

# Axiomatic Constructor Classes in HOL

Brian Huffman

February 7, 2006

## Contents

<b>1</b>	<b>Finite representable types</b>	<b>3</b>
<b>2</b>	<b>Idempotents</b>	<b>4</b>
2.1	Definition and basic properties . . . . .	4
2.2	Composing idempotents . . . . .	5
2.3	A type of idempotents . . . . .	5
2.4	Operations on idempotents . . . . .	5
2.4.1	Casting values into an idempotent . . . . .	5
2.4.2	Ordering of idempotents . . . . .	6
2.4.3	Membership in a idempotent . . . . .	6
2.4.4	The identity idempotent . . . . .	7
<b>3</b>	<b>Representable Types</b>	<b>7</b>
3.1	A HOL type universe . . . . .	8
3.2	Class of representable types . . . . .	8
3.3	Instances of class <i>rep</i> . . . . .	8
3.3.1	Universal Domain . . . . .	8
3.3.2	Representable base types . . . . .	8
3.3.3	Representable type constructors . . . . .	9
3.4	Representations of types: <i>rep-of</i> . . . . .	10
3.5	Making <i>rep</i> the default class . . . . .	11
<b>4</b>	<b>Coercion Operators</b>	<b>11</b>
<b>5</b>	<b>Type Application</b>	<b>12</b>
5.1	Type of monotone type constructor functions . . . . .	12
5.2	Class of type constructors . . . . .	12
5.3	Type constructor for type application . . . . .	13

<b>6</b>	<b>Functor Class</b>	<b>13</b>
6.1	Class axioms . . . . .	14
6.2	<i>fmap</i> . . . . .	14
6.3	Proving that <i>fmap coerce = coerce</i> . . . . .	15
6.4	Functor locale . . . . .	16
<b>7</b>	<b>Monad Class</b>	<b>19</b>
7.1	locale for proving stuff about monad representations . . . . .	22
<b>8</b>	<b>Functor and Monad Examples</b>	<b>26</b>
8.1	Lists . . . . .	26
8.1.1	<i>rep</i> instance . . . . .	26
8.1.2	<i>functor</i> instance . . . . .	27
8.1.3	<i>monad</i> instance . . . . .	27
8.2	Option type . . . . .	28
8.2.1	<i>rep</i> instance . . . . .	28
8.2.2	<i>functor</i> instance . . . . .	29
8.2.3	<i>monad</i> instance . . . . .	29
8.3	Trivial type constructor . . . . .	30
8.3.1	<i>functor</i> instance . . . . .	30
8.3.2	<i>monad</i> instance . . . . .	30
<b>9</b>	<b>State monad transformer</b>	<b>31</b>
9.1	Type definition and monad operations . . . . .	31
9.2	<i>rep</i> instance . . . . .	32
9.3	<i>functor</i> instance . . . . .	33
9.4	StateT is isomorphic to stateT . . . . .	33
9.5	<i>monad</i> instance . . . . .	34
9.6	Other functions . . . . .	35

# 1 Finite representable types

```
theory FinRep  
imports Main  
begin
```

A type is finitely representable if we can define a list containing all of its elements.

```
consts
```

```
  LIST :: 'a list
```

```
axclass finrep  $\subseteq$  type
```

```
  set-LIST [simp]: set LIST = UNIV
```

```
instance finrep  $\subseteq$  finite
```

```
apply (intro-classes)
```

```
apply (subst set-LIST [symmetric])
```

```
apply (rule finite-set)
```

```
done
```

```
defs (overloaded)
```

```
  LIST-unit-def: LIST  $\equiv$  [()]
```

```
  LIST-bool-def: LIST  $\equiv$  [True, False]
```

```
  LIST-sum-def: LIST  $\equiv$  map Inl LIST @ map Inr LIST
```

```
  LIST-opt-def: LIST  $\equiv$  None # map Some LIST
```

```
  LIST-prod-def: LIST  $\equiv$  concat (map ( $\lambda a.$  map (Pair a) LIST) LIST)
```

```
  LIST-set-def: LIST  $\equiv$  foldr ( $\lambda x$  As. map (insert x) As @ As) LIST [{}]
```

```
  LIST-fun-def:
```

```
  LIST  $\equiv$  foldr ( $\lambda x$  fs. concat (map ( $\lambda y.$  map ( $\lambda f.$  f(x:=y)) fs) LIST))
```

```
  LIST [arbitrary]
```

```
instance unit :: finrep
```

```
by (intro-classes, unfold LIST-unit-def, auto)
```

```
instance bool :: finrep
```

```
by (intro-classes, unfold LIST-bool-def, auto)
```

```
instance + :: (finrep, finrep) finrep
```

```
apply (intro-classes, unfold LIST-sum-def)
```

```
apply (auto, case-tac x, simp-all)
```

```
done
```

```
instance option :: (finrep) finrep
```

```
apply (intro-classes, unfold LIST-opt-def)
```

```
apply (auto, case-tac x, simp-all)
```

```
done
```

```
instance * :: (finrep, finrep) finrep
```

```
by (intro-classes, unfold LIST-prod-def, auto)
```

```

instance set :: (finrep) finrep
apply (intro-classes, auto, rename-tac A)
apply (subgoal-tac A  $\cap$  set LIST  $\in$  set LIST, simp)
apply (unfold LIST-set-def)
apply (rule-tac x=LIST::'a list in spec, clarify, rename-tac xs)
apply (induct-tac xs)
apply simp
apply (simp add: Int-insert-right)
done

```

```

instance fun :: (finrep, finrep) finrep
apply (intro-classes, auto, rename-tac f)
apply (subgoal-tac  $\exists g \in \text{set LIST}. \forall x \in \text{set LIST}. f x = g x$ )
apply (simp add: expand-fun-eq [symmetric])
apply (unfold LIST-fun-def)
apply (rule-tac x=LIST::'a list in spec, clarify, rename-tac xs)
apply (induct-tac xs)
apply simp
apply clarify
apply (rule-tac x=g(a:=f a) in be1)
apply simp
apply (simp del: fun-upd-apply)
apply (rule-tac x=f a in ex1)
apply simp
done

```

end

## 2 Idempotents

```

theory Idempotent
imports Main
begin

```

### 2.1 Definition and basic properties

```

constdefs
  idempotent :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  bool
  idempotent p  $\equiv \forall x. p (p x) = p x$ 

```

```

lemma idempotent: idempotent p  $\implies p (p x) = p x$ 
by (simp add: idempotent-def)

```

```

lemma idempotent-id: idempotent id
by (simp add: idempotent-def)

```

```

lemma idempotent-const: idempotent ( $\lambda x. c$ )

```

**by** (*simp add: idempotent-def*)

**lemma** *idempotent-functor*:

**assumes**  $F2: \bigwedge f g xs. m f (m g xs) = m (\lambda x. f (g x)) xs$

**shows**  $idempotent\ p \implies idempotent\ (m\ p)$

**by** (*simp add: idempotent-def F2*)

The composition of two idempotents is equal to the lesser of the two (if they are comparable).

**lemma** *idempotent-less-comp1*:

$\llbracket idempotent\ f; idempotent\ g; range\ f \subseteq range\ g \rrbracket \implies f (g\ x) = f\ x$

**oops**

**lemma** *idempotent-less-comp2*:

$\llbracket idempotent\ f; idempotent\ g; range\ f \subseteq range\ g \rrbracket \implies g (f\ x) = f\ x$

**apply** (*subgoal-tac f x ∈ range g*)

**apply** *clarify*

**apply** (*simp add: idempotent-def*)

**apply** (*erule subsetD*)

**apply** *simp*

**done**

## 2.2 Composing idempotents

**lemma** *idempotent-fun*:

$\llbracket idempotent\ p1; idempotent\ p2 \rrbracket$

$\implies idempotent\ (\lambda f. p2 \circ f \circ p1)$

**by** (*simp add: idempotent-def expand-fun-eq*)

**lemma** *idempotent-prod*:

$\llbracket idempotent\ p1; idempotent\ p2 \rrbracket$

$\implies idempotent\ (\lambda(x,y). (p1\ x, p2\ y))$

**by** (*simp add: idempotent-def*)

**lemma** *idempotent-sum*:

$\llbracket idempotent\ p1; idempotent\ p2 \rrbracket$

$\implies idempotent\ (sum\ case\ (Inl \circ p1)\ (Inr \circ p2))$

**by** (*simp add: idempotent-def split: sum.split*)

## 2.3 A type of idempotents

**typedef** (**open**)  $'a\ idem = \{p::'a \Rightarrow 'a. idempotent\ p\}$

**by** (*rule exI, auto intro: idempotent-id*)

## 2.4 Operations on idempotents

### 2.4.1 Casting values into an idempotent

**constdefs**

$cast :: 'a\ idem \Rightarrow 'a \Rightarrow 'a$

$cast \equiv Rep-idem$

**lemma** *idempotent-cast*:  $idempotent (cast A)$   
**by** (*simp add: cast-def Rep-idem [simplified]*)

**lemma** *Abs-idem-cast*:  $Abs-idem (cast A) = A$   
**by** (*simp add: cast-def Rep-idem-inverse*)

**lemma** *cast-Abs-idem*:  $idempotent f \implies cast (Abs-idem f) = f$   
**by** (*simp add: cast-def Abs-idem-inverse*)

**lemma** *cast-idem* [*simp*]:  $cast A (cast A x) = cast A x$   
**by** (*simp add: idempotent-cast idempotent*)

**lemma** *cast-idem2* [*simp*]:  $cast A \circ cast A = cast A$   
**by** (*rule ext, simp*)

**lemma** *idempotent-equality*:  $cast A = cast B \implies A = B$   
**by** (*simp add: cast-def Rep-idem-inject*)

## 2.4.2 Ordering of idempotents

**instance** *idem* :: (*type*) *ord* ..

**defs** (**overloaded**)

*le-idem-def*:

$A \leq B \equiv range (cast A) \subseteq range (cast B)$

**lemma** *idem-le-refl* [*simp, intro!*]:  $(A::'a idem) \leq A$   
**by** (*simp add: le-idem-def*)

## 2.4.3 Membership in a idempotent

**constdefs**

*in-idem* ::  $'a \Rightarrow 'a idem \Rightarrow bool$  (**infixl** ::: 50)

$x$  ::  $A \equiv cast A x = x$

**lemma** *in-idemI*:  $cast A x = x \implies x$  ::  $A$   
**by** (*simp add: in-idem-def*)

**lemma** *in-idemD*:  $x$  ::  $A \implies \exists y. x = cast A y$   
**by** (*unfold in-idem-def, rule exI, erule sym*)

**lemma** *in-idemE*:  $\llbracket x$  ::  $A; \bigwedge y. x = cast A y \implies R \rrbracket \implies R$   
**by** (*drule in-idemD, auto*)

**lemma** *cast-in-idem* [*simp*]:  $cast A x$  ::  $A$   
**by** (*simp add: in-idemI*)

**lemma** *cast-fixed*:  $x$  ::  $A \implies cast A x = x$

```

by (simp only: in-idem-def)

lemma subidemD:  $\llbracket A \leq B; x \text{ :: } A \rrbracket \implies x \text{ :: } B$ 
  apply (unfold in-idem-def)
  apply (unfold le-idem-def)
  apply (subgoal-tac  $x \in \text{range (cast B)}$ )
  apply clarsimp
  apply (erule subsetD)
  apply (erule subst)
  apply simp
done

lemma rev-subidemD:  $\llbracket x \text{ :: } A; A \leq B \rrbracket \implies x \text{ :: } B$ 
by (rule subidemD)

lemma subidemI:  $\llbracket \bigwedge x. x \text{ :: } A \implies x \text{ :: } B \rrbracket \implies A \leq B$ 
  apply atomize
  apply (simp add: le-idem-def)
  apply (rule subsetI)
  apply (drule-tac  $x=x$  in spec)
  apply clarsimp
  apply (simp add: in-idem-def)
  apply (erule subst, simp)
done

lemma nonempty-idem:  $\exists x. x \text{ :: } A$ 
  apply (unfold in-idem-def)
  apply (rule-tac  $x=\text{cast } A \text{ } x$  in exI)
  apply simp
done

2.4.4 The identity idempotent

lemma cast-id-idem [simp]:  $\text{cast (Abs-idem id)} = \text{id}$ 
by (simp add: cast-Abs-idem idempotent-id)

lemma in-idem-id-idem [simp, intro]:  $x \text{ :: } \text{Abs-idem id}$ 
by (simp add: in-idemI)

lemma less-id-idemp [simp, intro]:  $A \leq \text{Abs-idem id}$ 
by (simp add: subidemI)

end

```

### 3 Representable Types

```

theory Representable
imports FinRep Idempotent

```

**begin**

### 3.1 A HOL type universe

**datatype**  $U = Utag\ nat\ U\ list$

### 3.2 Class of representable types

Overloaded embedding and projection functions between a representable type and the universal domain.

**axclass**  $rep-consts < type$

**consts**

$emb :: 'a::rep-consts \Rightarrow U$

$proj :: U \Rightarrow 'a::rep-consts$

**axclass**  $rep < rep-consts$

$emb-inverse [simp]: proj (emb\ x) = x$

**lemma**  $inj-emb: inj (emb::'a::rep \Rightarrow U)$

**by** ( $rule\ inj-on-inverseI, rule\ emb-inverse$ )

**lemmas**  $emb-inject [simp] = inj-eq [OF\ inj-emb, standard]$

### 3.3 Instances of class $rep$

#### 3.3.1 Universal Domain

The Universe itself is trivially representable.

**instance**  $U :: rep-consts ..$

**defs** (**overloaded**)

$emb-U-def [simp]: emb \equiv id :: U \Rightarrow U$

$proj-U-def [simp]: proj \equiv id :: U \Rightarrow U$

**instance**  $U :: rep$

**by** ( $intro-classes, simp$ )

#### 3.3.2 Representable base types

The unit type is representable.

**instance**  $unit :: rep-consts ..$

**defs** (**overloaded**)

$emb-unit-def: emb \equiv \lambda x::unit. Utag\ 0\ []$

$proj-unit-def: proj \equiv \lambda u. ()$

**instance**  $unit :: rep$

**by** ( $intro-classes, simp$ )

The booleans are representable

```
instance bool :: rep-consts ..
defs (overloaded)
  emb-bool-def: emb ≡ λb. if b then Utag 0 [] else Utag 1 []
  proj-bool-def: proj ≡ λu. u = Utag 0 []
```

```
instance bool :: rep
by (intro-classes, simp add: emb-bool-def proj-bool-def)
```

The naturals are representable

```
instance nat :: rep-consts ..
defs (overloaded)
  emb-nat-def: emb ≡ λn. Utag n []
  proj-nat-def: proj ≡ λu. case u of Utag n xs ⇒ n
```

```
instance nat :: rep
by (intro-classes, simp add: emb-nat-def proj-nat-def)
```

### 3.3.3 Representable type constructors

Products of representable types are representable.

```
instance * :: (rep, rep) rep-consts ..
defs (overloaded)
  emb-prod-def:
    emb ≡ λx. Utag 0 [emb (fst x), emb (snd x)]
  proj-prod-def:
    proj ≡ λu. case u of Utag n xs ⇒ (proj (xs!0), proj (xs!1))
```

```
instance * :: (rep, rep) rep
by (intro-classes, simp add: emb-prod-def proj-prod-def)
```

Sums of representable types are representable.

```
instance + :: (rep, rep) rep-consts ..
defs (overloaded)
  emb-sum-def: emb ≡ λx. case x of
    Inl a ⇒ Utag 0 [emb a] |
    Inr b ⇒ Utag 1 [emb b]
  proj-sum-def: proj ≡ λu. case u of Utag n xs ⇒
    if n = 0 then Inl (proj (xs!0)) else Inr (proj (xs!0))
```

```
instance + :: (rep, rep) rep
by (intro-classes, unfold emb-sum-def proj-sum-def, simp split: sum.split)
```

Functions from finite to representable types are representable.

```

instance fun :: (finrep, rep) rep-consts ..
defs (overloaded)
  emb-fun-def: emb  $\equiv \lambda f. Utag\ 0\ (map\ (emb\ \circ\ f)\ LIST)$ 
  proj-fun-def: proj  $\equiv \lambda u. case\ u\ of\ Utag\ n\ xs \Rightarrow$ 
    ( $\lambda a. proj\ (xs\ !\ (LEAST\ n.\ LIST\ !\ n = a))$ )

instance fun :: (finrep, rep) rep
apply (intro-classes, simp add: emb-fun-def proj-fun-def)
apply (rename-tac f, rule ext)
apply (rule-tac x=a and xs=LIST in all-nth-imp-all-set [rule-format])
apply (subst nth-map)
apply (erule order-le-less-trans [OF Least-le [OF refl]])
apply simp-all
apply (rule-tac f=f in arg-cong)
apply (rule LeastI [OF refl])
done

```

### 3.4 Representations of types: rep-of

```

constdefs
  rep-of :: 'a::rep itself  $\Rightarrow U\ idem$ 
  rep-of (a::'a::rep itself)  $\equiv Abs-idem\ (\lambda u. emb\ (proj\ u::'a))$ 

lemma emb-proj: emb (proj x::'a::rep) = cast (rep-of TYPE('a)) x
apply (unfold rep-of-def cast-def)
apply (simp add: Abs-idem-inverse idempotent-def)
done

lemma cast-rep-of:
  cast (rep-of TYPE('a::rep)) = ( $\lambda u. emb\ (proj\ u::'a)$ )
by (rule ext, simp add: emb-proj)

lemma proj-inverse [simp]:
  x :: rep-of TYPE('a::rep)  $\Longrightarrow emb\ (proj\ x::'a) = x$ 
by (simp add: emb-proj cast-fixed)

lemma emb-in-rep-of: emb (x::'a::rep) :: rep-of TYPE('a)
by (rule in-idemI, simp add: cast-rep-of)

lemma emb-in-rep-of-le [simp]:
  rep-of TYPE('a)  $\leq rep-of\ B \Longrightarrow emb\ (x::'a::rep) :: rep-of\ B$ 
by (erule subidemD, rule in-idemI, simp add: cast-rep-of)

  rep-of TYPE(U)

lemma rep-of-U: rep-of TYPE(U) = Abs-idem id
by (simp add: rep-of-def id-def)

lemma less-rep-of-U [simp, intro!]: A  $\leq rep-of\ TYPE(U)$ 

```

**by** (*simp add: rep-of-U*)

**lemma** *in-rep-of-U* [*simp, intro!*]:  $x :: \text{rep-of } \text{TYPE}(U)$   
**by** (*simp add: rep-of-U*)

**lemma** *cast-rep-of-U*:  $\text{cast } (\text{rep-of } \text{TYPE}(U)) = \text{id}$   
**by** (*rule ext, simp add: cast-rep-of*)

### 3.5 Making *rep* the default class

From now on, free type variables are assumed to be in class *rep*, unless specified otherwise.

**defaultsort** *rep*

**end**

## 4 Coercion Operators

**theory** *Coerce*  
**imports** *Representable*  
**begin**

**constdefs**  
*coerce* ::  $'a \Rightarrow 'b$   
*coerce*  $x \equiv \text{proj } (\text{emb } x)$

**lemma** *proj-emb*:  $\text{proj } (\text{emb } x) = \text{coerce } x$   
**by** (*simp add: coerce-def*)

Trivial instances of *coerce*

**lemma** *coerce-eq-id* [*simp*]:  $(\text{coerce}::'a \Rightarrow 'a) = \text{id}$   
**by** (*rule ext, simp add: coerce-def*)

**lemma** *coerce-eq-emb* [*simp*]:  $(\text{coerce}::'a \Rightarrow U) = \text{emb}$   
**by** (*rule ext, simp add: coerce-def*)

**lemma** *coerce-eq-proj* [*simp*]:  $(\text{coerce}::U \Rightarrow 'a) = \text{proj}$   
**by** (*rule ext, simp add: coerce-def*)

Cancellation rules

**lemma** *emb-coerce* [*simp*]:  
 $\text{emb } x :: \text{rep-of } \text{TYPE}('a) \Longrightarrow \text{emb } ((\text{coerce } x)::'a) = \text{emb } x$   
**by** (*simp add: coerce-def*)

**lemma** *emb-coerce2*:  
 $\text{rep-of } \text{TYPE}('a) \leq \text{rep-of } \text{TYPE}('b)$   
 $\Longrightarrow \text{emb } ((\text{coerce}::'a \Rightarrow 'b) x) = \text{emb } x$

```

by (simp add: coerce-def)

lemma coerce-idem [simp]:
  rep-of TYPE('a) ≤ rep-of TYPE('b)
  ⇒ coerce ((coerce::'a ⇒ 'b) x) = coerce x
by (simp add: coerce-def)

lemma coerce-inverse [simp]:
  emb (x::'a) :: rep-of TYPE('b) ⇒ coerce (coerce x :: 'b) = x
by (simp only: coerce-def proj-inverse emb-inverse)

lemma coerce-type:
  rep-of TYPE('a) ≤ rep-of TYPE('b)
  ⇒ emb ((coerce::'a ⇒ 'b) x) :: rep-of TYPE('a)
by (simp add: coerce-def)

lemma inj-coerce:
  rep-of TYPE('a) ≤ rep-of TYPE('b)
  ⇒ inj (coerce::'a ⇒ 'b)
apply (rule inj-on-inverseI)
apply (rule coerce-inverse)
apply simp
done

end

```

## 5 Type Application

```

theory TypeApp
imports Representable
begin

```

### 5.1 Type of monotone type constructor functions

```

typedef (open) monotc = {f::U idem ⇒ U idem. mono f}
by (rule-tac x=id in exI, simp add: mono-def)

```

### 5.2 Class of type constructors

```

axclass tycon < type
consts monotc :: 'f::tycon itself ⇒ monotc

constdefs
  tc :: 'f::tycon itself ⇒ U idem ⇒ U idem
  tc t ≡ Rep-monotc (monotc t)

lemma mono-tc: mono (tc t)
by (unfold tc-def, rule Rep-monotc [simplified])

```

### 5.3 Type constructor for type application

The representatives of a type constructor  $'f$  applied to representable type  $'a$ .

```
typedef (open) ('a,'f)App (infixl $ 65)
  = {x. x :: tc TYPE('f::tycon) (rep-of TYPE('a::rep))}
by (simp, rule nonempty-idem)
```

```
syntax (xsymbols)
  App :: [type, type]  $\Rightarrow$  type (( $\dots$ ) [999,1000] 999)
```

```
declare Rep-App-inverse[simp] Rep-App-inject[simp]
  Abs-App-inverse[simp] Abs-App-inject[simp]
  Abs-App-inject[THEN iffD2,intro!]
```

The result of domain application is representable.

```
instance App :: (rep, tycon) rep-consts ..
defs (overloaded)
  emb-App-def: emb  $\equiv$  Rep-App
  proj-App-def:
    proj::U  $\Rightarrow$  'a::rep.'f::tycon  $\equiv$ 
       $\lambda u.$  Abs-App (cast (tc TYPE('f) (rep-of TYPE('a))) u)
```

```
instance App :: (rep, tycon) rep
apply (intro-classes, unfold emb-App-def proj-App-def)
apply (subst cast-fixed)
apply (rule Rep-App [simplified])
apply (rule Rep-App-inverse)
done
```

```
lemma rep-of-App:
  rep-of TYPE('a::rep.'m::tycon) = tc TYPE('m) (rep-of TYPE('a))
by (simp add: rep-of-def emb-App-def proj-App-def Abs-idem-cast
  del: proj-inverse)
```

```
lemma rep-of-App-mono [simp, intro]:
  rep-of TYPE('a)  $\leq$  rep-of TYPE('b)
   $\implies$  rep-of TYPE('a.'m::tycon)  $\leq$  rep-of TYPE('b.'m)
apply (simp add: rep-of-App)
apply (erule mono-tc [THEN monoD])
done
```

**end**

## 6 Functor Class

```
theory FunctorClass
```

```

imports Coerce TypeApp
begin

```

## 6.1 Class axioms

**consts**

```

  rep-fmap :: (U ⇒ U) ⇒ U·f ⇒ U·f::tycon

```

**axclass** functor < tycon

rep-fmap-type:

```

  [[ $\bigwedge x. x :: A \implies f x :: B$ ; emb (xs::U·f::tycon) :: tc TYPE('f) A]]
   $\implies$  emb (rep-fmap f xs) :: tc TYPE('f) B

```

rep-fmap-cast:

```

  rep-fmap (cast A) (xs::U·f::tycon) =
  proj (cast (tc TYPE('f) A) (emb xs))

```

rep-fmap-fmap:

```

  rep-fmap f (rep-fmap g xs) = rep-fmap ( $\lambda x. f (g x)$ ) xs

```

## 6.2 fmap

**constdefs**

```

  fmap :: ('a ⇒ 'b) ⇒ 'a·f ⇒ 'b·f::functor

```

```

  fmap ≡  $\lambda f xs. coerce (rep-fmap (emb \circ f \circ proj)) (coerce xs::U·f)$ 

```

**lemmas** tc-rep-of = rep-of-App[symmetric]

**lemmas** rep-fmap-type2 =

rep-fmap-type

```

  [of rep-of TYPE('b) - rep-of TYPE('c),
  simplified tc-rep-of, standard]

```

**theorem** fmap-id [simp]: fmap id xs = xs

**apply** (simp add: fmap-def o-def)

**apply** (simp add: emb-proj)

**apply** (subst rep-fmap-cast)

**apply** (subst rep-of-App [symmetric])

**apply** (simp add: coerce-def cast-rep-of)

**done**

**lemmas** fmap-ident [simp] = fmap-id [unfolded id-def]

**theorem** fmap-fmap:

```

  fmap f (fmap g xs) = fmap ( $\lambda x. f (g x)$ ) xs

```

**apply** (simp add: fmap-def)

**apply** (subst coerce-inverse)

**apply** (rule rep-fmap-type2, simp)

**apply** (rule coerce-type, simp)

**apply** (*simp add: rep-fmap-fmap o-def*)  
**done**

**lemma** *fmap-comp*:  $fmap (f \circ g) = fmap f \circ fmap g$   
**by** (*simp add: o-def fmap-fmap*)

**lemma** *inj-fmap*:  $inj f \implies inj (fmap f)$   
**apply** (*rule-tac g=fmap (inv f) in inj-on-inverseI*)  
**apply** (*simp add: fmap-fmap*)  
**done**

### 6.3 Proving that $fmap\ coerce = coerce$

**lemma** *rep-fmap-eq-fmap*:  $rep-fmap = fmap$   
**by** (*simp add: fmap-def*)

**lemma** *in-rep-of-App-U*:  
 $xs :: tc\ TYPE('f::tycon)\ A \implies xs :: rep-of\ TYPE(U\cdot'f)$   
**apply** (*erule rev-subidemD*)  
**apply** (*simp add: rep-of-App*)  
**apply** (*rule monoD [OF mono-tc]*)  
**apply** (*rule less-rep-of-U*)  
**done**

**lemma** *rep-fmap-cast-rep-of*:  
 $rep-fmap (cast (rep-of\ TYPE('a))) =$   
 $proj \circ cast (rep-of\ TYPE('a\cdot'f::functor)) \circ (emb::U\cdot'f \Rightarrow U)$   
**by** (*rule ext, simp add: rep-fmap-cast rep-of-App*)

**lemma** *rep-fmap-comp*:  
 $(rep-fmap (f \circ g) :: U\cdot'f::functor \Rightarrow U\cdot'f) = rep-fmap f \circ rep-fmap g$   
**by** (*simp add: rep-fmap-eq-fmap fmap-comp*)

**lemma** *coerce-coerce*:  
 $(emb \circ (coerce::'a \Rightarrow 'b)) \circ proj :: U \Rightarrow U$   
 $= cast (rep-of\ TYPE('b)) \circ cast (rep-of\ TYPE('a))$   
**apply** (*rule ext, simp*)  
**apply** (*simp only: coerce-def emb-proj*)  
**done**

**lemma** *fmap-coerce*:  $fmap\ coerce = (coerce::'a\cdot'f \Rightarrow 'b\cdot'f::functor)$   
**apply** (*rule ext*)  
**apply** (*simp add: fmap-def*)  
**apply** (*subst coerce-coerce*)  
**apply** (*subst rep-fmap-comp*)  
**apply** (*simp add: rep-fmap-cast-rep-of*)  
**apply** (*simp add: cast-rep-of*)  
**apply** (*simp add: coerce-def*)  
**done**

## 6.4 Functor locale

It is easy to prove instances of the functor class: All you need to do is define *tc* and *rep-fmap* in a standard way, and then prove that the functor laws hold at one specific type.

**constdefs**

```

functor-tc :: ((U ⇒ U) ⇒ 'l ⇒ 'l) ⇒ monotc
functor-tc umap ≡ Abs-monotc (λA. Abs-idem (emb ∘ umap (cast A) ∘ proj))

```

```

rep-fmap-of :: ((U ⇒ U) ⇒ 'l ⇒ 'l) ⇒ ((U ⇒ U) ⇒ U·'f ⇒ U·'f::tycon)
rep-fmap-of umap ≡ λf. coerce ∘ umap f ∘ coerce

```

**locale** *functor-locale* =

```

fixes umap :: (U ⇒ U) ⇒ 'l ⇒ 'l
assumes tc-umap: monotc TYPE('f::tycon) ≡ functor-tc umap
and rep-fmap: rep-fmap :: (U ⇒ U) ⇒ U·'f ⇒ U·'f::tycon
                ≡ rep-fmap-of umap
and umap-id: λxs. umap id xs = xs
and umap-umap: λf g xs. umap f (umap g xs) = umap (λx. f (g x)) xs

```

**lemma** (in *functor-locale*) *cast-tc*:

```

cast (tc TYPE('f::tycon) A) = emb ∘ umap (cast A) ∘ proj
apply (unfold tc-def tc-umap functor-tc-def)
apply (subst Abs-monotc-inverse)
apply (simp)
apply (rule monoI)
apply (rule subidemI)
apply (simp add: in-idem-def)
apply (simp add: cast-Abs-idem idempotent-def umap-umap)
apply (erule subst)
apply (simp add: umap-umap)
apply (subst cast-fixed)
apply (erule subidemD)
apply (rule cast-in-idem)
apply (rule refl)
apply (subst cast-Abs-idem)
apply (simp add: idempotent-def umap-umap)
apply (rule refl)
done

```

**lemma** (in *functor-locale*) *rep-of-App-U-functor* [*simp*]:

```

rep-of TYPE('l) = rep-of TYPE(U·'f::tycon)
apply (rule idempotent-equality, rule ext)
apply (simp add: rep-of-App)
apply (simp add: cast-tc cast-rep-of)
apply (simp add: umap-id [unfolded id-def])
done

```

**lemma** (in *functor-locale*) *in-tc-iff*:

$(x :: tc \text{TYPE}(f::\text{tycon}) A) = (\text{emb } (\text{umap } (\text{cast } A) (\text{proj } x)) = x)$   
**by** (*simp add: in-idem-def cast-tc*)

**lemma** (*in functor-locale*) *emb-in-tc*:  
 $(\text{emb } (x::l) :: tc \text{TYPE}(f::\text{tycon}) A) = (\text{umap } (\text{cast } A) x = x)$   
**by** (*simp add: in-tc-iff coerce-def*)

**lemma** (*in functor-locale*) *emb-in-tc-2*:  
 $(\text{emb } (x::U \cdot f) :: tc \text{TYPE}(f::\text{tycon}) A) =$   
 $(\text{umap } (\text{cast } A) (\text{coerce } x) = \text{coerce } x)$   
**apply** (*simp add: in-tc-iff coerce-def*)  
**apply** *safe*  
**apply** (*erule subst, rule emb-inverse[symmetric]*)  
**apply** *simp*  
**done**

**lemma** (*in functor-locale*) *umap-type*:  
 $\text{emb } (\text{umap } g \text{ xs}) :: \text{rep-of } \text{TYPE}(U \cdot f::\text{tycon})$   
**by** (*simp add: rep-of-App in-tc-iff cast-rep-of-U umap-id*)

**theorem** (*in functor-locale*) *type*:  
**fixes**  $xs :: U \cdot f::\text{tycon}$   
**assumes**  $P: \bigwedge x. x :: A \implies f x :: B$   
**shows**  $\text{emb } xs :: tc \text{TYPE}(f) A \implies \text{emb } (\text{rep-fmap } f \text{ xs}) :: tc \text{TYPE}(f) B$   
**apply** (*simp add: rep-fmap rep-fmap-of-def*)  
**apply** (*erule in-idemE*)  
**apply** (*rule in-idemI*)  
**apply** (*simp add: cast-tc*)  
**apply** (*simp add: coerce-def*)  
**apply** (*simp add: umap-umap*)  
**apply** (*rule-tac x=proj y in fun-cong*)  
**apply** (*rule-tac f=umap in arg-cong*)  
**apply** (*rule ext*)  
**apply** (*rule cast-fixed*)  
**apply** (*rule P [OF cast-in-idem]*)  
**done**

**lemma** (*in functor-locale*) *cast*:  
**fixes**  $xs :: U \cdot f::\text{tycon}$   
**shows**  $\text{rep-fmap } (\text{cast } A) \text{ xs} = \text{proj } (\text{cast } (tc \text{TYPE}(f) A) (\text{emb } \text{xs}))$   
**by** (*simp add: rep-fmap cast-tc rep-fmap-of-def coerce-def*)

**lemma** (*in functor-locale*) *comp*:  
**fixes**  $xs :: U \cdot f::\text{tycon}$   
**shows**  $\text{rep-fmap } f (\text{rep-fmap } g \text{ xs}) = \text{rep-fmap } (\lambda x. f (g x)) \text{ xs}$   
**by** (*simp add: rep-fmap rep-fmap-of-def umap-umap*)

**lemma** (*in functor-locale*) *functor-class*:

```

    OFCLASS('f::tycon, functor-class)
  apply (intro-classes)
    apply (rule type, fast, assumption)
    apply (rule cast)
    apply (rule comp)
  done

```

Isomorphism at other type instances

```

lemma (in functor-locale) rep-of-App-functor:
  fixes map-aU :: ('a ⇒ U) ⇒ 'k ⇒ 'l
  fixes map-Ua :: (U ⇒ 'a) ⇒ 'l ⇒ 'k
  assumes 1:  $\bigwedge xs. emb\ xs = emb\ (map-aU\ emb\ xs)$ 
  assumes 2:  $\bigwedge u. proj\ u = map-Ua\ proj\ (proj\ u)$ 
  assumes 3:  $\bigwedge f\ g\ xs. map-aU\ f\ (map-Ua\ g\ xs) = umap\ (\lambda x. f\ (g\ x))\ xs$ 
  shows rep-of TYPE('k) = rep-of TYPE('a·'f::tycon)
  apply (rule idempotent-equality, rule ext)
  apply (simp add: rep-of-App)
  apply (simp add: cast-tc cast-rep-of)
  apply (simp add: prems)
done

```

```

lemma functor-locale-fmap-def:
  fixes map-Ua :: (U ⇒ 'a) ⇒ 'l ⇒ 'k
  fixes map-Ub :: (U ⇒ 'b) ⇒ 'l ⇒ 'm
  fixes map-ab :: ('a ⇒ 'b) ⇒ 'k ⇒ 'm
  fixes map-bU :: ('b ⇒ U) ⇒ 'm ⇒ 'l
  fixes map-UU :: (U ⇒ U) ⇒ 'l ⇒ 'l
  assumes fl: functor-locale TYPE('f::functor) map-UU
  assumes rews:
     $\bigwedge xs. emb\ xs = emb\ (map-bU\ emb\ xs)$ 
     $\bigwedge u. proj\ u = map-Ua\ proj\ (proj\ u)$ 
     $\bigwedge f\ g\ xs. map-ab\ f\ (map-Ua\ g\ xs) = map-Ub\ (\lambda x. f\ (g\ x))\ xs$ 
     $\bigwedge f\ g\ xs. map-bU\ f\ (map-Ub\ g\ xs) = map-UU\ (\lambda x. f\ (g\ x))\ xs$ 
  shows fmap ≡  $\lambda f\ (xs::'a·'f::functor). coerce\ (map-ab\ f\ (coerce\ xs))$ 
  apply (rule eq-reflection, rule ext, rule ext)
  apply (unfold fmap-def)
  apply (simp add: fmap-coerce [symmetric])
  apply (simp add: fmap-def)
  apply (simp add: fl [THEN functor-locale.rep-fmap])
  apply (simp add: rep-fmap-of-def)
  apply (simp add: fl [THEN functor-locale.rep-of-App-U-functor])
  apply (simp add: fl [THEN functor-locale.umap-umap])
  apply (simp add: coerce-def rews)
done

```

end

## 7 Monad Class

```
theory MonadClass
imports FunctorClass
begin
```

**consts**

```
rep-return :: U ⇒ U·'m::tycon
rep-bind :: U·'m ⇒ (U ⇒ U·'m) ⇒ U·'m::tycon
```

**axclass** monad < functor, tycon

monad-rep-return-type:

```
x :: A ⇒ emb ((rep-return x)::U·'m::tycon) :: tc TYPE('m) A
```

monad-rep-bind-type:

```
[[emb (m::U·'m::tycon) :: tc TYPE('m) A;
  ∧x. x :: A ⇒ emb (f x) :: tc TYPE('m) B]]
⇒ emb (rep-bind m f) :: tc TYPE('m) B
```

monad-rep-fmap:

```
rep-fmap f m = rep-bind m (λx. rep-return (f x))
```

monad-rep-1:

```
rep-bind (rep-return x) f = f x
```

monad-rep-3:

```
rep-bind (rep-bind m f) g = rep-bind m (λx. rep-bind (f x) g)
```

**lemmas** rep-return-type = monad-rep-return-type

```
[of - rep-of TYPE('b), simplified tc-rep-of, standard]
```

**lemmas** rep-bind-type = monad-rep-bind-type

```
[of - rep-of TYPE('b) - rep-of TYPE('c),
  simplified tc-rep-of, standard]
```

**constdefs**

```
return :: 'a ⇒ 'a·'m::monad
return ≡ λx. coerce (rep-return (emb x)::U·'m)
```

**constdefs**

```
bind :: 'a·'m::monad ⇒ ('a ⇒ 'b·'m) ⇒ 'b·'m (infixl >>= 55)
bind ≡ λm k. coerce (rep-bind (coerce m::U·'m) (coerce ∘ k ∘ proj))
```

**syntax** (xsymbols)

```
bind :: 'a·'m::monad ⇒ ('a ⇒ 'b·'m) ⇒ 'b·'m (infixl ▷ 55)
```

do notation

**nonterminals**

do-bind do-binds

**syntax**

```

-bind :: [pttrn, 'a.'m::monad] ⇒ do-bind ((2- <- -) 10)
-binds :: [do-bind, do-binds] ⇒ do-binds (-;/ -)
:: do-bind ⇒ do-binds (-)
-do :: [do-binds, 'a.'m::monad] ⇒ 'a.'m ((do {-; (-)}))

```

**syntax** (*xsymbols*)

```

-bind :: pttrn ⇒ 'a.'m::monad ⇒ do-bind ((2- ← -) 10)

```

**translations**

```

-do (-binds b bs) e == -do b (-do bs e)
-do (-bind x m) k == bind m (λx. k)

```

## monad laws

**theorem** *monad-fmap*:

```

fmap f xs = xs ▷ (λx. return (f x))
apply (simp add: fmap-def bind-def return-def o-def)
apply (simp add: monad-rep-fmap)
apply (rule-tac f=coerce in arg-cong)
apply (rule-tac f=rep-bind (coerce xs) in arg-cong)
apply (rule ext)
apply (rule coerce-inverse[symmetric])
apply (simp add: rep-return-type)
done

```

**theorem** *monad-left-unit* [simp]: (return x ▷ f) = (f x)

```

apply (simp add: bind-def return-def o-def)
apply (subst coerce-inverse)
apply (simp add: rep-return-type)
apply (simp add: monad-rep-1)
done

```

**theorem** *monad-right-unit* [simp]: (m ▷ return) = m

```

apply (subgoal-tac fmap id m = m)
apply (simp only: monad-fmap)
apply simp
apply simp
done

```

**theorem** *monad-bind-assoc*: ((m ▷ f) ▷ g) = (m ▷ (λx. f x ▷ g))

```

apply (simp add: bind-def o-def)
apply (subst coerce-inverse)
apply (rule rep-bind-type)
apply (rule coerce-type, simp)
apply (simp add: coerce-type)
apply (simp add: monad-rep-3)
apply (rule-tac f=coerce in arg-cong)
apply (rule-tac f=rep-bind (coerce m) in arg-cong)

```

```

apply (rule ext)
apply (rule coerce-inverse[symmetric])
apply (rule rep-bind-type)
  apply (rule coerce-type, simp)
apply (simp add: coerce-type)
done

```

laws for fmap

```

lemma fmap-return: fmap f (return x) = return (f x)
by (simp add: monad-fmap)

```

```

lemma fmap-bind: fmap f (bind m k) = bind m (λx. fmap f (k x))
by (simp add: monad-fmap monad-bind-assoc)

```

```

lemma bind-fmap: bind (fmap f m) k = bind m (λx. k (f x))
by (simp add: monad-fmap monad-bind-assoc)

```

```

lemma congruent-bind: (∀ m. m ▷ k1 = m ▷ k2) = (k1 = k2)
apply (safe, rule ext)
apply (drule-tac x=return x in spec, simp)
done

```

laws for join

**constdefs**

```

  join :: ('a.'m::monad). 'm ⇒ 'a.'m
  join ≡ λm. m ▷ (λx. x)

```

```

lemma join-fmap-fmap: join (fmap (fmap f) xss) = fmap f (join xss)
by (simp add: join-def monad-fmap monad-bind-assoc)

```

```

lemma join-return: join (return xs) = xs
by (simp add: join-def)

```

```

lemma join-fmap-return: join (fmap return xs) = xs
by (simp add: join-def monad-fmap monad-bind-assoc)

```

```

lemma join-fmap-join: join (fmap join xsss) = join (join xsss)
by (simp add: join-def monad-fmap monad-bind-assoc)

```

```

lemma bind-def2: m ▷ k = join (fmap k m)
by (simp add: join-def monad-fmap monad-bind-assoc)

```

equivalence of monad laws and fmap/join laws

```

lemma (return x ▷ f) = (f x)
by (simp only: bind-def2 fmap-return join-return)

```

```

lemma (m ▷ return) = m
by (simp only: bind-def2 join-fmap-return)

```

```

lemma ((m ▷ f) ▷ g) = (m ▷ (λx. f x ▷ g))
apply (simp only: bind-def2)
apply (subgoal-tac join (fmap g (join (fmap f m))) =
  join (fmap join (fmap (fmap g) (fmap f m))))
apply (simp add: fmap-fmap)
apply (simp add: join-fmap-join join-fmap-fmap)
done

```

## 7.1 locale for proving stuff about monad representations

**constdefs**

```

rep-return-of :: (U ⇒ 'l) ⇒ (U ⇒ U·'f::tycon)
rep-return-of uret ≡ coerce ∘ uret

```

```

rep-bind-of :: ('l ⇒ (U ⇒ 'l) ⇒ 'l) ⇒
  (U·'f ⇒ (U ⇒ U·'f) ⇒ U·'f::tycon)
rep-bind-of ubind ≡ λm k. coerce (ubind (coerce m) (coerce ∘ k))

```

**locale** monad-locale = functor-locale +

**fixes** uret :: U ⇒ 'l

**and** ubind :: 'l ⇒ (U ⇒ 'l) ⇒ 'l

**assumes** rep-return: rep-return :: U ⇒ U·'f::tycon

≡ rep-return-of uret

**and** rep-bind: rep-bind :: U·'f ⇒ (U ⇒ U·'f) ⇒ U·'f::tycon

≡ rep-bind-of ubind

**and** umap-def:  $\bigwedge f xs. \text{umap } f xs = \text{ubind } xs (\lambda x. \text{uret } (f x))$

**and** m1:  $\bigwedge f x. \text{ubind } (\text{uret } x) f = f x$

**and** m3:  $\bigwedge xs f g. \text{ubind } (\text{ubind } xs f) g = \text{ubind } xs (\lambda x. \text{ubind } (f x) g)$

**lemma** (in monad-locale) umap-uret:  $\text{umap } f (\text{uret } x) = \text{uret } (f x)$

**by** (simp add: umap-def m1)

**lemma** (in monad-locale) ubind-umap:

$\text{ubind } (\text{umap } f xs) k = \text{ubind } xs (\lambda x. k (f x))$

**by** (simp add: umap-def m3 m1)

**lemma** (in monad-locale) umap-ubind:

$\text{umap } f (\text{ubind } xs k) = \text{ubind } xs (\lambda x. \text{umap } f (k x))$

**by** (simp add: umap-def m3)

**lemma** (in monad-locale) return-type:

$x :: A \implies \text{emb } ((\text{rep-return } x)::U·'f::tycon)$

$::: \text{tc } \text{TYPE}'f A$

**apply** (simp add: rep-return rep-return-of-def)

**apply** (subst emb-in-tc)

**apply** (simp add: umap-uret)

**apply** (simp add: cast-fixed)

**done**

```

lemma (in monad-locale) bind-type:
assumes  $Pf: \bigwedge x. x :: A \implies \text{emb } (f x) :: \text{tc } \text{TYPE}('f::\text{tycon}) B$ 
shows  $\text{emb } (m::U \cdot 'f) :: \text{tc } \text{TYPE}('f::\text{tycon}) A$ 
   $\implies \text{emb } (\text{rep-bind } m f) :: \text{tc } \text{TYPE}('f) B$ 
apply (simp add: rep-bind rep-bind-of-def)
apply (simp add: emb-in-tc emb-in-tc-2)
apply (erule subst)
apply (simp add: ubind-umap umap-ubind)
apply (rule-tac f=ubind (coerce m) in arg-cong)
apply (rule ext)
apply (rule emb-in-tc [THEN iffD1])
apply simp
apply (rule Pf [OF cast-in-idem])
done

```

```

lemma (in monad-locale) monad-fmap:
fixes  $m :: U \cdot 'f::\text{tycon}$ 
shows  $\text{rep-fmap } f m = \text{rep-bind } m (\lambda x. \text{rep-return } (f x))$ 
apply (simp add: rep-fmap rep-bind rep-return)
apply (simp add: rep-fmap-of-def rep-bind-of-def rep-return-of-def)
apply (simp only: umap-def)
apply (simp add: o-def)
done

```

```

lemma (in monad-locale) monad1:
fixes  $f :: U \Rightarrow U \cdot 'f::\text{tycon}$ 
shows  $\text{rep-bind } (\text{rep-return } x) f = f x$ 
by (simp add: rep-bind rep-return rep-bind-of-def rep-return-of-def m1)

```

```

lemma (in monad-locale) monad3:
fixes  $f :: U \Rightarrow U \cdot 'f::\text{tycon}$ 
  and  $g :: U \Rightarrow U \cdot 'f::\text{tycon}$ 
shows  $\text{rep-bind } (\text{rep-bind } m f) g$ 
   $= \text{rep-bind } m (\lambda x. \text{rep-bind } (f x) g)$ 
apply (simp add: rep-bind rep-bind-of-def)
apply (subst m3)
apply (simp add: o-def)
done

```

alternate introduction rule for monad-locale

```

lemma functor-locale-intro2:
fixes  $\text{umap} :: (U \Rightarrow U) \Rightarrow 'l \Rightarrow 'l$ 
  and  $\text{uret} :: U \Rightarrow 'l$ 
  and  $\text{ubind} :: 'l \Rightarrow (U \Rightarrow 'l) \Rightarrow 'l$ 
assumes tc-umap:  $\text{monotc } \text{TYPE}('f::\text{tycon}) \equiv \text{functor-tc } \text{umap}$ 
  and rep-fmap:  $\text{rep-fmap} :: (U \Rightarrow U) \Rightarrow U \cdot 'f \Rightarrow U \cdot 'f::\text{tycon}$ 
   $\equiv \text{rep-fmap-of } \text{umap}$ 
  and umap-def:  $\bigwedge f xs. \text{umap } f xs = \text{ubind } xs (\lambda x. \text{uret } (f x))$ 
  and m1:  $\bigwedge f x. \text{ubind } (\text{uret } x) f = f x$ 

```

```

    and m2:  $\bigwedge xs. \text{ubind } xs \text{ uret} = xs$ 
    and m3:  $\bigwedge xs f g. \text{ubind } (\text{ubind } xs f) g = \text{ubind } xs (\lambda x. \text{ubind } (f x) g)$ 
shows functor-locale TYPE('f::tycon) umap
apply (rule functor-locale.intro)
  apply (rule tc-umap)
  apply (rule rep-fmap)
  apply (simp add: umap-def m2)
  apply (simp add: umap-def m3 m1)
done

```

```

lemma monad-locale-intro2:
  fixes umap :: (U  $\Rightarrow$  U)  $\Rightarrow$  'l  $\Rightarrow$  'l
  and uret :: U  $\Rightarrow$  'l
  and ubind :: 'l  $\Rightarrow$  (U  $\Rightarrow$  'l)  $\Rightarrow$  'l
assumes tc-umap: monotc TYPE('f::tycon)  $\equiv$  functor-tc umap
  and rep-fmap: rep-fmap :: (U  $\Rightarrow$  U)  $\Rightarrow$  U.'f  $\Rightarrow$  U.'f::tycon
   $\equiv$  rep-fmap-of umap
  and rep-return: rep-return :: U  $\Rightarrow$  U.'f::tycon
   $\equiv$  rep-return-of uret
  and rep-bind: rep-bind :: U.'f  $\Rightarrow$  (U  $\Rightarrow$  U.'f)  $\Rightarrow$  U.'f::tycon
   $\equiv$  rep-bind-of ubind
  and umap-def:  $\bigwedge f xs. \text{umap } f xs = \text{ubind } xs (\lambda x. \text{uret } (f x))$ 
  and m1:  $\bigwedge f x. \text{ubind } (\text{uret } x) f = f x$ 
  and m2:  $\bigwedge xs. \text{ubind } xs \text{ uret} = xs$ 
  and m3:  $\bigwedge xs f g. \text{ubind } (\text{ubind } xs f) g = \text{ubind } xs (\lambda x. \text{ubind } (f x) g)$ 
shows monad-locale TYPE('f::tycon) umap uret ubind
apply (rule monad-locale.intro)
  apply (rule functor-locale-intro2, assumption+)
  apply (rule monad-locale-axioms.intro, assumption+)
done

```

```

lemma (in monad-locale) monad-class:
  OFCLASS('f::tycon, monad-class)
apply (intro-classes)
apply (rule type, fast, assumption)
apply (rule cast)
apply (rule comp)
apply (erule return-type)
apply (rule bind-type, fast, assumption)
apply (rule monad-fmap)
apply (rule monad1)
apply (rule monad3)
done

```

Other type instances

```

lemmas ml2fl = monad-locale.axioms(1)

```

```

lemma monad-locale-return-def:
  fixes return-a :: 'a  $\Rightarrow$  'k

```

**fixes** *return-U* ::  $U \Rightarrow 'l$   
**fixes** *bind-UU* ::  $'l \Rightarrow (U \Rightarrow 'l) \Rightarrow 'l$   
**fixes** *map-aU* ::  $('a \Rightarrow U) \Rightarrow 'k \Rightarrow 'l$   
**fixes** *map-UU* ::  $(U \Rightarrow U) \Rightarrow 'l \Rightarrow 'l$   
**assumes** *ml*: *monad-locale* *TYPE*('f::monad) *map-UU* *return-U* *bind-UU*  
**assumes** *rews*:  
 $\bigwedge xs. emb\ xs = emb\ (map-aU\ emb\ xs)$   
 $\bigwedge f\ xs. map-aU\ f\ xs = bind-aU\ xs\ (\lambda x. return-U\ (f\ x))$   
 $\bigwedge f\ x. bind-aU\ (return-a\ x)\ f = f\ x$   
**shows** *return*  $\equiv \lambda x. (coerce::'k \Rightarrow 'a \cdot 'f::monad)\ (return-a\ x)$   
**apply** (*rule eq-reflection*, *rule ext*)  
**apply** (*unfold return-def*)  
**apply** (*simp add*: *ml* [*THEN monad-locale.rep-return*])  
**apply** (*simp add*: *rep-return-of-def*)  
**apply** (*simp add*: *fmap-coerce* [*symmetric*])  
**apply** (*simp add*: *fmap-def*)  
**apply** (*simp add*: *ml* [*THEN ml2fl* [*THEN functor-locale.rep-fmap*]])  
**apply** (*simp add*: *rep-fmap-of-def*)  
**apply** (*simp add*: *ml* [*THEN ml2fl* [*THEN functor-locale.rep-of-App-U-functor*]])  
**apply** (*simp add*: *ml* [*THEN monad-locale.umap-def*])  
**apply** (*simp add*: *ml* [*THEN monad-locale.m1*])  
**apply** (*simp add*: *coerce-def rews*)  
**done**

**lemma** *monad-locale-bind-def*:

**fixes** *return-U* ::  $U \Rightarrow 'l$   
**fixes** *return-a* ::  $'a \Rightarrow 'k$   
**fixes** *return-b* ::  $'b \Rightarrow 'm$   
**fixes** *bind-UU* ::  $'l \Rightarrow (U \Rightarrow 'l) \Rightarrow 'l$   
**fixes** *bind-aU* ::  $'k \Rightarrow ('a \Rightarrow 'l) \Rightarrow 'l$   
**fixes** *bind-bU* ::  $'m \Rightarrow ('b \Rightarrow 'l) \Rightarrow 'l$   
**fixes** *bind-Ub* ::  $'l \Rightarrow (U \Rightarrow 'm) \Rightarrow 'm$   
**fixes** *bind-ab* ::  $'k \Rightarrow ('a \Rightarrow 'm) \Rightarrow 'm$   
**fixes** *map-Ua* ::  $(U \Rightarrow 'a) \Rightarrow 'l \Rightarrow 'k$   
**fixes** *map-aU* ::  $('a \Rightarrow U) \Rightarrow 'k \Rightarrow 'l$   
**fixes** *map-bU* ::  $('b \Rightarrow U) \Rightarrow 'm \Rightarrow 'l$   
**fixes** *map-Ub* ::  $(U \Rightarrow 'b) \Rightarrow 'l \Rightarrow 'm$   
**fixes** *map-UU* ::  $(U \Rightarrow U) \Rightarrow 'l \Rightarrow 'l$   
**assumes** *ml*: *monad-locale* *TYPE*('f::monad) *map-UU* *return-U* *bind-UU*  
**assumes** *rews*:  
 $\bigwedge xs. emb\ xs = emb\ (map-bU\ emb\ xs)$   
 $\bigwedge u. proj\ u = map-Ua\ proj\ (proj\ u)$   
 $\bigwedge u. proj\ u = map-Ub\ proj\ (proj\ u)$   
 $\bigwedge f\ xs. map-bU\ f\ xs = bind-bU\ xs\ (\lambda x. return-U\ (f\ x))$   
 $\bigwedge f\ xs. map-Ua\ f\ xs = bind-Ua\ xs\ (\lambda x. return-a\ (f\ x))$   
 $\bigwedge f\ xs. map-Ub\ f\ xs = bind-Ub\ xs\ (\lambda x. return-b\ (f\ x))$   
 $\bigwedge x\ k. bind-aU\ (return-a\ x)\ k = k\ x$   
 $\bigwedge x\ k. bind-bU\ (return-b\ x)\ k = k\ x$   
 $\bigwedge xs\ f\ g. bind-bU\ (bind-ab\ xs\ f)\ g = bind-aU\ xs\ (\lambda x. bind-bU\ (f\ x)\ g)$

```

     $\wedge xs f g. \text{bind-aU} (\text{bind-Ua } xs f) g = \text{bind-UU } xs (\lambda x. \text{bind-aU} (f x) g)$ 
     $\wedge xs f g. \text{bind-bU} (\text{bind-Ub } xs f) g = \text{bind-UU } xs (\lambda x. \text{bind-bU} (f x) g)$ 
  shows  $\text{bind} \equiv \lambda(m::'a.'f::\text{monad}) (k::'a \Rightarrow 'b.'f).$ 
     $\text{coerce} (\text{bind-ab} (\text{coerce } m) (\lambda x. \text{coerce} (k x)))$ 
  apply (rule eq-reflection, rule ext, rule ext)
  apply (unfold bind-def)
  apply (simp add: ml [THEN monad-locale.rep-bind])
  apply (simp add: rep-bind-of-def)
  apply (simp add: fmap-coerce [symmetric])
  apply (simp add: fmap-def o-def)
  apply (simp add: ml [THEN ml2fl [THEN functor-locale.rep-fmap]])
  apply (simp add: rep-fmap-of-def)
  apply (simp add: ml [THEN ml2fl [THEN functor-locale.rep-of-App-U-functor]])
  apply (simp add: ml [THEN monad-locale.umap-ubind])
  apply (simp add: ml [THEN ml2fl [THEN functor-locale.umap-umap]])
  apply (simp add: ml [THEN monad-locale.umap-def])
  apply (simp add: coerce-def rews)
done

end

```

## 8 Functor and Monad Examples

```

theory Examples
imports MonadClass
begin

```

### 8.1 Lists

#### 8.1.1 *rep* instance

Lists of representable elements are representable.

```

consts

```

```

  list-emb :: U list  $\Rightarrow$  U
  list-proj :: U  $\Rightarrow$  U list

```

```

primrec

```

```

  list-emb [] = Utag 0 []
  list-emb (x # xs) = Utag 1 [x, list-emb xs]

```

```

recdef list-proj measure size

```

```

  list-proj (Utag 0 []) = []
  list-proj (Utag (Suc 0) [x,y]) = x # list-proj y

```

```

instance list :: (rep) rep-consts ..

```

```

defs (overloaded)

```

```

  emb-list-def: emb  $\equiv$  list-emb  $\circ$  map emb
  proj-list-def: proj  $\equiv$  map proj  $\circ$  list-proj

```

**instance** *list* :: (*rep*) *rep*  
**by** (*intro-classes*, *unfold emb-list-def proj-list-def*, *induct-tac x*, *simp-all*)

### 8.1.2 functor instance

**datatype** *List* = *List*

**instance** *List* :: *tycon* ..

**defs** (**overloaded**)

*monotc-List-def*: *monotc (l::List itself) ≡ functor-tc map*

*List* is in *functor-locale*

**defs** (**overloaded**)

*rep-fmap-List-def*:

*rep-fmap*::(*U* ⇒ *U*) ⇒ *U*·*List* ⇒ *U*·*List*

≡ *rep-fmap-of map*

**lemmas** *map-id* = *map-ident* [*folded id-def*]

**lemmas** *map-map* = *map-compose* [*unfolded o-def*, *symmetric*, *standard*]

**lemma** *functor-locale-List*: *functor-locale TYPE(List) map*

**apply** (*rule functor-locale.intro*)

**apply** (*rule monotc-List-def*)

**apply** (*rule rep-fmap-List-def*)

**apply** (*simp add: map-id*)

**apply** (*rule map-map*)

**done**

**instance** *List* :: *functor*

**apply** (*rule functor-locale.functor-class*)

**apply** (*rule functor-locale-List*)

**done**

### 8.1.3 monad instance

**constdefs**

*return-list* :: '*a* ⇒ '*a list*

*return-list* ≡ λ*x*. [*x*]

*bind-list* :: '*a list* ⇒ ('*a* ⇒ '*b list*) ⇒ '*b list*

*bind-list* ≡ λ*xs f*. *concat (map f xs)*

**lemma** *monad-fmap-list*: *map f xs = bind-list xs (λx. return-list (f x))*

**by** (*unfold bind-list-def return-list-def*, *induct-tac xs*, *simp-all*)

**lemma** *monad-left-unit-list*: *bind-list (return-list x) f = f x*

**by** (*unfold bind-list-def return-list-def*, *simp*)

**lemma** *monad-bind-assoc-list*:

$bind-list (bind-list xs f) g = bind-list xs (\lambda x. bind-list (f x) g)$   
**by** (*unfold bind-list-def return-list-def, induct-tac xs, simp-all*)

**defs (overloaded)**

*rep-return-List-def*:  
 $rep-return :: U \Rightarrow U \cdot List$   
 $\equiv rep-return-of return-list$

*rep-bind-List-def*:  
 $rep-bind :: U \cdot List \Rightarrow (U \Rightarrow U \cdot List) \Rightarrow U \cdot List$   
 $\equiv rep-bind-of bind-list$

**lemma monad-locale-List**:

*monad-locale TYPE(List) map return-list bind-list*  
**apply** (*rule monad-locale.intro*)  
**apply** (*rule functor-locale-List*)  
**apply** (*rule monad-locale-axioms.intro*)  
**apply** (*rule rep-return-List-def*)  
**apply** (*rule rep-bind-List-def*)  
**apply** (*rule monad-fmap-list*)  
**apply** (*rule monad-left-unit-list*)  
**apply** (*rule monad-bind-assoc-list*)  
**done**

**instance List :: monad**

**apply** (*rule monad-locale.monad-class*)  
**apply** (*rule monad-locale-List*)  
**done**

## 8.2 Option type

### 8.2.1 rep instance

Options of representable elements are representable

**consts** *emb-option* :: 'a::rep option  $\Rightarrow U$

**primrec**

$emb-option None = Utag 0 []$   
 $emb-option (Some x) = Utag 1 [emb x]$

**lemma inj-emb-option**: *inj emb-option*

**apply** (*simp add: inj-on-def*)  
**apply** (*rule allI, induct-tac x*)  
**apply** (*rule allI, case-tac y, simp, simp*)  
**apply** (*rule allI, case-tac y, simp, simp*)  
**done**

**instance option** :: (rep) rep-consts ..

**defs (overloaded)**

*option-emb-def*:  $emb \equiv emb-option$

*option-proj-def*:  $\text{proj} \equiv \text{inv emb-option}$

```
instance option :: (rep) rep
apply (intro-classes, unfold option-emb-def option-proj-def)
apply (rule inv-f-f, rule inj-emb-option)
done
```

### 8.2.2 functor instance

```
datatype Option = Option
```

```
instance Option :: tycon ..
```

```
defs (overloaded)
```

*Option-monotc-def*:  $\text{monotc } (l::\text{Option itself}) \equiv \text{functor-tc option-map}$

*Option* is in *functor-locale*

```
defs (overloaded)
```

*Option-rep-fmap-def*:

$\text{rep-fmap}::(U \Rightarrow U) \Rightarrow U \cdot \text{Option} \Rightarrow U \cdot \text{Option}$   
 $\equiv \text{rep-fmap-of option-map}$

```
lemma functor-locale-Option: functor-locale TYPE(Option) option-map
```

```
apply (rule functor-locale.intro)
```

```
apply (rule Option-monotc-def)
```

```
apply (rule Option-rep-fmap-def)
```

```
apply (case-tac xs, simp, simp)
```

```
apply (simp add: option-map-comp o-def)
```

```
done
```

```
instance Option :: functor
```

```
apply (rule functor-locale.functor-class)
```

```
apply (rule functor-locale-Option)
```

```
done
```

### 8.2.3 monad instance

```
defs (overloaded)
```

*Option-rep-return-def*:

$\text{rep-return}::U \Rightarrow U \cdot \text{Option}$   
 $\equiv \text{rep-return-of Some}$

*Option-rep-bind-def*:

$\text{rep-bind}::U \cdot \text{Option} \Rightarrow (U \Rightarrow U \cdot \text{Option}) \Rightarrow U \cdot \text{Option}$   
 $\equiv \text{rep-bind-of } (\lambda m k. \text{option-case None } k m)$

```
lemma monad-locale-Option:
```

*monad-locale TYPE(Option) option-map Some*  $(\lambda m k. \text{option-case None } k m)$

```
apply (rule monad-locale.intro)
```

```
apply (rule functor-locale-Option)
```

```
apply (rule monad-locale-axioms.intro)
```

```

apply (rule Option-rep-return-def)
apply (rule Option-rep-bind-def)
apply (simp split: option.split)
apply simp
apply (simp split: option.split)
done

```

```

instance Option :: monad
apply (rule monad-locale.monad-class)
apply (rule monad-locale-Option)
done

```

## 8.3 Trivial type constructor

### 8.3.1 functor instance

```

datatype Trivial = Trivial

```

```

instance Trivial :: tycon ..

```

```

defs (overloaded)

```

```

  Trivial-monotc-def: monotc (l::Trivial itself)  $\equiv$  functor-tc ( $\lambda f$  xs. ())

```

*Trivial* is in *functor-locale*

```

defs (overloaded)

```

```

  Trivial-rep-fmap-def:

```

```

  rep-fmap::(U  $\Rightarrow$  U)  $\Rightarrow$  U·Trivial  $\Rightarrow$  U·Trivial
   $\equiv$  rep-fmap-of ( $\lambda f$  xs. ())

```

```

lemma functor-locale-Trivial: functor-locale TYPE(Trivial) ( $\lambda f$  xs. ())

```

```

apply (rule functor-locale.intro)

```

```

apply (rule Trivial-monotc-def)

```

```

apply (rule Trivial-rep-fmap-def)

```

```

apply simp

```

```

apply simp

```

```

done

```

```

instance Trivial :: functor

```

```

apply (rule functor-locale.functor-class)

```

```

apply (rule functor-locale-Trivial)

```

```

done

```

### 8.3.2 monad instance

```

defs (overloaded)

```

```

  Trivial-rep-return-def:

```

```

  rep-return::U  $\Rightarrow$  U·Trivial
   $\equiv$  rep-return-of ( $\lambda x$ . ())

```

```

  Trivial-rep-bind-def:

```

```

  rep-bind::U·Trivial  $\Rightarrow$  (U  $\Rightarrow$  U·Trivial)  $\Rightarrow$  U·Trivial

```

$\equiv \text{rep-bind-of } (\lambda m k. ())$

```
lemma monad-locale-Trivial:  
  monad-locale TYPE(Trivial) ( $\lambda f xs. ()$ ) ( $\lambda x. ()$ ) ( $\lambda m k. ()$ )  
apply (rule monad-locale.intro)  
apply (rule functor-locale-Trivial)  
apply (rule monad-locale-axioms.intro)  
apply (rule Trivial-rep-return-def)  
apply (rule Trivial-rep-bind-def)  
apply simp  
apply simp  
apply simp  
done  
  
instance Trivial :: monad  
apply (rule monad-locale.monad-class)  
apply (rule monad-locale-Trivial)  
done  
  
end
```

## 9 State monad transformer

```
theory StateT  
imports MonadClass  
begin
```

### 9.1 Type definition and monad operations

```
datatype ('a,'m,'s) stateT =  
  stateT 's  $\Rightarrow$  ('a  $\times$  's)·'m::tycon
```

```
consts  
  run-stateT :: ('a,'m,'s) stateT  $\Rightarrow$  's  $\Rightarrow$  ('a  $\times$  's)·'m::tycon
```

```
primrec  
  run-stateT (stateT x) = x
```

```
lemma run-stateT-inverse [simp]: stateT (run-stateT x) = x  
by (induct x, simp)
```

```
constdefs  
  fmap-stateT :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a,'m::functor,'s)stateT  $\Rightarrow$  ('b,'m,'s)stateT  
  fmap-stateT  $\equiv$   $\lambda f m. \text{stateT } (\lambda s. \text{fmap } (\lambda(x, s). (f x, s)) (\text{run-stateT } m s))$   
  
  return-stateT :: 'a  $\Rightarrow$  ('a,'m::monad,'s)stateT
```

$return\text{-}stateT \equiv \lambda x. stateT (\lambda s. return (x,s))$

$bind\text{-}stateT :: ('a,'m::monad,'s)stateT \Rightarrow$   
 $( 'a \Rightarrow ('c,'m,'s)stateT ) \Rightarrow ('c,'m,'s)stateT$   
 $bind\text{-}stateT \equiv \lambda m k. stateT (\lambda s.$   
 $do \{(x,s') \leftarrow run\text{-}stateT m s; run\text{-}stateT (k x) s'\})$

### constdefs

$lift\text{-}stateT :: 'a.'m \Rightarrow ('a,'m::functor,'s)stateT$   
 $lift\text{-}stateT \equiv \lambda m. stateT (\lambda s. fmap (\lambda x. (x,s)) m)$

$get\text{-}stateT :: ('s,'m::monad,'s)stateT$   
 $get\text{-}stateT \equiv stateT (\lambda s. return (s, s))$

$set\text{-}stateT :: 's \Rightarrow (unit,'m::monad,'s)stateT$   
 $set\text{-}stateT \equiv \lambda x. stateT (\lambda s. return ((),x))$

**lemma**  $fmap\text{-}id\text{-}stateT$ :  $fmap\text{-}stateT id xs = xs$   
**by** (*simp add: fmap-stateT-def split-def*)

**lemma**  $fmap\text{-}fmap\text{-}stateT$ :  
 $fmap\text{-}stateT f (fmap\text{-}stateT g xs) = fmap\text{-}stateT (\lambda x. f (g x)) xs$   
**by** (*simp add: fmap-stateT-def fmap-fmap split-def*)

**lemma**  $monad\text{-}fmap\text{-}stateT$ :  
 $fmap\text{-}stateT f xs = bind\text{-}stateT xs (\lambda x. return\text{-}stateT (f x))$   
**apply** (*simp add: fmap-stateT-def bind-stateT-def return-stateT-def*)  
**apply** (*simp add: monad-fmap split-def*)  
**done**

**lemma**  $monad\text{-}left\text{-}unit\text{-}stateT$ :  $bind\text{-}stateT (return\text{-}stateT x) f = f x$   
**by** (*simp add: bind-stateT-def return-stateT-def*)

**lemma**  $monad\text{-}bind\text{-}assoc\text{-}stateT$ :  
 $bind\text{-}stateT (bind\text{-}stateT xs f) g =$   
 $bind\text{-}stateT xs (\lambda x. bind\text{-}stateT (f x) g)$   
**by** (*simp add: bind-stateT-def monad-bind-assoc split-def*)

## 9.2 rep instance

**instance**  $stateT :: (rep, functor, \{rep,finrep\}) rep\text{-}consts ..$

### defs (overloaded)

$emb\text{-}stateT\text{-}def$ :  $emb \equiv emb \circ run\text{-}stateT \circ fmap\text{-}stateT emb$   
 $proj\text{-}stateT\text{-}def$ :  $proj \equiv fmap\text{-}stateT proj \circ stateT \circ proj$

**instance**  $stateT :: (rep, functor, \{rep,finrep\}) rep$   
**apply** (*intro-classes*)  
**apply** (*unfold emb-stateT-def proj-stateT-def*)

```

apply (simp add: fmap-fmap-stateT fmap-id-stateT [unfolded id-def])
done

```

### 9.3 functor instance

```

datatype ('m,'s) StateT = StateT

```

```

instance StateT :: (functor, {rep,finrep}) tycon ..

```

```

defs (overloaded)

```

```

  monotc-StateT-def:

```

```

    monotc (t::('m::functor,'s::{rep,finrep}) StateT itself) ≡ functor-tc
    (fmap-stateT :: (U ⇒ U) ⇒ (U,'m,'s) stateT ⇒ (U,'m,'s) stateT)

```

StateT is in *functor-locale*

```

defs (overloaded)

```

```

  rep-fmap-StateT-def:

```

```

  rep-fmap::(U ⇒ U) ⇒

```

```

    U·('m::functor,'s::{rep,finrep})StateT ⇒ U·('m,'s)StateT

```

```

    ≡ rep-fmap-of

```

```

    (fmap-stateT :: (U ⇒ U) ⇒ (U,'m,'s) stateT ⇒ (U,'m,'s) stateT)

```

```

lemma functor-locale-StateT:

```

```

  functor-locale TYPE(('m::functor,'s::{rep,finrep}) StateT)

```

```

  (fmap-stateT :: (U ⇒ U) ⇒ (U,'m,'s) stateT ⇒ (U,'m,'s) stateT)

```

```

apply (rule functor-locale.intro)

```

```

apply (rule monotc-StateT-def)

```

```

apply (rule rep-fmap-StateT-def)

```

```

apply (rule fmap-id-stateT)

```

```

apply (rule fmap-fmap-stateT)

```

```

done

```

```

instance StateT :: (functor, {rep,finrep}) functor

```

```

apply (rule functor-locale.functor-class)

```

```

apply (rule functor-locale-StateT)

```

```

done

```

### 9.4 StateT is isomorphic to stateT

```

constdefs

```

```

  abs-StateT :: ('a,'m::functor,'s::{finrep,rep}) stateT ⇒ 'a·('m,'s) StateT

```

```

  abs-StateT ≡ coerce

```

```

  rep-StateT :: 'a·('m::functor,'s::{finrep,rep}) StateT ⇒ ('a,'m,'s) stateT

```

```

  rep-StateT ≡ coerce

```

```

lemma emb-stateT: emb xs = emb (fmap-stateT emb xs)

```

```

by (simp add: emb-stateT-def fmap-id-stateT)

```

```

lemma proj-stateT: proj y = fmap-stateT proj (proj y)

```

by (simp add: proj-stateT-def fmap-id-stateT)

**lemma** rep-of-stateT:

rep-of TYPE(('a,'m::functor,'s::{finrep,rep}) stateT) =  
rep-of TYPE('a·('m,'s) StateT)  
apply (rule functor-locale.rep-of-App-functor)  
apply (rule functor-locale-StateT)  
apply (rule emb-stateT)  
apply (rule proj-stateT)  
apply (rule fmap-fmap-stateT)  
done

**lemma** StateT-iso [simp]:

rep-StateT (abs-StateT x) = x  
abs-StateT (rep-StateT y) = y  
by (simp-all add: rep-StateT-def abs-StateT-def rep-of-stateT)

## 9.5 monad instance

**defs** (overloaded)

rep-return-StateT-def:  
rep-return::U  $\Rightarrow$  U·('m::monad,'s::{rep,finrep})StateT  
 $\equiv$  rep-return-of (return-stateT::U  $\Rightarrow$  (U,'m,'s)stateT)

rep-bind-StateT-def:  
rep-bind::U·('m::monad,'s::{rep,finrep})StateT  $\Rightarrow$   
(U  $\Rightarrow$  U·('m,'s)StateT)  $\Rightarrow$  U·('m,'s)StateT  
 $\equiv$  rep-bind-of (bind-stateT::  
(U,'m,'s)stateT  $\Rightarrow$  (U  $\Rightarrow$  (U,'m,'s)stateT)  $\Rightarrow$  (U,'m,'s)stateT)

**lemma** monad-locale-StateT:

monad-locale TYPE(('m::monad,'s::{rep,finrep})StateT)  
fmap-stateT (return-stateT::U  $\Rightarrow$  (U,'m,'s)stateT) bind-stateT  
apply (rule monad-locale.intro)  
apply (rule functor-locale-StateT)  
apply (rule monad-locale-axioms.intro)  
apply (rule rep-return-StateT-def)  
apply (rule rep-bind-StateT-def)  
apply (rule monad-fmap-stateT)  
apply (rule monad-left-unit-stateT)  
apply (rule monad-bind-assoc-stateT)  
done

**instance** StateT :: (monad, {rep,finrep}) monad

apply (rule monad-locale.monad-class)  
apply (rule monad-locale-StateT)  
done

## 9.6 Other functions

**constdefs**

$lift-StateT :: 'a \cdot 'm \Rightarrow 'a \cdot ('m::functor, 's::\{finrep, rep\}) StateT$   
 $lift-StateT \equiv \lambda m. abs-StateT (lift-stateT m)$

$get-StateT :: 's \cdot ('m::monad, 's::\{finrep, rep\}) StateT$   
 $get-StateT \equiv abs-StateT get-stateT$

$set-StateT :: 's \Rightarrow unit \cdot ('m::monad, 's::\{finrep, rep\}) StateT$   
 $set-StateT \equiv \lambda x. abs-StateT (set-stateT x)$

**lemma** *fmap-StateT-def*:

$fmap \equiv \lambda f xs. abs-StateT (fmap-stateT f (rep-StateT xs))$   
**apply** (*unfold abs-StateT-def rep-StateT-def*)  
**apply** (*rule functor-locale-fmap-def*)  
**apply** (*rule functor-locale-StateT*)  
**apply** (*rule emb-stateT*)  
**apply** (*rule proj-stateT*)  
**apply** (*rule fmap-fmap-stateT*)+  
**done**

**lemma** *return-StateT-def*:  $return \equiv \lambda x. abs-StateT (return-stateT x)$

**apply** (*unfold abs-StateT-def*)  
**apply** (*rule monad-locale-return-def*)  
**apply** (*rule monad-locale-StateT*)  
**apply** (*rule emb-stateT*)  
**apply** (*rule monad-fmap-stateT*)  
**apply** (*rule monad-left-unit-stateT*)  
**done**

**lemma** *bind-StateT-def*:

$bind \equiv \lambda m k. abs-StateT$   
 $(bind-stateT (rep-StateT m) (\lambda x. rep-StateT (k x)))$   
**apply** (*unfold abs-StateT-def rep-StateT-def*)  
**apply** (*rule monad-locale-bind-def*)  
**apply** (*rule monad-locale-StateT*)  
**apply** (*rule emb-stateT*)  
**apply** (*rule proj-stateT*)+  
**apply** (*rule monad-fmap-stateT*)+  
**apply** (*rule monad-left-unit-stateT*)+  
**apply** (*rule monad-bind-assoc-stateT*)+  
**done**

**lemma**  $do \{u \leftarrow set-StateT x; get-StateT\} = do \{u \leftarrow set-StateT x; return x\}$

**apply** (*simp add*):  
 $bind-StateT-def return-StateT-def set-StateT-def get-StateT-def$   
**apply** (*simp add*):  
 $bind-stateT-def return-stateT-def set-stateT-def get-stateT-def$   
**done**

end