*bind* :: (*′a*, *′a id*) *bind*)
unfolding *bind-id′-def return-id-def* **by**(*rule monad-commute-id*)

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*)
unfolding *return-prob-def bind-prob-def* **by**(*rule monad-prob*)

lemma *monad-commute-prob′* [*locale-witness*]: *monad-commute return* (*bind* :: (*′a*, *′a prob*) *bind*)
unfolding *return-prob-def bind-prob-def* **by**(*rule monad-commute-prob*)

lemma *monad-prob-prob′* [*locale-witness*]: *monad-prob return* (*bind* :: (*′a*, *′a prob*) *bind*) (*sample* :: (*′p*, *′a prob*) *sample*)

unfolding *return-prob-def bind-prob-def sample-prob-def* **by**(*rule monad-prob-prob*)

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*#} (+) **.**

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 (λA. bind m2 (λB. 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 {#} λx. {#x#} (+)* **.**

lemma *run-bind-nondetT*:
fixes *f :: 'a ⇒ ('a, 'm) nondetT* **shows**
run-nondet (bind m f) = bind (run-nondet m) (λA. mUnionMT TYPE('a) (image-mset (run-nondet ∘ f) A))
by(*simp add: bind-nondetT-def mUnionMT-def munionMT-def[abs-def] mmerge-def*)

lemma *run-return-nondetT [simp]: run-nondet (return x :: ('a, 'm) nondetT) = return {#x#}* **for** *x :: 'a*
by(*simp add: return-nondetT-def*)

lemma *run-fail-nondetT [simp]: run-nondet (fail :: ('a, 'm) nondetT) = return ({#} :: 'a multiset)*
by(*simp add: fail-nondetT-def*)

lemma *run-ask-nondetT [simp]: run-nondet (ask f) = ask (λr. run-nondet (f r))*
by(*simp add: ask-nondetT-def*)

lemma *run-get-nondetT [simp]: run-nondet (get f) = get (λs. run-nondet (f s))*
by(*simp add: get-nondetT-def*)

lemma *run-put-nondetT [simp]: run-nondet (put s m) = put s (run-nondet m)*
by(*simp add: put-nondetT-def*)

lemma *run-alt-nondetT [simp]*:
run-nondet (alt m m' :: ('a, 'm) nondetT) =
bind (run-nondet m) (λA :: 'a multiset. bind (run-nondet m') (λB. return (A + B)))
by(*simp add: alt-nondetT-def*)

end

lemma *monad-nondetT' [locale-witness]*:
monad-commute return (bind :: ('a multiset, 'm) bind)
⟹ monad return (bind :: ('a, ('a, 'm) nondetT) bind)
unfolding *return-nondetT-def bind-nondetT-def* **by**(*rule mset-nondetMs*)

lemma *monad-fail-nondetT′* [*locale-witness*]:
monad-commute return (*bind* :: (′*a multiset*, ′*m*) *bind*)
⟹ *monad-fail return* (*bind* :: (′*a*, (′*a*, ′*m*) *nondetT*) *bind*) *fail*
unfolding *return-nondetT-def bind-nondetT-def fail-nondetT-def* **by**(*rule mset-nondetMs*)

lemma *monad-alt-nondetT′* [*locale-witness*]:
monad-commute return (*bind* :: (′*a multiset*, ′*m*) *bind*)
⟹ *monad-alt return* (*bind* :: (′*a*, (′*a*, ′*m*) *nondetT*) *bind*) *alt*
unfolding *return-nondetT-def bind-nondetT-def alt-nondetT-def* **by**(*rule mset-nondetMs*)

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*
unfolding *return-nondetT-def bind-nondetT-def fail-nondetT-def alt-nondetT-def*
by(*rule mset-nondetMs*)

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*)
unfolding *return-nondetT-def bind-nondetT-def ask-nondetT-def* **by**(*rule mset-nondetMs*)

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*′)
by(*simp add*: *bind-stateT-def*)

lemma *run-return-stateT* [*simp*]: *run-state* (*return x*) *s* = *return* (*x*, *s*)
by(*simp add*: *return-stateT-def*)

lemma *run-put-stateT* [*simp*]: *run-state* (*put s m*) *s*′ = *run-state m s*
by(*simp add*: *put-stateT-def*)

lemma *run-get-state* [*simp*]: *run-state* (*get f*) *s* = *run-state* (*f s*) *s*
by(*simp add*: *get-stateT-def*)

lemma *run-fail-stateT* [*simp*]: *run-state fail s* = *fail*
by(*simp add*: *fail-stateT-def*)

lemma *run-ask-stateT* [*simp*]: *run-state* (*ask f*) *s* = *ask* (λ*r*. *run-state* (*f r*) *s*)
by(*simp add*: *ask-stateT-def*)

lemma *run-sample-stateT* [*simp*]: *run-state* (*sample p f*) *s* = *sample p* (λ*x*. *run-state* (*f x*) *s*)
by(*simp add*: *sample-stateT-def*)

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

by(*simp add*: *tell-stateT-def*)

lemma *run-alt-stateT* [*simp*]: *run-state* (*alt m m′*) *s* = *alt* (*run-state m s*) (*run-state m′ s*)
by(*simp add*: *alt-stateT-def*)

lemma *run-altc-stateT* [*simp*]: *run-state* (*altc C f*) *s* = *altc C* (*λx. run-state* (*f x*) *s*)
by(*simp add*: *altc-stateT-def*)

lemma *run-pause-stateT* [*simp*]: *run-state* (*pause out c*) *s* = *pause out* (*λinput. run-state* (*c input*) *s*)
by(*simp add*: *pause-stateT-def*)

lemma *monad-stateT′* [*locale-witness*]:
monad return (*bind* :: (*′a* × *′s*, *′m*) *bind*) ⟹ *monad return* (*bind* :: (*′a*, (*′s*, *′m*) *stateT*) *bind*)
unfolding *return-stateT-def bind-stateT-def* **by**(*rule monad-stateT*)

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*)
unfolding *return-stateT-def bind-stateT-def get-stateT-def put-stateT-def* **by**(*rule monad-state-stateT*)

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*
unfolding *return-stateT-def bind-stateT-def fail-stateT-def* **by**(*rule monad-fail-stateT*)

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*)
unfolding *return-stateT-def bind-stateT-def ask-stateT-def* **by**(*rule monad-reader-stateT*)

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*
unfolding *return-stateT-def bind-stateT-def ask-stateT-def* **by**(*rule monad-reader-state-stateT*)

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*)
unfolding *return-stateT-def bind-stateT-def sample-stateT-def* **by**(*rule monad-prob-stateT*)

lemma *monad-state-prob-stateT'* [*locale-witness*]:
monad-prob return (*bind* :: (*'a* × *'s*, *'m*) *bind*) (*sample* :: (*'p*, *'m*) *sample*)
⟹ *monad-state-prob return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) *get* (*put* :: (*'s*, (*'s*, *'m*) *stateT*) *put*) (*sample* :: (*'p*, (*'s*, *'m*) *stateT*) *sample*)
unfolding *return-stateT-def bind-stateT-def sample-stateT-def get-stateT-def put-stateT-def*
by(*rule monad-state-prob-stateT*)

lemma *monad-writer-stateT'* [*locale-witness*]:
monad-writer return (*bind* :: (*'a* × *'s*, *'m*) *bind*) (*tell* :: (*'w*, *'m*) *tell*)
⟹ *monad-writer return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) (*tell* :: (*'w*, (*'s*, *'m*) *stateT*) *tell*)
unfolding *return-stateT-def bind-stateT-def tell-stateT-def* **by**(*rule monad-writer-stateT*)

lemma *monad-alt-stateT'* [*locale-witness*]:
monad-alt return (*bind* :: (*'a* × *'s*, *'m*) *bind*) *alt*
⟹ *monad-alt return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) *alt*
unfolding *return-stateT-def bind-stateT-def alt-stateT-def* **by**(*rule monad-alt-stateT*)

lemma *monad-state-alt-stateT'* [*locale-witness*]:
monad-alt return (*bind* :: (*'a* × *'s*, *'m*) *bind*) *alt*
⟹ *monad-state-alt return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) (*get* :: (*'s*, (*'s*, *'m*) *stateT*) *get*) *put alt*
unfolding *return-stateT-def bind-stateT-def get-stateT-def put-stateT-def alt-stateT-def*
by(*rule monad-state-alt-stateT*)

lemma *monad-fail-alt-stateT'* [*locale-witness*]:
monad-fail-alt return (*bind* :: (*'a* × *'s*, *'m*) *bind*) *fail alt*
⟹ *monad-fail-alt return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) *fail alt*
unfolding *return-stateT-def bind-stateT-def fail-stateT-def alt-stateT-def* **by**(*rule monad-fail-alt-stateT*)

lemma *monad-altc-stateT'* [*locale-witness*]:
monad-altc return (*bind* :: (*'a* × *'s*, *'m*) *bind*) (*altc* :: (*'c*, *'m*) *altc*)
⟹ *monad-altc return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) (*altc* :: (*'c*, (*'s*, *'m*) *stateT*) *altc*)
unfolding *return-stateT-def bind-stateT-def altc-stateT-def* **by**(*rule monad-altc-stateT*)

lemma *monad-state-altc-stateT'* [*locale-witness*]:
monad-altc return (*bind* :: (*'a* × *'s*, *'m*) *bind*) (*altc* :: (*'c*, *'m*) *altc*)
⟹ *monad-state-altc return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) (*get* :: (*'s*, (*'s*, *'m*) *stateT*) *get*) *put* (*altc* :: (*'c*, (*'s*, *'m*) *stateT*) *altc*)
unfolding *return-stateT-def bind-stateT-def get-stateT-def put-stateT-def altc-stateT-def*
by(*rule monad-state-altc-stateT*)

lemma *monad-resumption-stateT'* [*locale-witness*]:
monad-resumption return (*bind* :: (*'a* × *'s*, *'m*) *bind*) (*pause* :: (*'o*, *'i*, *'m*) *pause*)
⟹ *monad-resumption return* (*bind* :: (*'a*, (*'s*, *'m*) *stateT*) *bind*) (*pause* :: (*'o*, *'i*, (*'s*, *'m*) *stateT*) *pause*)
unfolding *return-stateT-def bind-stateT-def fail-stateT-def pause-stateT-def* **by**(*rule*

monad-resumption-stateT)

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* ⇒ (*'a*, *'m*) *optionT* **shows**
  run-option (*bind x f*) = *bind* (*run-option x*) (*λx. case x of None* ⇒ *return* (*None* :: *'a option*) | *Some y* ⇒ *run-option* (*f y*))
by(*simp add*: *bind-optionT-def run-bind-option*)

lemma *run-return-optionT* [*simp*]: *run-option* (*return x* :: (*'a*, *'m*) *optionT*) = *return* (*Some x*) **for** *x* :: *'a*
by(*simp add*: *return-optionT-def*)

lemma *run-fail-optionT* [*simp*]: *run-option* (*fail* :: (*'a*, *'m*) *optionT fail*) = *return* (*None* :: *'a option*)
by(*simp add*: *fail-optionT-def*)

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*)
by(*simp add*: *catch-optionT-def*)

lemma *run-ask-optionT* [*simp*]: *run-option* (*ask f*) = *ask* (*λr. run-option* (*f r*))
by(*simp add*: *ask-optionT-def*)

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

lemma *run-put-optionT* [*simp*]: *run-option* (*put s m*) = *put s* (*run-option m*)
by(*simp add*: *put-optionT-def*)

lemma *run-sample-optionT* [*simp*]: *run-option* (*sample p f*) = *sample p* (*λx. run-option* (*f x*))
by(*simp add*: *sample-optionT-def*)

lemma *run-tell-optionT* [*simp*]: *run-option* (*tell w m*) = *tell w* (*run-option m*)
by(*simp add*: *tell-optionT-def*)

lemma *run-alt-optionT* [*simp*]: *run-option* (*alt m m'*) = *alt* (*run-option m*) (*run-option m'*)
by(*simp add*: *alt-optionT-def*)

lemma *run-altc-optionT* [*simp*]: *run-option* (*altc C f*) = *altc C* (*run-option* ∘ *f*)

by(*simp add*: *altc-optionT-def o-def*)

lemma *run-pause-optionT* [*simp*]: *run-option* (*pause out c*) = *pause out* (*λinput*. *run-option* (*c input*))
by(*simp add*: *pause-optionT-def*)

lemma *monad-optionT'* [*locale-witness*]:
monad return (*bind* :: (*'a option*, *'m*) *bind*)
⟹ *monad return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*)
unfolding *return-optionT-def bind-optionT-def* **by**(*rule monad-optionT*)

lemma *monad-fail-optionT'* [*locale-witness*]:
monad return (*bind* :: (*'a option*, *'m*) *bind*)
⟹ *monad-fail return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) *fail*
unfolding *return-optionT-def bind-optionT-def fail-optionT-def* **by**(*rule monad-fail-optionT*)

lemma *monad-catch-optionT'* [*locale-witness*]:
monad return (*bind* :: (*'a option*, *'m*) *bind*)
⟹ *monad-catch return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) *fail catch*
unfolding *return-optionT-def bind-optionT-def fail-optionT-def catch-optionT-def*
by(*rule monad-catch-optionT*)

lemma *monad-reader-optionT'* [*locale-witness*]:
monad-reader return (*bind* :: (*'a option*, *'m*) *bind*) (*ask* :: (*'r*, *'m*) *ask*)
⟹ *monad-reader return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*ask* :: (*'r*, (*'a*, *'m*) *optionT*) *ask*)
unfolding *return-optionT-def bind-optionT-def ask-optionT-def*
by(*rule monad-reader-optionT*)

lemma *monad-state-optionT'* [*locale-witness*]:
monad-state return (*bind* :: (*'a option*, *'m*) *bind*) (*get* :: (*'s*, *'m*) *get*) *put*
⟹ *monad-state return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*get* :: (*'s*, (*'a*, *'m*) *optionT*) *get*) *put*
unfolding *return-optionT-def bind-optionT-def get-optionT-def put-optionT-def*
by(*rule monad-state-optionT*)

lemma *monad-catch-state-optionT'* [*locale-witness*]:
monad-state return (*bind* :: (*'a option*, *'m*) *bind*) (*get* :: (*'s*, *'m*) *get*) *put*
⟹ *monad-catch-state return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) *fail catch* (*get* :: (*'s*, (*'a*, *'m*) *optionT*) *get*) *put*
unfolding *return-optionT-def bind-optionT-def fail-optionT-def catch-optionT-def get-optionT-def put-optionT-def*
by(*rule monad-catch-state-optionT*)

lemma *monad-prob-optionT'* [*locale-witness*]:
monad-prob return (*bind* :: (*'a option*, *'m*) *bind*) (*sample* :: (*'p*, *'m*) *sample*)
⟹ *monad-prob return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*sample* :: (*'p*, (*'a*, *'m*) *optionT*) *sample*)
unfolding *return-optionT-def bind-optionT-def sample-optionT-def*

by(*rule monad-prob-optionT*)

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*)
⟹ *monad-state-prob return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*get* :: (*'s*, (*'a*,
'm) *optionT*) *get*) *put*(*sample* :: (*'p*, (*'a*, *'m*) *optionT*) *sample*)
unfolding *return-optionT-def bind-optionT-def get-optionT-def put-optionT-def sample-optionT-def*
by(*rule monad-state-prob-optionT*)

lemma *monad-writer-optionT'* [*locale-witness*]:
monad-writer return (*bind* :: (*'a option*, *'m*) *bind*) (*tell* :: (*'w*, *'m*) *tell*)
⟹ *monad-writer return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*tell* :: (*'w*, (*'a*, *'m*)
optionT) *tell*)
unfolding *return-optionT-def bind-optionT-def tell-optionT-def* **by**(*rule monad-writer-optionT*)

lemma *monad-alt-optionT'* [*locale-witness*]:
monad-alt return (*bind* :: (*'a option*, *'m*) *bind*) *alt*
⟹ *monad-alt return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) *alt*
unfolding *return-optionT-def bind-optionT-def alt-optionT-def* **by**(*rule monad-alt-optionT*)

lemma *monad-state-alt-optionT'* [*locale-witness*]:
monad-state-alt return (*bind* :: (*'a option*, *'m*) *bind*) (*get* :: (*'s*, *'m*) *get*) *put alt*
⟹ *monad-state-alt return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*get* :: (*'s*, (*'a*,
'm) *optionT*) *get*) *put alt*
unfolding *return-optionT-def bind-optionT-def alt-optionT-def get-optionT-def put-optionT-def*
by(*rule monad-state-alt-optionT*)

lemma *monad-altc-optionT'* [*locale-witness*]:
monad-altc return (*bind* :: (*'a option*, *'m*) *bind*) (*altc* :: (*'c*, *'m*) *altc*)
⟹ *monad-altc return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*altc* :: (*'c*, (*'a*, *'m*)
optionT) *altc*)
unfolding *return-optionT-def bind-optionT-def altc-optionT-def* **by**(*rule monad-altc-optionT*)

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*)
⟹ *monad-state-altc return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*get* :: (*'s*, (*'a*,
'm) *optionT*) *get*) *put* (*altc* :: (*'c*, (*'a*, *'m*) *optionT*) *altc*)
unfolding *return-optionT-def bind-optionT-def altc-optionT-def get-optionT-def
put-optionT-def* **by**(*rule monad-state-altc-optionT*)

lemma *monad-resumption-optionT'* [*locale-witness*]:
monad-resumption return (*bind* :: (*'a option*, *'m*) *bind*) (*pause* :: (*'o*, *'i*, *'m*) *pause*)
⟹ *monad-resumption return* (*bind* :: (*'a*, (*'a*, *'m*) *optionT*) *bind*) (*pause* :: (*'o*,
'i, (*'a*, *'m*) *optionT*) *pause*)
unfolding *return-optionT-def bind-optionT-def pause-optionT-def* **by**(*rule monad-resumption-optionT*)

lemma *monad-commute-optionT′* [*locale-witness*]:
$\llbracket$ *monad-commute return* (*bind* :: (′*a option*, ′*m*) *bind*); *monad-discard return* (*bind* :: (′*a option*, ′*m*) *bind*) $\rrbracket$
$\Longrightarrow$ *monad-commute return* (*bind* :: (′*a*, (′*a*, ′*m*) *optionT*) *bind*)
unfolding *return-optionT-def bind-optionT-def* **by**(*rule monad-commute-optionT*)

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-envT ≡ *put* :: (′*s*, (′*r*, ′*m*) *envT*) *put*
sample-envT ≡ *sample* :: (′*p*, (′*r*, ′*m*) *envT*) *sample*
ask-envT ≡ *ask* :: (′*r*, (′*r*, ′*m*) *envT*) *ask*
catch-envT ≡ *catch* :: (′*r*, ′*m*) *envT catch*
alt-envT ≡ *alt* :: (′*r*, ′*m*) *envT alt*
altc-envT ≡ *altc* :: (′*c*, (′*r*, ′*m*) *envT*) *altc*
pause-envT ≡ *pause* :: (′*o*, ′*i*, (′*r*, ′*m*) *envT*) *pause*
tell-envT ≡ *tell* :: (′*w*, (′*r*, ′*m*) *envT*) *tell*
begin

definition *return-envT* :: (′*a*, (′*r*, ′*m*) *envT*) *return*
where [*code-unfold*, *monad-unfold*]: *return-envT* = *return-env return*

definition *bind-envT* :: (′*a*, (′*r*, ′*m*) *envT*) *bind*
where [*code-unfold*, *monad-unfold*]: *bind-envT* = *bind-env bind*

definition *ask-envT* :: (′*r*, (′*r*, ′*m*) *envT*) *ask*
where [*code-unfold*, *monad-unfold*]: *ask-envT* = *ask-env*

definition *fail-envT* :: (′*r*, ′*m*) *envT fail*
where [*code-unfold*, *monad-unfold*]: *fail-envT* = *fail-env fail*

definition *get-envT* :: (′*s*, (′*r*, ′*m*) *envT*) *get*
where [*code-unfold*, *monad-unfold*]: *get-envT* = *get-env get*

definition *put-envT* :: (′*s*, (′*r*, ′*m*) *envT*) *put*
where [*code-unfold*, *monad-unfold*]: *put-envT* = *put-env put*

definition *sample-envT* :: (′*p*, (′*r*, ′*m*) *envT*) *sample*
where [*code-unfold*, *monad-unfold*]: *sample-envT* = *sample-env sample*

definition *catch-envT* :: (′*r*, ′*m*) *envT catch*
where [*code-unfold*, *monad-unfold*]: *catch-envT* = *catch-env catch*

definition *alt-envT* :: (′*r*, ′*m*) *envT alt*
where [*code-unfold*, *monad-unfold*]: *alt-envT* = *alt-env alt*

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

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

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

end

lemma *run-bind-envT* [*simp*]: *run-env* (*bind x f*) *r* = *bind* (*run-env x r*) (λ*y*. *run-env* (*f y*) *r*)
by(*simp add*: *bind-envT-def*)

lemma *run-return-envT* [*simp*]: *run-env* (*return x*) *r* = *return x*
by(*simp add*: *return-envT-def*)

lemma *run-ask-envT* [*simp*]: *run-env* (*ask f*) *r* = *run-env* (*f r*) *r*
by(*simp add*: *ask-envT-def*)

lemma *run-fail-envT* [*simp*]: *run-env fail r* = *fail*
by(*simp add*: *fail-envT-def*)

lemma *run-get-envT* [*simp*]: *run-env* (*get f*) *r* = *get* (λ*s*. *run-env* (*f s*) *r*)
by(*simp add*: *get-envT-def*)

lemma *run-put-envT* [*simp*]: *run-env* (*put s m*) *r* = *put s* (*run-env m r*)
by(*simp add*: *put-envT-def*)

lemma *run-sample-envT* [*simp*]: *run-env* (*sample p f*) *r* = *sample p* (λ*x*. *run-env* (*f x*) *r*)
by(*simp add*: *sample-envT-def*)

lemma *run-catch-envT* [*simp*]: *run-env* (*catch m h*) *r* = *catch* (*run-env m r*) (*run-env h r*)
by(*simp add*: *catch-envT-def*)

lemma *run-alt-envT* [*simp*]: *run-env* (*alt m m'*) *r* = *alt* (*run-env m r*) (*run-env m' r*)
by(*simp add*: *alt-envT-def*)

lemma *run-altc-envT* [*simp*]: *run-env* (*altc C f*) *r* = *altc C* (λ*x*. *run-env* (*f x*) *r*)
by(*simp add*: *altc-envT-def*)

lemma *run-pause-envT* [*simp*]: *run-env* (*pause out c*) *r* = *pause out* (λ*input*. *run-env* (*c input*) *r*)
by(*simp add*: *pause-envT-def*)

lemma *run-tell-envT* [*simp*]: *run-env* (*tell s m*) *r* = *tell s* (*run-env m r*)
by(*simp add*: *tell-envT-def*)

lemma *monad-envT'* [*locale-witness*]:
monad return (*bind* :: ('*a*, '*m*) *bind*)
⟹ *monad return* (*bind* :: ('*a*, ('*r*, '*m*) *envT*) *bind*)
unfolding *return-envT-def bind-envT-def* **by**(*rule monad-envT*)

lemma *monad-reader-envT'* [*locale-witness*]:
monad return (*bind* :: ('*a*, '*m*) *bind*)
⟹ *monad-reader return* (*bind* :: ('*a*, ('*r*, '*m*) *envT*) *bind*) (*ask* :: ('*r*, ('*r*, '*m*) *envT*) *ask*)
unfolding *return-envT-def bind-envT-def ask-envT-def* **by**(*rule monad-reader-envT*)

lemma *monad-fail-envT'* [*locale-witness*]:
monad-fail return (*bind* :: ('*a*, '*m*) *bind*) *fail*
⟹ *monad-fail return* (*bind* :: ('*a*, ('*r*, '*m*) *envT*) *bind*) *fail*
unfolding *return-envT-def bind-envT-def fail-envT-def* **by**(*rule monad-fail-envT*)

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*
unfolding *return-envT-def bind-envT-def fail-envT-def catch-envT-def* **by**(*rule monad-catch-envT*)

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*
unfolding *return-envT-def bind-envT-def get-envT-def put-envT-def* **by**(*rule monad-state-envT*)

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*)
unfolding *return-envT-def bind-envT-def sample-envT-def* **by**(*rule monad-prob-envT*)

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*)
unfolding *return-envT-def bind-envT-def sample-envT-def get-envT-def put-envT-def*
by(*rule monad-state-prob-envT*)

lemma *monad-alt-envT'* [*locale-witness*]:
monad-alt return (*bind* :: ('*a*, '*m*) *bind*) *alt*
⟹ *monad-alt return* (*bind* :: ('*a*, ('*r*, '*m*) *envT*) *bind*) *alt*
unfolding *return-envT-def bind-envT-def alt-envT-def* **by**(*rule monad-alt-envT*)

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*
unfolding *return-envT-def bind-envT-def fail-envT-def alt-envT-def* **by**(*rule monad-fail-alt-envT*)

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*
unfolding *return-envT-def bind-envT-def fail-envT-def get-envT-def put-envT-def alt-envT-def* **by**(*rule monad-state-alt-envT*)

lemma *monad-altc-envT'* [*locale-witness*]:
monad-altc return (*bind* :: (*'a*, *'m*) *bind*) (*altc* :: (*'c*, *'m*) *altc*)
⟹ *monad-altc return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*) (*altc* :: (*'c*, (*'r*, *'m*) *envT*) *altc*)
unfolding *return-envT-def bind-envT-def altc-envT-def* **by**(*rule monad-altc-envT*)

lemma *monad-state-altc-envT'* [*locale-witness*]:
monad-state-altc return (*bind* :: (*'a*, *'m*) *bind*) (*get* :: (*'s*, *'m*) *get*) *put* (*altc* :: (*'c*, *'m*) *altc*)
⟹ *monad-state-altc return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*) (*get* :: (*'s*, (*'r*, *'m*) *envT*) *get*) *put* (*altc* :: (*'c*, (*'r*, *'m*) *envT*) *altc*)
unfolding *return-envT-def bind-envT-def fail-envT-def get-envT-def put-envT-def altc-envT-def* **by**(*rule monad-state-altc-envT*)

lemma *monad-resumption-envT'* [*locale-witness*]:
monad-resumption return (*bind* :: (*'a*, *'m*) *bind*) (*pause* :: (*'o*, *'i*, *'m*) *pause*)
⟹ *monad-resumption return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*) (*pause* :: (*'o*, *'i*, (*'r*, *'m*) *envT*) *pause*)
unfolding *return-envT-def bind-envT-def pause-envT-def* **by**(*rule monad-resumption-envT*)

lemma *monad-writer-readerT'* [*locale-witness*]:
monad-writer return (*bind* :: (*'a*, *'m*) *bind*) (*tell* :: (*'w*, *'m*) *tell*)
⟹ *monad-writer return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*) (*tell* :: (*'w*, (*'r*, *'m*) *envT*) *tell*)
unfolding *return-envT-def bind-envT-def tell-envT-def* **by**(*rule monad-writer-envT*)

lemma *monad-commute-envT'* [*locale-witness*]:
monad-commute return (*bind* :: (*'a*, *'m*) *bind*)
⟹ *monad-commute return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*)
unfolding *return-envT-def bind-envT-def* **by**(*rule monad-commute-envT*)

lemma *monad-discard-envT'* [*locale-witness*]:
monad-discard return (*bind* :: (*'a*, *'m*) *bind*)
⟹ *monad-discard return* (*bind* :: (*'a*, (*'r*, *'m*) *envT*) *bind*)
unfolding *return-envT-def bind-envT-def* **by**(*rule monad-discard-envT*)

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'*)))
by(*simp add*: *bind-writerT-def*)

lemma *run-return-writerT* [*simp*]: *run-writer* (*return x* :: (*'w*, *'a*, *'m*) *writerT*) = *return* (*x* :: *'a*, [] :: *'w list*)
by(*simp add*: *return-writerT-def*)

lemma *run-ask-writerT* [*simp*]: *run-writer* (*ask f*) = *ask* (λ*r*. *run-writer* (*f r*))
by(*simp add*: *ask-writerT-def*)

lemma *run-fail-writerT* [*simp*]: *run-writer fail* = *fail*
by(*simp add*: *fail-writerT-def*)

lemma *run-get-writerT* [*simp*]: *run-writer* (*get f*) = *get* (λ*s*. *run-writer* (*f s*))
by(*simp add*: *get-writerT-def*)

lemma *run-put-writerT* [*simp*]: *run-writer* (*put s m*) = *put s* (*run-writer m*)
by(*simp add*: *put-writerT-def*)

lemma *run-sample-writerT* [*simp*]: *run-writer* (*sample p f*) = *sample p* (λ*x*. *run-writer* (*f x*))
by(*simp add*: *sample-writerT-def*)

lemma *run-alt-writerT* [*simp*]: *run-writer* (*alt m m'*) = *alt* (*run-writer m*) (*run-writer m'*)
by(*simp add*: *alt-writerT-def*)

lemma *run-altc-writerT* [*simp*]: *run-writer* (*altc C f*) = *altc C* (*run-writer* ∘ *f*)
by(*simp add*: *altc-writerT-def o-def*)

lemma *run-pause-writerT* [*simp*]: *run-writer* (*pause out c*) = *pause out* (λ*input*. *run-writer* (*c input*))
by(*simp add*: *pause-writerT-def*)

lemma *run-tell-writerT* [*simp*]:
run-writer (*tell* (*w* :: *'w*) *m* :: (*'w*, *'a*, *'m*) *writerT*) =
bind (*run-writer m*) (λ(*a* :: *'a*, *ws* :: *'w list*). *return* (*a*, *w* # *ws*))
by(*simp add*: *tell-writerT-def*)

lemma *monad-writerT'* [*locale-witness*]:
monad return (*bind* :: (*'a* × *'w list*, *'m*) *bind*)
⟹ *monad return* (*bind* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *bind*)
unfolding *return-writerT-def bind-writerT-def* **by**(*rule monad-writerT*)

lemma *monad-writer-writerT'* [*locale-witness*]:
monad return (*bind* :: (*'a* × *'w list*, *'m*) *bind*)

⟹ *monad-writer return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) (*tell* :: (′*w*, (′*w*, ′*a*, ′*m*) *writerT*) *tell*)
unfolding *return-writerT-def bind-writerT-def tell-writerT-def* **by**(*rule monad-writer-writerT*)

lemma *monad-fail-writerT′* [*locale-witness*]:
monad-fail return (*bind* :: (′*a* × ′*w list*, ′*m*) *bind*) *fail*
⟹ *monad-fail return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) *fail*
unfolding *return-writerT-def bind-writerT-def fail-writerT-def* **by**(*rule monad-fail-writerT*)

lemma *monad-state-writerT′* [*locale-witness*]:
monad-state return (*bind* :: (′*a* × ′*w list*, ′*m*) *bind*) (*get* :: (′*s*, ′*m*) *get*) *put*
⟹ *monad-state return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) (*get* :: (′*s*, (′*w*, ′*a*, ′*m*) *writerT*) *get*) *put*
unfolding *return-writerT-def bind-writerT-def get-writerT-def put-writerT-def* **by**(*rule monad-state-writerT*)

lemma *monad-prob-writerT′* [*locale-witness*]:
monad-prob return (*bind* :: (′*a* × ′*w list*, ′*m*) *bind*) (*sample* :: (′*p*, ′*m*) *sample*)
⟹ *monad-prob return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) (*sample* :: (′*p*, (′*w*, ′*a*, ′*m*) *writerT*) *sample*)
unfolding *return-writerT-def bind-writerT-def sample-writerT-def* **by**(*rule monad-prob-writerT*)

lemma *monad-state-prob-writerT′* [*locale-witness*]:
monad-state-prob return (*bind* :: (′*a* × ′*w list*, ′*m*) *bind*) (*get* :: (′*s*, ′*m*) *get*) *put* (*sample* :: (′*p*, ′*m*) *sample*)
⟹ *monad-state-prob return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) (*get* :: (′*s*, (′*w*, ′*a*, ′*m*) *writerT*) *get*) *put* (*sample* :: (′*p*, (′*w*, ′*a*, ′*m*) *writerT*) *sample*)
unfolding *return-writerT-def bind-writerT-def sample-writerT-def get-writerT-def put-writerT-def* **by**(*rule monad-state-prob-writerT*)

lemma *monad-reader-writerT′* [*locale-witness*]:
monad-reader return (*bind* :: (′*a* × ′*w list*, ′*m*) *bind*) (*ask* :: (′*r*, ′*m*) *ask*)
⟹ *monad-reader return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) (*ask* :: (′*r*, (′*w*, ′*a*, ′*m*) *writerT*) *ask*)
unfolding *return-writerT-def bind-writerT-def ask-writerT-def* **by**(*rule monad-reader-writerT*)

lemma *monad-reader-state-writerT′* [*locale-witness*]:
monad-reader-state return (*bind* :: (′*a* × ′*w list*, ′*m*) *bind*) (*ask* :: (′*r*, ′*m*) *ask*) (*get* :: (′*s*, ′*m*) *get*) *put*
⟹ *monad-reader-state return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) (*ask* :: (′*r*, (′*w*, ′*a*, ′*m*) *writerT*) *ask*) (*get* :: (′*s*, (′*w*, ′*a*, ′*m*) *writerT*) *get*) *put*
unfolding *return-writerT-def bind-writerT-def ask-writerT-def get-writerT-def put-writerT-def* **by**(*rule monad-reader-state-writerT*)

lemma *monad-resumption-writerT′* [*locale-witness*]:
monad-resumption return (*bind* :: (′*a* × ′*w list*, ′*m*) *bind*) (*pause* :: (′*o*, ′*i*, ′*m*) *pause*)
⟹ *monad-resumption return* (*bind* :: (′*a*, (′*w*, ′*a*, ′*m*) *writerT*) *bind*) (*pause* :: (′*o*, ′*i*, (′*w*, ′*a*, ′*m*) *writerT*) *pause*)

unfolding *return-writerT-def bind-writerT-def pause-writerT-def* **by**(*rule monad-resumption-writerT*)

lemma *monad-alt-writerT'* [*locale-witness*]:
monad-alt return (*bind* :: (*'a* × *'w list*, *'m*) *bind*) *alt*
⟹ *monad-alt return* (*bind* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *bind*) *alt*
unfolding *return-writerT-def bind-writerT-def alt-writerT-def* **by**(*rule monad-alt-writerT*)

lemma *monad-fail-alt-writerT'* [*locale-witness*]:
monad-fail-alt return (*bind* :: (*'a* × *'w list*, *'m*) *bind*) *fail alt*
⟹ *monad-fail-alt return* (*bind* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *bind*) *fail alt*
unfolding *return-writerT-def bind-writerT-def fail-writerT-def alt-writerT-def* **by**(*rule monad-fail-alt-writerT*)

lemma *monad-state-alt-writerT'* [*locale-witness*]:
monad-state-alt return (*bind* :: (*'a* × *'w list*, *'m*) *bind*) (*get* :: (*'s*, *'m*) *get*) *put alt*
⟹ *monad-state-alt return* (*bind* :: (*'a*, (*'w*, *'a*, *'m*) *writerT*) *bind*) (*get* :: (*'s*, (*'w*, *'a*, *'m*) *writerT*) *get*) *put alt*
unfolding *return-writerT-def bind-writerT-def get-writerT-def put-writerT-def alt-writerT-def* **by**(*rule monad-state-alt-writerT*)

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*)
unfolding *return-writerT-def bind-writerT-def altc-writerT-def* **by**(*rule monad-altc-writerT*)

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*)
unfolding *return-writerT-def bind-writerT-def get-writerT-def put-writerT-def altc-writerT-def* **by**(*rule monad-state-altc-writerT*)

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*)
unfolding *return-contT-def bind-contT-def* **by**(*rule monad-contT*)

lemma *monad-fail-contT'* [*locale-witness*]: *monad-fail return* (*bind* :: ('*a*, ('*a*, '*m*) *contT*) *bind*) *fail*
unfolding *return-contT-def bind-contT-def fail-contT-def* **by**(*rule monad-fail-contT*)

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*
unfolding *return-contT-def bind-contT-def get-contT-def put-contT-def* **by**(*rule monad-state-contT*)

end

7 Examples

7.1 Monadic interpreter

theory *Interpreter* **imports** *Monomorphic-Monad* **begin**

declare [[*show-variants*]]

definition *apply* :: ('*a* $\Rightarrow$ '*b*) $\Rightarrow$ '*a* $\Rightarrow$ '*b* **where** *apply f x* = *f x*

lemma *apply-eq-onp*: **includes** *lifting-syntax* **shows** (*eq-onp P* ===> (=) ===> (=)) *apply apply*
by(*simp add*: *rel-fun-def eq-onp-def*)

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*]:

$rel\text{-}exp\ V\ (Var\ x)\ e' \longleftrightarrow (\exists\, y.\ e' = Var\ y \wedge V\ x\ y)$
$rel\text{-}exp\ V\ (Const\ n)\ e' \longleftrightarrow e' = Const\ n$
$rel\text{-}exp\ V\ (Plus\ e1\ e2)\ e' \longleftrightarrow (\exists\, e1'\ e2'.\ e' = Plus\ e1'\ e2' \wedge rel\text{-}exp\ V\ e1\ e1' \wedge rel\text{-}exp\ V\ e2\ e2')$
$rel\text{-}exp\ V\ (Div\ e1\ e2)\ e' \longleftrightarrow (\exists\, e1'\ e2'.\ e' = Div\ e1'\ e2' \wedge rel\text{-}exp\ V\ e1\ e1' \wedge rel\text{-}exp\ V\ e2\ e2')$
$rel\text{-}exp\ V\ e\ (Var\ y) \longleftrightarrow (\exists\, x.\ e = Var\ x \wedge V\ x\ y)$
$rel\text{-}exp\ V\ e\ (Const\ n) \longleftrightarrow e = Const\ n$
$rel\text{-}exp\ V\ e\ (Plus\ e1'\ e2') \longleftrightarrow (\exists\, e1\ e2.\ e = Plus\ e1\ e2 \wedge rel\text{-}exp\ V\ e1\ e1' \wedge rel\text{-}exp\ V\ e2\ e2')$
$rel\text{-}exp\ V\ e\ (Div\ e1'\ e2') \longleftrightarrow (\exists\, e1\ e2.\ e = Div\ e1\ e2 \wedge rel\text{-}exp\ V\ e1\ e1' \wedge rel\text{-}exp\ V\ e2\ e2')$
by(*auto elim*: *exp.rel-cases*)

lemma *finite-vars* [*simp*]: *finite* (*vars e*)
by *induction auto*

locale *exp-base* = *monad-fail-base return bind fail*
for *return* :: (*int*, *'m*) *return*
and *bind* :: (*int*, *'m*) *bind*
and *fail* :: *'m fail*
begin

context fixes $E :: 'v \Rightarrow 'm$ **begin**
primrec $eval :: 'v\ exp \Rightarrow 'm$
where
$eval\ (Var\ x) = E\ x$
$|\ eval\ (Const\ i) = return\ i$
$|\ eval\ (Plus\ e1\ e2) = bind\ (eval\ e1)\ (\lambda i.\ bind\ (eval\ e2)\ (\lambda j.\ return\ (i + j)))$
$|\ eval\ (Div\ e1\ e2) = bind\ (eval\ e1)\ (\lambda i.\ bind\ (eval\ e2)\ (\lambda j.\ if\ j = 0\ then\ fail\ else\ return\ (i\ div\ j)))$

end

context fixes $\sigma :: 'v \Rightarrow 'w\ exp$ **begin**
primrec $subst :: 'v\ exp \Rightarrow 'w\ exp$
where
$subst\ (Const\ n) = Const\ n$
$|\ subst\ (Var\ x) = \sigma\ x$
$|\ subst\ (Plus\ e1\ e2) = Plus\ (subst\ e1)\ (subst\ e2)$
$|\ subst\ (Div\ e1\ e2) = Div\ (subst\ e1)\ (subst\ e2)$
end

lemma *compositional*: $eval\ E\ (subst\ \sigma\ e) = eval\ (eval\ E \circ \sigma)\ e$
by *induction simp-all*

end

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

includes *lifting-syntax* **shows**
$(((=) ===> M) ===> (M ===> ((=) ===> M) ===> M) ===> M ===> (V ===> M) ===> \mathit{rel\text{-}exp}\ V ===> M)$
exp-base.eval exp-base.eval
unfolding *exp-base.eval-def* **by** *transfer-prover*

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

context *exp-base* **begin**

lemma *eval-cong*:
assumes $\bigwedge x.\ x \in \mathit{vars}\ e \Longrightarrow E\ x = E'\ x$
shows $\mathit{eval}\ E\ e = \mathit{eval}\ E'\ e$
including *lifting-syntax*
proof −
define V **where** $V \equiv \mathit{eq\text{-}onp}\ (\lambda x.\ x \in \mathit{vars}\ e)$
have [*transfer-rule*]: *rel-exp V e e* **by**(*rule exp.rel-refl-strong*)(*simp add*: *V-def eq-onp-def*)
have [*transfer-rule*]: $(V ===> (=))\ E\ E'$ **using** *assms* **by**(*auto simp add*: *V-def rel-fun-def eq-onp-def*)
show *?thesis* **by** *transfer-prover*
qed

end

7.1.2 Memoisation

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

lemma (**in** *monad-base*) *bind-if2*:
$\mathit{bind}\ m\ (\lambda x.\ \mathit{if}\ b\ \mathit{then}\ t\ x\ \mathit{else}\ e\ x) = (\mathit{if}\ b\ \mathit{then}\ \mathit{bind}\ m\ t\ \mathit{else}\ \mathit{bind}\ m\ e)$
by *simp*

lemma (**in** *monad-base*) *bind-case-option2*:
$\mathit{bind}\ m\ (\lambda x.\ \mathit{case\text{-}option}\ (\mathit{none}\ x)\ (\mathit{some}\ x)\ y) = \mathit{case\text{-}option}\ (\mathit{bind}\ m\ \mathit{none})\ (\lambda a.\ \mathit{bind}\ m\ (\lambda x.\ \mathit{some}\ x\ a))\ y$
by(*simp split*: *option.split*)

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 (*λtable*.
case table x of Some y ⇒ *return y*
| *None* ⇒ *bind* (*f x*) (*λy*. *update* (*λm*. *m*(*x* ↦ *y*)) (*return y*)))

lemma *memo-cong* [*cong*, *fundef-cong*]: ⟦ *x* = *y*; *f y* = *g y* ⟧ ⟹ *memo f x* = *memo g y*
by(*simp add*: *memo-def cong del*: *option.case-cong-weak*)

end

declare *memoization-base.memo-def* [*code*]

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

lemma *memo-idem*: *memo* (*memo f*) *x* = *memo f x*
proof −
have *memo* (*memo f*) *x* = *get*
(*λtable*. *case table x of*
None ⇒ *get* (*λtable*′. *bind* (*case table*′ *x of None* ⇒ *bind* (*f x*) (*λy*. *update* (*λm*. *m*(*x* ↦ *y*)) (*return y*))
| *Some x* ⇒ *return x*)
(*λy*. *update* (*λm*. *m* ++ [*x* ↦ *y*]) (*return y*)))
| *Some y* ⇒ *get* (*λ*-. *return y*))
by(*simp add*: *memo-def get-const bind-get cong del*: *option.case-cong-weak*)
also have ... = *memo f x*
by(*simp add*: *option.case-distrib*[**where** *h*=*get*, *symmetric*] *get-get case-option-apply bind-assoc update-update bind-update return-bind o-def memo-def cong*: *option.case-cong*)
finally show *?thesis* **.**
qed

lemma *memo-same*:
bind (*memo f x*) (*λa*. *bind* (*memo f x*) (*g a*)) = *bind* (*memo f x*) (*λa*. *g a a*)
apply(*simp cong*: *option.case-cong add*: *memo-def bind-get option.case-distrib*[**where** *h*=*λx*. *bind x* -] *bind-assoc bind-update return-bind update-get o-def get-const*)
apply(*subst* (*3*) *get-const*[*symmetric*])
apply(*subst option.case-distrib*[**where** *h*=*get*, *symmetric*])
apply(*subst get-get*)
apply(*simp add*: *case-option-apply cong*: *option.case-cong*)
done

lemma *memo-commute*:
assumes *f-bind*: ⋀*m x g*. *bind m* (*λa*. *bind* (*f x*) (*g a*)) = *bind* (*f x*) (*λb*. *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 −
 note *option.case-cong*[*cong*]
 have *update-f*: *update F* (*bind* (*f x*) *g*) = *bind* (*f x*) ($\lambda a.$ *update F* (*g a*)) **for** *F x g*
 proof −
  fix *UU*
  have *update F* (*bind* (*f x*) *g*) = *bind* (*update F* (*return UU*)) (λ-. *bind* (*f x*) *g*)
   by(*simp add*: *bind-update return-bind*)
  also have ... = *bind* (*f x*) ($\lambda a.$ *bind* (*update F* (*return UU*)) (λ-. *g a*))
   by(*rule f-bind*)
  also have ... = *bind* (*f x*) ($\lambda a.$ *update F* (*g a*))
   by(*simp add*: *bind-update return-bind*)
  finally show *?thesis* **.**
 qed
 show *?thesis*
  apply(*clarsimp simp add*: *memo-def bind-get option.case-distrib*[**where** *h*=$\lambda x.$ *bind x* -] *bind-assoc bind-update return-bind update-get o-def f-get*[*symmetric*] *option.case-distrib*[**where** *h*=*get*, *symmetric*] *get-get case-option-apply if-distrib*[**where** *f*=*case-option* - -] *if-distrib*[**where** *f*=*update* -] *option.case-distrib*[**where** *h*=*update* -] *update-f update-update cong*: *if-cong*)
  apply(*clarsimp intro*!: *arg-cong*[**where** *f*=*get*] *ext split*!: *option.split simp add*: *bind-if2*)
  apply(*subst f-bind*)
  apply(*simp add*: *fun-upd-twist*)
  done
qed

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* ⇒ *fail* | *Some y* ⇒ *return y*)

lemma *lookup-alt-def*: *lookup x* = *get* ($\lambda s.$ *case apply s x of None* ⇒ *fail* | *Some y*

⇒ *return y*)
by(*simp add*: *apply-def lookup-def*)

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*) (λ-. *m*)) *m A*

definition *eager* :: (*'v* ⇒ *int pmf*) ⇒ *'v exp* ⇒ *'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* ⇀ *int*, *'m*) *get*
  and *put* :: (*'v* ⇀ *int*, *'m*) *put*
  and *sample* :: (*int*, *'m*) *sample*
begin

lemma *comp-fun-commute-sample-var*: *comp-fun-commute* (λ*x m*. *bind* (*memo*

(*sample-var X*) *x*) (*λ-. m*))
by *unfold-locales*(*auto intro*!: *memo-commute simp add*: *fun-eq-iff sample-var-def bind-sample1 bind-sample2 return-bind sample-get*)

interpretation *sample-var*: *comp-fun-commute λx m. bind* (*memo* (*sample-var X*) *x*) (*λ-. m*)
rewrites ⋀*X m A. Finite-Set.fold* (*λx m. bind* (*memo* (*sample-var X*) *x*) (*λ-. m*)) *m A* ≡ *sample-vars X A m*
for *X*
by(*rule comp-fun-commute-sample-var*)(*simp add*: *sample-vars-def*)

lemma *comp-fun-idem-sample-var*: *comp-fun-idem* (*λx m. bind* (*memo* (*sample-var X*) *x*) (*λ-. m*))
by *unfold-locales*(*simp add*: *fun-eq-iff memo-same*)

interpretation *sample-var*: *comp-fun-idem λx m. bind* (*memo* (*sample-var X*) *x*) (*λ-. m*)
rewrites ⋀*X m A. Finite-Set.fold* (*λx m. bind* (*memo* (*sample-var X*) *x*) (*λ-. m*)) *m A* ≡ *sample-vars X A m*
for *X*
by(*rule comp-fun-idem-sample-var*)(*simp add*: *sample-vars-def*)

lemma *sample-vars-empty* [*simp*]: *sample-vars X* {} *m* = *m*
by(*simp add*: *sample-vars-def*)

lemma *sample-vars-insert*:
finite A ⟹ *sample-vars X* (*insert x A*) *m* = *bind* (*memo* (*sample-var X*) *x*) (*λ-. sample-vars X A m*)
by(*fact sample-var.fold-insert-idem*)

lemma *sample-vars-insert2*:
finite A ⟹ *sample-vars X* (*insert x A*) *m* = *sample-vars X A* (*bind* (*memo* (*sample-var X*) *x*) (*λ-. m*))
by(*fact sample-var.fold-insert-idem2*)

lemma *sample-vars-union*:
⟦ *finite A*; *finite B* ⟧ ⟹ *sample-vars X* (*A* ∪ *B*) *m* = *sample-vars X A* (*sample-vars X B m*)
by(*subst Un-commute*)(*rule sample-var.fold-set-union*)

lemma *memo-lookup*:
bind (*memo f x*) (*λi. bind* (*lookup x*) (*g i*)) = *bind* (*memo f x*) (*λi. g i i*)
apply(*simp cong del*: *option.case-cong-weak add*: *lookup-def memo-def bind-get option.case-distrib*[**where** *h*=*λx. bind x -*] *bind-assoc bind-update return-bind update-get o-def get-const*)
apply(*subst* (*3*) *get-const*[*symmetric*])
apply(*subst option.case-distrib*[**where** *h*=*get*, *symmetric*])
apply(*simp add*: *get-get case-option-apply cong*: *option.case-cong*)
done

lemma *lazy-eq-eager*:
assumes *put-fail*: $\bigwedge s.$ *put s fail = fail*
shows *lazy X e = eager X e*
proof −
note *option.case-cong* [*cong*]
have *sample-var-get*: *bind* (*sample-var X x*) (*λi. get* (*f i*)) = *get* (*λs. bind* (*sample-var X x*) (*λi. f i s*)) **for** *x f*
by(*simp add*: *sample-var-def bind-sample1 return-bind sample-get*)
have *update-fail* [*simp*]: *update f fail = fail* **for** *f*
by(*simp add*: *update-def put-fail get-const*)
have *sample-vars-fail*: *sample-vars X A fail = fail* **if** *finite A* **for** *A* **using** *that*
by *induction*(*simp-all add*: *memo-def bind-get option.case-distrib*[**where** *h=λx. bind x -*] *bind-assoc bind-update return-bind sample-var-def bind-sample1 sample-const case-option-collapse get-const cong del*: *option.case-cong-weak*)
have *sample-var-const*: *bind* (*sample-var X x*) (*λ-. m*) = *m* **for** *x m*
by(*simp add*: *sample-var-def bind-sample1 return-bind sample-const*)
have *sample-var-lookup-same*: *bind* (*memo* (*sample-var X*) *x*) (*λi. bind* (*lookup x*) (*f i*)) = *bind* (*memo* (*sample-var X*) *x*) (*λi. f i i*) **for** *x f*
by(*simp add*: *lookup-def bind-get memo-def option.case-distrib*[**where** *h=λx. bind x -*] *bind-assoc bind-update return-bind update-get sample-var-get option.case-distrib*[**where** *h=get*, *symmetric*] *get-get case-option-apply*)
have *sample-var-lookup-other*: *bind* (*memo* (*sample-var X*) *y*) (*λi. bind* (*lookup x*) (*f i*)) = *bind* (*lookup x*) (*λj. bind* (*memo* (*sample-var X*) *y*) (*λi. f i j*))
if *x ≠ y* **for** *x y f* **using** *that*
apply(*simp add*: *lookup-def memo-def bind-get option.case-distrib*[**where** *h=λx. bind x -*] *bind-assoc return-bind bind-update update-get sample-var-get fail-bind option.case-distrib*[**where** *h=get*, *symmetric*] *get-get case-option-apply*)
apply(*subst* (*13*) *get-const*[*symmetric*])
apply(*clarsimp simp add*: *option.case-distrib*[**where** *h=get*, *symmetric*] *get-get case-option-apply fun-eq-iff sample-var-const intro*!: *arg-cong*[**where** *f=get*] *split*: *option.split*)
done
have *sample-vars-lookup*: *sample-vars X V* (*bind* (*lookup x*) *f*) = *bind* (*lookup x*) (*λi. sample-vars X V* (*f i*))
if *finite V x ∉ V* **for** *V x f* **using** *that*
by(*induction*)(*auto simp add*: *sample-var-lookup-other bind-return*)

have *lazy-sample-vars*: *sample-vars X V* (*bind* (*lazy X e*) *f*) = *bind* (*lazy X e*) (*λi. sample-vars X V* (*f i*))
if *finite V* **for** *f e V* **using** *that* **unfolding** *lazy-def*
proof(*induction e arbitrary*: *f*)
case (*Var x*)
have *bind* (*memo* (*sample-var X*) *x*) (*λi. sample-vars X V* (*f i*)) = *sample-vars X V* (*bind* (*memo* (*sample-var X*) *x*) *f*) (**is** *?lhs V = ?rhs V*)
using *Var*
proof(*cases x ∈ V*)
{ **fix** *V*
assume *False*: *x ∉ V* **and** *fin*: *finite V*

have *?lhs V* = *bind* (*memo* (*sample-var X*) *x*) (λ-. *bind* (*lookup x*) (λ*i*. *sample-vars X V* (*f i*)))
by(*simp add*: *sample-var-lookup-same*)
also have ... = *bind* (*memo* (*sample-var X*) *x*) (λ-. *sample-vars X V* (*bind* (*lookup x*) *f*))
using *fin False* **by**(*simp add*: *sample-vars-lookup*)
also have ... = *sample-vars X* (*insert x V*) (*bind* (*lookup x*) *f*) **using** *fin*
by(*simp add*: *sample-vars-insert*)
also have ... = *sample-vars X V* (*bind* (*memo* (*sample-var X*) *x*) (λ-. *bind* (*lookup x*) *f*)) **using** *fin*
by(*simp only*: *sample-vars-insert2*)
also have ... = *?rhs V*
by(*simp add*: *sample-var-lookup-same*)
finally show *?lhs V* = *?rhs V* **. }**
note *False* = *this*

case *True*
hence *V*: *V* = *insert x* (*V* − {*x*}) **by** *auto*
have *?lhs V* = *bind* (*memo* (*sample-var X*) *x*) (λ*i*. *bind* (*memo* (*sample-var X*) *x*) (λ-. *sample-vars X* (*V* − {*x*}) (*f i*)))
using *Var* **by**(*subst V*)(*simp add*: *sample-vars-insert del*: *Diff-insert0 insert-Diff-single*)
also have ... = *bind* (*memo* (*sample-var X*) *x*) (λ-. *bind* (*memo* (*sample-var X*) *x*) (λ*i*. *sample-vars X* (*V* − {*x*}) (*f i*)))
by(*simp add*: *memo-same*)
also have ... = *bind* (*memo* (*sample-var X*) *x*) (λ-. *sample-vars X* (*V* − {*x*}) (*bind* (*memo* (*sample-var X*) *x*) *f*))
using *Var* **by**(*subst False*)(*simp-all*)
also have ... = *?rhs V* **using** *Var*
by(*rewrite* **in** - = ⌑ *V*)(*simp add*: *sample-vars-insert del*: *Diff-insert0 insert-Diff-single*)
finally show *?thesis* **.**
qed
then show *?case* **by** *simp*
next
case (*Const x*)
then show *?case* **by**(*simp add*: *return-bind*)
next
case (*Plus e1 e2*)
then show *?case*
by(*simp add*: *bind-assoc return-bind*)
next
case (*Div e1 e2*)
then show *?case*
apply(*simp add*: *bind-assoc if-distrib*[**where** *f*=λ*x*. *bind x* -] *fail-bind return-bind cong del*: *if-weak-cong*)
apply(*subst* (*6*) *sample-vars-fail*[*OF* ‹*finite V*›, *symmetric*])
apply(*simp add*: *if-distrib*[**where** *f*=*sample-vars* - -, *symmetric*])
done

qed

define *V* **where** $V \equiv$ *vars e*
then have *vars e* $\subseteq$ *V finite V* **by** *simp-all*
then have *sample-vars X V* (*bind* (*eval lookup e*) *f*) = *sample-vars X V* (*bind* (*lazy X e*) *f*) **for** *f*
unfolding *lazy-def*
proof(*induction e arbitrary*: *f*)
case (*Var x*)
then have *V*: *V* = *insert x* (*V* − {*x*}) **by** *auto*
show *?case* **using** *Var*
apply(*subst* (*1 2*) *V*)
apply(*subst* (*1 2*) *sample-vars-insert2*)
apply(*simp-all add*: *memo-same memo-lookup*)
done
qed(*simp-all add*: *bind-assoc lazy-sample-vars*[*unfolded lazy-def*])
note *this*[*of return*, *unfolded V-def*]
also have *sample-vars X* (*vars e*) (*bind* (*lazy X e*) *f*) = *bind* (*lazy X e*) *f* **for** *f*
unfolding *lazy-def*
proof(*induction e arbitrary*: *f*)
{ case *Var* **show** *?case* **by**(*simp add*: *memo-same bind-return*) **}**
{ case *Const* **show** *?case* **by**(*simp add*: *bind-return*) **}**
{ case *Plus* **show** *?case*
by(*simp add*: *bind-assoc sample-vars-union lazy-sample-vars*[*unfolded lazy-def*] *Plus.IH*) **}**
{ case *Div* **show** *?case*
by(*simp add*: *bind-assoc sample-vars-union lazy-sample-vars*[*unfolded lazy-def*] *Div.IH*) **}**
qed
finally show *?thesis* **by**(*simp add*: *bind-return V-def eager-def*)
qed

end

interpretation *F*: *exp-base*
return-option return-id
bind-option return-id bind-id
fail-option return-id
.

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*
.

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
..

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*
..

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*)
..

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* (*λs. case s x of None* ⇒ *fail* | *Some y* ⇒ *return y*)

lemma *lookup-alt-def*:
lookup x = *ask* (*λs. case apply s x of None* ⇒ *fail* | *Some y* ⇒ *return y*)
by(*simp add*: *lookup-def apply-def*)

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*) (λ*j*. *bind* (*eval E e1*) (λ*i*. *return* (*i* + *j*)))
eval E (*Div e1 e2*) = *bind* (*eval E e2*) (λ*j*. *bind* (*eval E e1*) (λ*i*. *if j* = *0 then fail else return* (*i div j*)))
by(*simp*; *rule bind-commute*)+

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*
.

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*)
unfolding *SFP.lookup-def SFI.lookup-def* **by** *transfer-prover*

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
unfolding *RFI.lookup-alt-def SFI.lookup-alt-def* **by** *transfer-prover*

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*)
by *transfer-prover*

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
unfolding *RFI.lookup-alt-def SFP.lookup-alt-def* **by** *transfer-prover*

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*)
by *transfer-prover*

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*)
by(*rule cr-envT-stateT-eval*[*of E*, *THEN rel-funD*, *OF refl*, *unfolded eq-alt*, *unfolded cr-prod1-Grp option.rel-Grp cr-id-prob-Grp rel-optionT-Grp*, *simplified*, *THEN cr-envT-stateTD*, *unfolded BNF-Def.Grp-def*, *THEN conjunct1*])

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* **.**

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* **by**(*rule monad-fail*)

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*
..

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*
..

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* **.**

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*
..

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*
by(*simp add*: *bind-return*)

lemma *repeat-add*: *repeat* (*n* + *m*) *mx* = *bind* (*repeat n mx*) (λ-. *repeat m mx*)
by(*induction n*)(*simp-all add*: *return-bind bind-assoc*)

lemma *hanoi-correct*: *hanoi n* = *repeat* (*2* ^ *n* − *1*) (*tick* (*return* ()))
proof(*induction n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*)
have *hanoi* (*Suc n*) = *repeat* ((*2* ^ *n* − *1*) + *1* + (*2* ^ *n* − *1*)) (*tick* (*return* ()))
by(*simp only*: *hanoi.simps repeat-add repeat-1 Suc.IH bind-assoc bind-tick return-bind*)
also have (*2* ^ *n* − *1*) + *1* + (*2* ^ *n* − *1*) = (*2* ^ *Suc n* − *1* :: *nat*) **by** *simp*
finally show *?case* **.**
qed

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*))
by(*induction xs*)(*simp-all add*: *fail-bind return-bind*)

lemma *fastprod-correct*: *fastprod xs* = *return* (*prod-list xs*)
by(*simp add*: *fastprod-def work-alt-def catch-fail catch-return prod-list-zero-iff* [*symmetric*])

end

end