Tycon: Type Constructor Classes 
and Monad Transformers

Brian Huffman
April 20, 2020

Abstract
These theories contain a formalization of first class type constructors and axiomatic constructor classes for HOLCF. This work is described in detail in the ICFP 2012 paper “Formal Verification of Monad Transformers” by the author [1]. The formalization is a revised and updated version of earlier joint work with Matthews and White [3].

Based on the hierarchy of type classes in Haskell, we define classes for functors, monads, monad-plus, etc. Each one includes all the standard laws as axioms. We also provide a new user command, tycondef, for defining new type constructors in HOLCF. Using tycondef, we instantiate the type class hierarchy with various monads and monad transformers.

Contents

1 Type Application 5
  1.1 Class of type constructors ......................... 5
  1.2 Type constructor for type application ............ 5

2 Coercion Operator 6
  2.1 Coerce ........................................... 6
  2.2 More lemmas about emb and prj .................... 8
  2.3 Coercing various datatypes ....................... 8
  2.4 Simplifying coercions ............................ 9

3 Functor Class 10
  3.1 Class definition ................................ 10
  3.2 Polymorphic functor laws ........................ 11
  3.3 Derived properties of fmap ...................... 12
  3.4 Proving that fmap-coerce = coerce ............... 13
  3.5 Lemmas for reasoning about coercion ............. 13
  3.6 Configuration of Domain package ................. 14
  3.7 Configuration of the Tycon package ............. 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Monad Class</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Class definition</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Naturality of bind and return</td>
<td>15</td>
</tr>
<tr>
<td>4.3</td>
<td>Polymorphic versions of class assumptions</td>
<td>16</td>
</tr>
<tr>
<td>4.4</td>
<td>Derived rules</td>
<td>16</td>
</tr>
<tr>
<td>4.5</td>
<td>Laws for join</td>
<td>17</td>
</tr>
<tr>
<td>4.6</td>
<td>Equivalence of monad laws and fmap/join laws</td>
<td>17</td>
</tr>
<tr>
<td>4.7</td>
<td>Simplification of coercions</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>Monad-Zero Class</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>Monad-Plus Class</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>Monad-Zero-Plus Class</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>Lazy list monad</td>
<td>21</td>
</tr>
<tr>
<td>8.1</td>
<td>Type definition</td>
<td>22</td>
</tr>
<tr>
<td>8.2</td>
<td>Class instances</td>
<td>22</td>
</tr>
<tr>
<td>8.3</td>
<td>Transfer properties to polymorphic versions</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>Maybe monad</td>
<td>25</td>
</tr>
<tr>
<td>9.1</td>
<td>Type definition</td>
<td>25</td>
</tr>
<tr>
<td>9.2</td>
<td>Class instance proofs</td>
<td>26</td>
</tr>
<tr>
<td>9.3</td>
<td>Transfer properties to polymorphic versions</td>
<td>27</td>
</tr>
<tr>
<td>9.4</td>
<td>Maybe is not in monad-plus</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>Error monad</td>
<td>28</td>
</tr>
<tr>
<td>10.1</td>
<td>Type definition</td>
<td>28</td>
</tr>
<tr>
<td>10.2</td>
<td>Monad class instance</td>
<td>29</td>
</tr>
<tr>
<td>10.3</td>
<td>Transfer properties to polymorphic versions</td>
<td>30</td>
</tr>
<tr>
<td>11</td>
<td>Writer monad</td>
<td>30</td>
</tr>
<tr>
<td>11.1</td>
<td>Monoid class</td>
<td>31</td>
</tr>
<tr>
<td>11.2</td>
<td>Writer monad type</td>
<td>31</td>
</tr>
<tr>
<td>11.3</td>
<td>Class instance proofs</td>
<td>31</td>
</tr>
<tr>
<td>11.4</td>
<td>Transfer properties to polymorphic versions</td>
<td>32</td>
</tr>
<tr>
<td>11.5</td>
<td>Extra operations</td>
<td>33</td>
</tr>
<tr>
<td>12</td>
<td>Binary tree monad</td>
<td>33</td>
</tr>
<tr>
<td>12.1</td>
<td>Type definition</td>
<td>33</td>
</tr>
<tr>
<td>12.2</td>
<td>Class instance proofs</td>
<td>34</td>
</tr>
<tr>
<td>12.3</td>
<td>Transfer properties to polymorphic versions</td>
<td>34</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>13</td>
<td>Lift monad</td>
<td>35</td>
</tr>
<tr>
<td>13.1</td>
<td>Type definition</td>
<td>35</td>
</tr>
<tr>
<td>13.2</td>
<td>Class instance proofs</td>
<td>36</td>
</tr>
<tr>
<td>13.3</td>
<td>Transfer properties to polymorphic versions</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>Resumption monad transformer</td>
<td>37</td>
</tr>
<tr>
<td>14.1</td>
<td>Type definition</td>
<td>37</td>
</tr>
<tr>
<td>14.2</td>
<td>Class instance proofs</td>
<td>38</td>
</tr>
<tr>
<td>14.3</td>
<td>Transfer properties to polymorphic versions</td>
<td>39</td>
</tr>
<tr>
<td>14.4</td>
<td>Nondeterministic interleaving</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>State monad transformer</td>
<td>41</td>
</tr>
<tr>
<td>15.1</td>
<td>Functor class instance</td>
<td>42</td>
</tr>
<tr>
<td>15.2</td>
<td>Monad class instance</td>
<td>42</td>
</tr>
<tr>
<td>15.3</td>
<td>Monad zero instance</td>
<td>43</td>
</tr>
<tr>
<td>15.4</td>
<td>Monad plus instance</td>
<td>44</td>
</tr>
<tr>
<td>15.5</td>
<td>Monad zero plus instance</td>
<td>44</td>
</tr>
<tr>
<td>15.6</td>
<td>Transfer properties to polymorphic versions</td>
<td>44</td>
</tr>
<tr>
<td>16</td>
<td>Error monad transformer</td>
<td>45</td>
</tr>
<tr>
<td>16.1</td>
<td>Type definition</td>
<td>45</td>
</tr>
<tr>
<td>16.2</td>
<td>Functor class instance</td>
<td>46</td>
</tr>
<tr>
<td>16.3</td>
<td>Transfer properties to polymorphic versions</td>
<td>47</td>
</tr>
<tr>
<td>16.4</td>
<td>Monad operations</td>
<td>47</td>
</tr>
<tr>
<td>16.5</td>
<td>Laws</td>
<td>49</td>
</tr>
<tr>
<td>16.6</td>
<td>Error monad transformer invariant</td>
<td>50</td>
</tr>
<tr>
<td>16.7</td>
<td>Invariant expressed as a deflation</td>
<td>51</td>
</tr>
<tr>
<td>17</td>
<td>Writer monad transformer</td>
<td>53</td>
</tr>
<tr>
<td>17.1</td>
<td>Type definition</td>
<td>53</td>
</tr>
<tr>
<td>17.2</td>
<td>Functor class instance</td>
<td>54</td>
</tr>
<tr>
<td>17.3</td>
<td>Monad operations</td>
<td>55</td>
</tr>
<tr>
<td>17.4</td>
<td>Laws</td>
<td>56</td>
</tr>
<tr>
<td>17.5</td>
<td>Writer monad transformer invariant</td>
<td>58</td>
</tr>
<tr>
<td>17.6</td>
<td>Invariant expressed as a deflation</td>
<td>60</td>
</tr>
</tbody>
</table>
1 Type Application

theory TypeApp
imports HOLCF
begin

1.1 Class of type constructors

In HOLCF, the type \( \text{udom defl} \) consists of deflations over the universal domain—each value of type \( \text{udom defl} \) represents a bifinite domain. In turn, values of the continuous function type \( \text{udom defl} \rightarrow \text{udom defl} \) represent functions from domains to domains, i.e. type constructors.

Class \textit{tycon}, defined below, will be populated with dummy types: For example, if the type \( \text{foo} \) is an instance of class \textit{tycon}, then users will never deal with any values \( x :: \text{foo} \) in practice. Such types are only used with the overloaded constant \textit{tc}, which associates each type \( 'a :: \text{tycon} \) with a value of type \( \text{udom defl} \rightarrow \text{udom defl} \).

\texttt{class tycon =}
\texttt{  fixes tc :: ('a::type) itself \Rightarrow udom defl \rightarrow udom defl}

Type \( 'a \) itself is defined in Isabelle’s meta-logic; it is inhabited by a single value, written \( \text{TYPE}('a) \). We define the syntax \( TC('a) \) to abbreviate \( tc \ \text{TYPE}('a) \).

\texttt{syntax -TC :: type \Rightarrow logic \ ((1TC/(1'('))) )}

\texttt{translations TC('a) = CONST tc \ \text{TYPE}('a) '}

1.2 Type constructor for type application

We now define a binary type constructor that models type application: Type \( ('a, 't) \ app \) is the result of applying the type constructor \( 't \) (from class \textit{tycon} \) to the type argument \( 'a \) (from class \textit{domain} \).

We define type \( ('a, 't) \ app \) using \textit{domaindef}, a low-level type-definition command provided by HOLCF (similar to \textit{typedef} in Isabelle/HOL) that defines a new domain type represented by the given deflation. Note that in HOLCF, \( \text{DEFL('a)} \) is an abbreviation for \( \text{defl TYPE('a)} \), where \( \text{defl :: ('a::domain) itself \Rightarrow udom defl} \) is an overloaded function from the \textit{domain} type class that yields the deflation representing the given type.

\texttt{domaindef ('a,'t) app = TC('t::tycon)\cdot\text{DEFL('a::domain)}}

We define the infix syntax \( 'a \cdot 't \) for the type \( ('a,'t) \ app \). Note that for consistency with Isabelle’s existing type syntax, we have used postfix order
for type application: type argument on the left, type constructor on the right.

type-notation app ((--) [999,1000] 999)

The domaindef command generates the theorem DEFL-app: DEFL(\(?\'a\cdot\'?t) = TC(\'?t)\cdot DEFL(\'?a), which we can use to derive other useful lemmas.

lemma TC-DEFL: TC(\'(t::tycon)\cdot DEFL(\'?a) = DEFL(\'?a)
by (rule DEFL-app [symmetric])

lemma DEFL-app-mono [simp, intro]:
DEFL(\'?a) ▪ DEFL(\'?b) ⊑ DEFL(\'(a\cdot tycon) ▪ DEFL(\'?b\cdot t)
apply (simp add: DEFL-app)
apply (erule monofun-cfun-arg)
done

end

2 Coercion Operator

theory Coerce
imports HOLCF
begin

2.1 Coerce

The domain type class, which is the default type class in HOLCF, fixes two overloaded functions: emb::\'a → udom and prj::udom → \'a. By composing the prj and emb functions together, we can coerce values between any two types in class domain.

definition coerce :: \'a → \'b
where coerce ≡ prj oo emb

When working with proofs involving emb, prj, and coerce, it is often difficult to tell at which types those constants are being used. To alleviate this problem, we define special input and output syntax to indicate the types.

syntax
-emb :: type ⇒ logic ((1EMB/(1'(-))))
-prj :: type ⇒ logic ((1PRJ/(1'(-))))
-coerce :: type ⇒ type ⇒ logic ((1COERCE/(1'(-)/-)))

translations
EMB(\'?a) → CONST emb :: \'a → udom
PRJ(\'?a) → CONST prj :: udom → \'a
COERCER(\'?a,\'?b) → CONST coerce :: \'a → \'b
let
fun emb-tr' (ctxt : Proof.context) (Type(\_, [T, _])) [] =
    Syntax.const @\{syntax-const -emb\} $ Syntax-Phases.term-of-typ ctxt T
fun prj-tr' ctxt (Type(\_, [\_, T])) [] =
    Syntax.const @\{syntax-const -prj\} $ Syntax-Phases.term-of-typ ctxt T
fun coerce-tr' ctxt (Type(\_, [T, U])) [] =
    Syntax.const @\{syntax-const -coerce\} $ Syntax-Phases.term-of-typ ctxt T $ Syntax-Phases.term-of-typ ctxt U
in
    [[@\{const-syntax emb\}, emb-tr'],
     (@\{const-syntax prj\}, prj-tr'),
     (@\{const-syntax coerce\}, coerce-tr')]
end

lemma beta-coerce: coerce \cdot x = prj \cdot (emb \cdot x)
  by (simp add: coerce-def)

lemma prj-emb: prj \cdot (emb \cdot x) = coerce \cdot x
  by (simp add: coerce-def)

lemma coerce-strict [simp]: coerce \cdot ⊥ = ⊥
  by (simp add: coerce-def)

Certain type instances of \texttt{coerce} may reduce to the identity function, \texttt{emb},
or \texttt{prj}.

lemma coerce-eq-ID [simp]: COERC\texttt{E}'(a, 'a) = ID
  by (rule cfun-eqI, simp add: beta-coerce)

lemma coerce-eq-emb [simp]: COERC\texttt{E}'(a, udom) = EMB'(a)
  by (rule cfun-eqI, simp add: beta-coerce)

lemma coerce-eq-prj [simp]: COERC\texttt{E}(udom, 'a) = PRJ'(a)
  by (rule cfun-eqI, simp add: beta-coerce)

Cancellation rules

lemma emb-coerce:
  \texttt{DEFL}'(a) ⊆ \texttt{DEFL}'(b)
  \implies EMB'(b) \cdot (COERC\texttt{E}'(a, 'b) \cdot x) = EMB'(a) \cdot x
  by (simp add: beta-coerce emb-prj-emb)

lemma coerce-prj:
  \texttt{DEFL}'(a) ⊆ \texttt{DEFL}'(b)
  \implies COERC\texttt{E}(b, 'a) \cdot (PRJ'(b) \cdot x) = PRJ'(a) \cdot x
  by (simp add: beta-coerce prj-emb-prj)
lemma coerce-idem [simp]:
\[ \text{DEFL}'(a) \sqsubseteq \text{DEFL}'(b) \implies \text{COERCE}'(b,c) \cdot (\text{COERCE}'(a,b) \cdot x) = \text{COERCE}'(a,c) \cdot x \]
by (simp add: beta-coerce emb-prj-emb)

2.2 More lemmas about emb and prj

lemma prj-cast-DEFL [simp]:
\[ \text{PRJ}'(a) \cdot (\text{cast} \cdot \text{DEFL}'(a) \cdot x) = \text{PRJ}'(a) \cdot x \]
by (simp add: cast-DEFL)

lemma cast-DEFL-emb [simp]:
\[ \text{cast} \cdot \text{DEFL}'(a) \cdot (\text{EMB}'(a) \cdot x) = \text{EMB}'(a) \cdot x \]
by (simp add: cast-DEFL)

DEFL(udom)

lemma below-DEFL-udom [simp]:
\[ A \sqsubseteq \text{DEFL}(udom) \]
apply (rule cast-below-imp-below)
apply (rule cast-belowI)
apply (simp add: cast-DEFL)
done

2.3 Coercing various datatypes

Coercing from the strict product type \( 'a \otimes 'b \) to another strict product type \( 'c \otimes 'd \) is equivalent to mapping the \text{coerce} function separately over each component using \text{sprod-map} \( :: \ ('a \to 'c) \to ('b \to 'd) \to 'a \otimes 'b \to 'c \otimes 'd \).
Each of the several type constructors defined in HOLCF satisfies a similar property, with respect to its own map combinator.

lemma coerce-u: \text{coerce} = \text{u-map-coerce}
apply (rule cfun-eqI, simp add: coerce-def)
apply (simp add: emb-u-def prj-u-def liftemb-eq liftprj-eq)
apply (subst ep-pair.e-inverse [OF ep-pair-u])
apply (simp add: u-map-map cfcomp1)
done

lemma coerce-sfun: \text{coerce} = \text{sfun-map-coerce-coerce}
apply (rule cfun-eqI, simp add: coerce-def)
apply (simp add: emb-sfun-def prj-sfun-def)
apply (subst ep-pair.e-inverse [OF ep-pair-sfun])
apply (simp add: sfun-map-map cfcomp1)
done

lemma coerce-cfun': \text{coerce} = \text{cfun-map-coerce-coerce}
apply (rule cfun-eqI, simp add: prj-emb [symmetric])
apply (simp add: emb-cfun-def prj-cfun-def)
apply (simp add: prj-emb coerce-sfun coerce-u)
apply (simp add: encode-cfun-map [symmetric])
done
2.4 Simplifying coercions

When simplifying applications of the \textit{coerce} function, rewrite rules are always oriented to replace \textit{coerce} at complex types with other applications of \textit{coerce} at simpler types.

The safest rewrite rules for \textit{coerce} are given the [simp] attribute. For other rules that do not belong in the global simpset, we use dynamic theorem list called \textit{coerce-simp}, which will collect additional rules for simplifying coercions.

\textbf{named-theorems} \textit{coerce-simp} rule for simplifying coercions

The \textit{coerce} function commutes with data constructors for various HOLCF datatypes.

\textbf{lemma} \textit{coerce-up} [simp]: \textit{coerce} \cdot \textit{up} \cdot x = \textit{up} \cdot \textit{coerce} \cdot x
\textbf{by} (simp add: coerence-frm)

\textbf{lemma} \textit{coerce-sinl} [simp]: \textit{coerce} \cdot \textit{sinl} \cdot x = \textit{sinl} \cdot \textit{coerce} \cdot x
\textbf{by} (simp add: coerence-sprod sprod-map-spair)

\textbf{lemma} \textit{coerce-sinr} [simp]: \textit{coerce} \cdot \textit{sinr} \cdot x = \textit{sinr} \cdot \textit{coerce} \cdot x
\textbf{by} (simp add: coerence-sprod sprod-map-spair)

\textbf{lemma} \textit{coerce-spair} [simp]: \textit{coerce} \cdot (x, y) = (\textit{coerce} \cdot x, \textit{coerce} \cdot y)
\textbf{by} (simp add: coerence-sprod sprod-map-spair)
lemma coerce-Pair [simp]: coerce·(x, y) = (coerce·x, coerce·y)
by (simp add: coerce-prod)

lemma beta-coerce-cfun [simp]: coerce·f·x = coerce·(f·(coerce·x))
by (simp add: coerce-cfun)

lemma coerce-cfun: coerce·f = coerce oo f oo coerce
by (simp add: cfun-eqI)

lemma coerce-cfun-app [coerce-simp]:
  coerce·f = (Λ x. coerce·(f·(coerce·x)))
by (simp add: cfun-eqI)

end

3 Functor Class

theory Functor
imports TypeApp Coerce
keywords tycondef :: thy-defn and ·
begin

3.1 Class definition

Here we define the functor class, which models the Haskell class Functor. For technical reasons, we split the definition of functor into two separate classes: First, we introduce prefunctor, which only requires fmap to preserve the identity function, and not function composition.

The Haskell class Functor f fixes a polymorphic function fmap :: (a -> b) -> f a -> f b. Since functions in Isabelle type classes can only mention one type variable, we have the prefunctor class fix a function fmapU that fixes both of the polymorphic types to be the universal domain. We will use the coercion operator to recover a polymorphic fmap.

The single axiom of the prefunctor class is stated in terms of the HOLCF constantisodefl, which relates a function f :: 'a -> 'a with a deflation t :: udom defl: isodefl f t = (cast·t = EMB('a) oo f oo PRJ('a)).

class prefunctor = tycon +
  fixes fmapU :: (udom -> udom) -> udom·'a -> udom·'a::tycon
  assumes isodefl-fmapU:
    isodefl (fmapU·(cast·t)) (TC('a::tycon·t))

The functor class extends prefunctor with an axiom stating that fmapU preserves composition.

class functor = prefunctor +
assumes fmapU-fmapU [coerce-simp]:
\[ f \mapsto g \ (xs :: \text{udom} \cdot 'a \Rightarrow 'b) \mapsto f \mapsto g \cdot xs \]

We define the polymorphic \( \text{fmap} \) by coercion from \( \text{fmap}U \), then we proceed to derive the polymorphic versions of the functor laws.

**Definition** \( \text{fmap} \) :: \( ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \) :: \( \text{functor} \)

where \( \text{fmap} = \text{coerce} \cdot (\text{fmap}U :: - \Rightarrow \text{udom} \cdot 'a \Rightarrow 'b) \cdot \text{xs} \)

### 3.2 Polymorphic functor laws

**Lemma** \( \text{fmapU-eq-fmap} \) : \( \text{fmapU} = \text{fmap} \) by \((\text{simp add: fmap-def eta-cfun})\)

**Lemma** \( \text{fmap-eq-fmapU} \) : \( \text{fmap} = \text{fmapU} \) by \((\text{simp only: fmapU-eq-fmap})\)

**Lemma** \( \text{cast-TC} \):
\[ \text{cast} \cdot (\text{TC} ('f) \cdot t) = \text{emb} \circ \text{fmapU} \circ (\text{cast} \cdot t) \circ \text{PRJ} ('\text{udom} \cdot 'f) \cdot \text{prefunctor} \]
by \((\text{rule isodefl-fmapU [unfolded isodefl-def]])\)

**Lemma** \( \text{isodefl-cast} \):
\[ \text{isodefl} \cdot (\text{cast} \cdot t) \cdot t \]
by \((\text{simp add: isodefl-def})\)

**Lemma** \( \text{cast-cast-below1} \): \( A \subseteq B \Rightarrow \text{cast} \cdot A \cdot (\text{cast} \cdot B \cdot x) = \text{cast} \cdot A \cdot x \)
by \((\text{intro deflation-below-comp1 deflation-cast monofun-cfun-arg})\)

**Lemma** \( \text{cast-cast-below2} \): \( A \subseteq B \Rightarrow \text{cast} \cdot B \cdot (\text{cast} \cdot A \cdot x) = \text{cast} \cdot A \cdot x \)
by \((\text{intro deflation-below-comp2 deflation-cast monofun-cfun-arg})\)

**Lemma** \( \text{isodefl-fmap} \):
assumes isodefl d t
shows isodefl \((\text{fmap} \cdot d :: 'a \Rightarrow 'b) \Rightarrow -) \cdot (\text{TC} ('f) \cdot \text{functor}) \cdot t\)
proof –
have deflation-d: deflation d
using assms by \((\text{rule isodefl-imp-deflation})\)
have cast-t: cast-t = emb oo d oo prj
using assms unfolding isodefl-def .
have t-below: t \(\subseteq\) DEFL ('a)
apply \((\text{rule deflation-below-comp1 deflation-cast monofun-cfun-arg})\)
apply \((\text{simp only: cast-t cast-DEFL})\)
apply \((\text{simp add: cfun-below-iff deflation.below [OF deflation-d]})\)
done
have fmap-eq: fmap-d = PRJ ('a \cdot 'f) oo cast-(\text{TC} ('f) \cdot t) oo emb
by \((\text{simp add: fmap-def coerce-cfun cast-TC cast-t prj-emb cfcomp1})\)
show \(?\)thesis
apply \((\text{simp add: fmap-eq isodefl-def cfun-eq-iff emb-prj})\)
apply \((\text{simp add: DEFL-app})\)
alineapply \((\text{simp add: cast-cast-below1 monofun-cfun t-below})\)

apply (simp add: cast-cast-below2 monofun-cfun t-below)
done

qed

lemma fmap-ID [simp]: fmap-ID = ID
apply (rule isodefl-DEFL-imp-ID)
apply (subst DEFL-app)
apply (rule isodefl-fmap)
apply (rule isodefl-ID-DEFL)
done

lemma fmap-ident [simp]: fmap · (Λ x. x) = ID
by (simp add: ID-def [symmetric])

lemma coerce-coerce-eq-fmapU-cast [coerce-simp]:
fixes xs :: udom·'f::functor
shows COERCΕ(udom·'f) · COERCΕ(udom·'f) · xs = fmapU · (cast·DEFL(udom·'f)) · xs
by (simp add: coerce-def emb-prj DEFL-app cast-TC)

lemma fmap-fmap:
fixes xs :: udom·'f::functor and g :: 'a → 'b and f :: 'b → 'c
shows fmap-f (fmap-g · xs) = fmap-(Λ x. f · (g · x)) · xs
unfolding fmap-def
by (simp add: coerce-simp)

lemma fmap-cfcomp: fmap · (f oo g) = fmap-f oo fmap-g
by (simp add: cfcomp1 fmap-fmap eta-cfun)

3.3 Derived properties of fmap

Other theorems about fmap can be derived using only the abstract functor laws.

lemma deflation-fmap:
deflation d =⇒ deflation (fmap·d)
apply (rule deflation.intro)
apply (simp add: fmap-fmap deflation.idem eta-cfun)
apply (subgoal-tac fmap·d·x ⊆ fmap·ID·x, simp)
apply (rule monofun-cfun-fun, rule monofun-cfun-arg)
apply (erule deflation.below-ID)
done

lemma ep-pair-fmap:
ep-pair e p =⇒ ep-pair (fmap·e) (fmap·p)
apply (rule ep-pair.intro)
apply (simp add: fmap-fmap ep-pair.e-inverse)
apply (simp add: fmap-fmap)
apply (rule-tac y=fmap·ID·y in below-trans)
apply (rule monofun-cfun-fun)
apply (rule monofun-cfun-arg)
apply (rule cfun-belowI, simp)
apply (erule ep-pair, e-p-below)
apply simp
done

lemma fmap-strict:
  fixes f :: 'a ⇒ 'b
  assumes f ·⊥ = ⊥ shows fmap·⊥ = (⊥::'b::functor)
proof (rule bottomI)
  have fmap·(⊥::'a::f) ⊑ fmap·(fmap·⊥·(⊥::'b::f))
    by (simp add: monofun-cfun)
  also have ... = fmap·(A x. f·(⊥·x))·(⊥::'b::f)
    by (simp add: fmap-fmap)
  also have ... ⊑ fmap·ID·⊥
    by (simp add: monofun-cfun assms del: fmap-ID)
  also have ... = ⊥
    by simp
  finally show fmap·⊥ ⊑ (⊥::'b::functor).
qed

3.4 Proving that fmap-coerce = coerc

lemma fmapU-cast-eq:
  fmapU·(cast·A) =
  PRJ(udom·f) oo cast·(TC(f::functor)·A) oo emb
by (subst cast-TC, rule cfun-eqI, simp)

lemma fmapU-cast-DEFL:
  fmapU·(cast·DEFL(a)) =
  PRJ(udom·f) oo cast·DEFL(a·f::functor) oo emb
by (simp add: fmapU-cast-DEFL-app)

lemma coerc-functor: COERCÉ(a·f, b·f::functor) = fmap-coerce
apply (rule cfun-eqI, rename-tac xs)
apply (simp add: fmap-def coerc-cfun)
apply (simp add: coerc-def)
apply (simp add: cfcomp1)
apply (simp only: emb-prj)
apply (subst fmapU-fmapU [symmetric])
apply (simp add: fmapU-cast-DEFL)
apply (simp add: emb-prj)
apply (simp add: cast-cast-below1 cast-cast-below2)
done

3.5 Lemmas for reasoning about coercion

lemma fmapU-cast-coerce [coerce-simp]:
  fixes m :: 'a·f::functor
\[ \text{shows } \text{fmapU} \cdot (\text{cast-DEFL}'a) \cdot (\text{COERCE}'a \cdot 'f, \text{udom}'f) \cdot m = \text{COERCE}'a \cdot 'f, \text{udom}'f \cdot m \]
by \(\text{simp add: coerce-functor cast-DEFL fmapU-eq-fmap fmap-fmap eta-cfun}\)

**lemma** `coerce-fmap` [coerce-simp]:
\[ \text{fixes } xs :: 'a'f::\text{functor and } f :: 'a \to 'b} \]
\[ \text{shows } \text{COERCE}'b \cdot 'f, 'c'f \cdot (\text{fmap} \cdot f \cdot xs) = \text{fmap} \cdot (\Lambda x. \text{COERCE}'b, 'c \cdot (f \cdot x)) \cdot xs \]
by \(\text{simp add: coerce-functor fmap-fmap}\)

**lemma** `fmap-coerce` [coerce-simp]:
\[ \text{fixes } xs :: 'a'f::\text{functor and } f :: 'b \to 'c} \]
\[ \text{shows } \text{fmap} \cdot f \cdot (\text{COERCE}'a \cdot 'f, 'b'f) \cdot xs = \text{fmap} \cdot (\Lambda x. f \cdot (\text{COERCE}'a, 'b) \cdot x) \cdot xs \]
by \(\text{simp add: coerce-functor fmap-fmap}\)

### 3.6 Configuration of Domain package

We make various theorem declarations to enable Domain package definitions that involve `tycon` application.

**setup** `{Domain-Take-Proofs.add-rec-type @\{type-name app\}, [true, false]}\)

**declare** `DEFL-app` [domain-defl-simps]
**declare** `fmap-ID` [domain-map-ID]
**declare** `deflation-fmap` [domain-deflation]
**declare** `isodefl-fmap` [domain-isodefl]

### 3.7 Configuration of the Tycon package

We now set up a new type definition command, which is used for defining new `tycon` instances. The `tycondef` command is implemented using much of the same code as the Domain package, and supports a similar input syntax. It automatically generates a `prefunctor` instance for each new type. (The user must provide a proof of the composition law to obtain a `functor` class instance.)

**ML-file** `{tycondef.ML}\)

end

### 4 Monad Class

theory `Monad`
imports `Functor`
begin

#### 4.1 Class definition

In Haskell, class `Monad` is defined as follows:
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

We formalize class monad in a manner similar to the functor class: We fix monomorphic versions of the class constants, replacing type variables with udom, and assume monomorphic versions of the class axioms.

Because the monad laws imply the composition rule for fmap, we declare prefunctor as the superclass, and separately prove a subclass relationship with functor.

class monad = prefunctor +
  fixes returnU :: udom -> udom’a::tycon
  fixes bindU :: udom’a -> (udom -> udom’a) -> udom’a
  assumes fmapU-eq-bindU:
    \forall f xs. fmapU · f · xs = bindU · xs · (\Lambda x. returnU · (f · x))
  assumes bindU-returnU:
    \forall x. bindU · (returnU · x) · f = f · x
  assumes bindU-bindU:
    \forall xs f g. bindU · (bindU · xs · f) · g = bindU · xs · (\Lambda x. bindU · (f · x) · g)

instance monad \subseteq functor
proof
  fix f g :: udom -> udom and xs :: udom’a
  show fmapU · f · (fmapU · g · xs) = fmapU · (\Lambda x. f · (g · x)) · xs
    by (simp add: fmapU-eq-bindU bindU-returnU bindU-bindU)
qed

As with fmap, we define the polymorphic return and bind by coercion from the monomorphic returnU and bindU.

definition return :: 'a -> 'a’m::monad
  where return = coerce · (returnU :: udom -> udom’m)

definition bind :: 'a’m::monad -> ('a -> 'b’m) -> 'b’m
  where bind = coerce · (bindU :: udom’m -> -)

abbreviation bind-syn :: 'a’m::monad => ('a -> 'b’m) => 'b’m (infixl \# 55)
  where m \# f == bind · m · f

4.2 Naturality of bind and return

The three class axioms imply naturality properties of returnU and bindU, i.e., that both commute with fmapU.

lemma fmapU-returnU [coerce-simp]:
  fmapU · f · (returnU · x) = returnU · (f · x)
by (simp add: fmapU-eq-bindU bindU-returnU)
lemma \texttt{fmapU-bindU} [coerce-simp]:
\[
fmapU \cdot f \cdot (\text{bindU} \cdot m \cdot k) = \text{bindU} \cdot m \cdot (\Lambda x \cdot \text{fmapU} \cdot f \cdot (k \cdot x))
\]
by (simp add: \texttt{fmapU-eq-bindU bindU-bindU})

lemma \texttt{bindU-fmapU}:
\[
\text{bindU} \cdot (\text{fmapU} \cdot f \cdot xs) \cdot k = \text{bindU} \cdot xs \cdot (\Lambda x \cdot k \cdot (f \cdot x))
\]
by (simp add: \texttt{fmapU-eq-bindU bindU-returnU bindU-bindU})

4.3 Polymorphic versions of class assumptions

lemma \texttt{Monad-fmap}:
\[
\text{fixes } \text{xs} :: 'a \cdot 'm::monad \text{ and } f :: 'a \rightarrow 'b
\wedge \text{shows } \text{fmap} \cdot f \cdot \text{xs} = \text{xs} \gg (\Lambda x \cdot \text{return} \cdot (f \cdot x))
\]
unfolding bind-def return-def
by (simp add: \texttt{coerce-simp fmapU-eq-bindU bindU-returnU})

lemma \texttt{Monad-left-unit} [simp]:
\[
(\text{return} \cdot x \gg f) = (f \cdot x)
\]
unfolding bind-def
by (simp add: \texttt{coerce-simp bindU-returnU})

lemma \texttt{bind-bind}:
\[
\text{fixes } m :: 'a \cdot 'm::monad
\wedge \text{shows } (m \gg f) \gg g = (m \gg (\Lambda x \cdot f \cdot x \gg g))
\]
unfolding bind-def
by (simp add: \texttt{coerce-simp bindU-bindU})

4.4 Derived rules

The following properties can be derived using only the abstract monad laws.

lemma \texttt{Monad-right-unit} [simp]:
\[
(m \gg \text{return}) = m
\]
apply (subgoal-tac \texttt{fmap-ID} \cdot m = m)
apply (simp only: monad-fmap)
apply (simp add: \texttt{eta-cfun})
apply simp
done

lemma \texttt{fmap-return}:
\[
\text{fmap} \cdot f \cdot (\text{return} \cdot x) = \text{return} \cdot (f \cdot x)
\]
by (simp add: monad-fmap)

lemma \texttt{fmap-bind}:
\[
\text{fmap} \cdot f \cdot (\text{bind} \cdot xs \cdot k) = \text{bind} \cdot xs \cdot (\Lambda x \cdot \text{fmap} \cdot f \cdot (k \cdot x))
\]
by (simp add: monad-fmap bind-bind)

lemma \texttt{bind-fmap}:
\[
\text{bind} \cdot (\text{fmap} \cdot f \cdot xs) \cdot k = \text{bind} \cdot xs \cdot (\Lambda x \cdot k \cdot (f \cdot x))
\]
by (simp add: monad-fmap bind-bind)

Bind is strict in its first argument, if its second argument is a strict function.

lemma \texttt{bind-strict}:
\[
\text{assumes } k \cdot \bot = \bot \text{ shows } \bot \gg k = \bot
\]
proof –

have \( \bot \gg k \gg \text{return} \gg \bot \gg k \)
  by (intro monofun-cfun below-refl minimal)

thus \( \bot \gg k = \bot \)
  by (simp add: assms)

qed

lemma congruent-bind:
  \( (\forall m. m \gg k1 = m \gg k2) = (k1 = k2) \)
apply (safe, rule cfun-eqI)
apply (drule-tac x = \text{return} \cdot x in spec, simp)
done

4.5 Laws for join

definition join :: \( \mathcal{A}' \cdot m \cdot \mathcal{A}' \cdot m \rightarrow \mathcal{A}' \cdot m \cdot \mathcal{A}' \cdot m \) where
  join \( \equiv \Lambda m. m \gg (\Lambda x. x) \)

lemma join-fmap-fmap: join \( \cdot (\text{fmap} \cdot (\text{fmap} \cdot f) \cdot \text{xs}) = \text{fmap} \cdot f \cdot (\text{join} \cdot \text{xs}) \)
by (simp add: join-def monad-fmap bind-bind)

lemma join-return: join \( \cdot (\text{return} \cdot \text{xs}) = \text{xs} \)
by (simp add: join-def)

lemma join-fmap-return: join \( \cdot (\text{fmap} \cdot \text{return} \cdot \text{xs}) = \text{xs} \)
by (simp add: join-def monad-fmap eta-cfun bind-bind)

lemma join-fmap-join: join \( \cdot (\text{fmap} \cdot \text{f} \cdot \text{join} \cdot \text{xs}) = \text{join} \cdot (\text{join} \cdot \text{xs}) \)
by (simp add: join-def monad-fmap bind-bind)

lemma bind-def2: m \( \gg k = \text{join} \cdot (\text{fmap} \cdot k \cdot m) \)
by (simp add: join-def monad-fmap eta-cfun bind-bind)

4.6 Equivalence of monad laws and fmap/join laws

lemma (\text{return} \cdot f \gg \text{x} = (f \cdot \text{x})
by (simp only: bind-def2 fmap-return join-return)

lemma (m \gg \text{return} = m
by (simp only: bind-def2 join-fmap-return)

lemma ((m \gg f \gg g) = (m \gg (\Lambda x. \text{f} \cdot \text{x} \gg g))
apply (simp only: bind-def2)
apply (subgoal-tac join \( \cdot (\text{fmap} \cdot g \cdot (\text{join} \cdot (\text{fmap} \cdot f \cdot \text{m})))) = 
  \text{join} \cdot (\text{fmap} \cdot (\text{fmap} \cdot g \cdot (\text{fmap} \cdot f \cdot \text{m}))))
apply (simp add: fmap-fmap)
apply (simp add: join-fmap-join join-fmap-fmap)
done

17
4.7 Simplification of coercions

We configure rewrite rules that push coercions inwards, and reduce them to coercions on simpler types.

**lemma** **coerce-return** [coerce-simp]:

\[
\text{COERCE}(\text{'a-m}',\text{'b-m::monad}) \cdot (\text{return} \cdot x) = \text{return} \cdot (\text{COERCE}(\text{'a','b} \cdot x)
\]

by (simp add: coerce-functor fmap-return)

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

\[
\text{fixes } m :: \text{'a-m::monad} \text{ and } k :: \text{'a} \rightarrow \text{'b-m}
\]

shows \[
\text{COERCE}(\text{'b-m'},\text{'c-m}) \cdot (m \gg k) = m \gg (\Lambda x. \text{COERCE}(\text{'b-m,'c-m}) \cdot (k \cdot x))
\]

by (simp add: coerce-functor bind-fmap)

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

\[
\text{fixes } m :: \text{'a-m::monad} \text{ and } k :: \text{'b} \rightarrow \text{'c-m}
\]

shows \[
\text{COERCE}(\text{'a-m},\text{'b-m}) \cdot m \gg k = m \gg (\Lambda x. \text{COERCE}(\text{'a,'b} \cdot x))
\]

by (simp add: coerce-functor bind-fmap)

end

5 Monad-Zero Class

**theory** Monad-Zero

**imports** Monad

**begin**

class zeroU = tycon +

fixes zeroU :: udom::'a::tycon

class functor-zero = zeroU + functor +

assumes fmapU-zeroU [coerce-simp]:

\[
fmapU \cdot f \cdot zeroU = zeroU
\]

class monad-zero = zeroU + monad +

assumes bindU-zeroU:

\[
bindU \cdot zeroU \cdot f = zeroU
\]

instance monad-zero \subseteq functor-zero

**proof**

fix f show \[
fmapU \cdot f \cdot zeroU = (zeroU :: udom::'a)
\]

unfolding fmapU-eq-bindU

by (rule bindU-zeroU)

qed

definition fzero :: 'a::'f::functor-zero

where fzero = coerce-(zeroU :: udom::'f)

**lemma** fmap-fzero:
fmap \cdot f \cdot \text{fzero} :: \text{functor-zero} = \text{fzero} :: \text{b'f}

unfolding fmap-def fzero-def
by (simp add: coerce-simp)

abbreviation mzero :: 'a :: monad-zero
where mzero = fzero

lemmas mzero-def = fzero-def [where 'f='m::monad-zero] for f
lemmas fmap-mzero = fmap-fzero [where 'f='m::monad-zero] for f

lemma bindU-eq-bind: bindU = bind
unfolding bind-def by simp

lemma bind-mzero:
bind \cdot (fzero :: 'a::monad-zero) \cdot k = (mzero :: 'b'\ m)
unfolding bind-def mzero-def
by (simp add: coerce-simp bindU-zeroU)

end

6 Monad-Plus Class

theory Monad-Plus
imports Monad
begin

hide-const (open) Fixrec.mplus

class plusU = tycon +
fixes plusU :: udom :: 'a \to udom :: 'a \to udom :: 'a::tycon

class functor-plus = plusU + functor +
assumes fmapU-plusU [coerce-simp]:
fmapU \cdot f \cdot (\text{plusU} \cdot a \cdot b) = \text{plusU} \cdot (\text{fmapU} \cdot f \cdot a) \cdot (\text{fmapU} \cdot f \cdot b)
assumes plusU-assoc:
\text{plusU} \cdot (\text{plusU} \cdot a \cdot b) \cdot c = \text{plusU} \cdot a \cdot (\text{plusU} \cdot b \cdot c)

class monad-plus = plusU + monad +
assumes bindU-plusU:
\text{bindU} \cdot (\text{plusU} \cdot x \cdot y) \cdot k = \text{plusU} \cdot (\text{bindU} \cdot x \cdot k) \cdot (\text{bindU} \cdot y \cdot k)
assumes plusU-assoc':
\text{plusU} \cdot (\text{plusU} \cdot a \cdot b) \cdot c = \text{plusU} \cdot a \cdot (\text{plusU} \cdot b \cdot c)

instance monad-plus \subseteq functor-plus
by standard (simp-all only: fmapU-eq-bindU bindU-plusU plusU-assoc')

definition fplus :: 'a :: functor-plus \to 'a :: f \to 'a :: f
where fplus = coerce \cdot (\text{plusU} :: udom :: 'f \to -)
lemma fmap-fplus:
fixes f :: 'a → 'b and a b :: 'a::functor-plus
shows fmap f · (fplus a · b) = fplus · (fmap f · a) · (fmap f · b)
unfolding fmap-def fplus-def
by (simp add: coerce-simp)

lemma fplus-assoc:
fixes a b c :: 'a::functor-plus
shows fplus · (fplus · a · b) · c = fplus · a · (fplus · b · c)
unfolding fplus-def
by (simp add: coerce-simp plusU-assoc)

abbreviation mplus :: 'a::monad-plus → 'a::monad-plus → 'a::monad-plus
where mplus ≡ fplus

lemmas mplus-def = fplus-def [where 'f = 'm::monad-plus for f]
lemmas fmap-mplus = fmap-fplus [where 'f = 'm::monad-plus for f]
lemmas mplus-assoc = fplus-assoc [where 'f = 'm::monad-plus for f]

lemma bind-mplus:
fixes a b :: 'a::monad-plus
shows bind · (mplus a · b) · k = mplus · (bind a · k) · (bind b · k)
unfolding bind-def mplus-def
by (simp add: coerce-simp bindU-plusU)

lemma join-mplus:
fixes xss yss :: (a::monad-plus) · (a::monad-plus)
shows join · (mplus xss · yss) = mplus · (join xss) · (join yss)
by (simp add: join-def bind-mplus)
end

7 Monad-Zero-Plus Class

theory Monad-Zero-Plus
imports Monad-Zero Monad-Plus
begin

hide-const (open) Fixrec.mplus

class functor-zero-plus = functor-zero + functor-plus +
assumes plusU-zeroU-left:
plusU · zeroU · m = m
assumes plusU-zeroU-right:
plusU · m · zeroU = m

class monad-zero-plus = monad-zero + monad-plus + functor-zero-plus

lemma fplus-fzero-left:
8 Lazy list monad

theory Lazy-List-Monad 
imports Monad-Zero-Plus
begin

To illustrate the general process of defining a new type constructor, we formalize the datatype of lazy lists. Below are the Haskell datatype definition and class instances.

data List a = Nil | Cons a (List a)

instance Functor List where
    fmap f Nil = Nil
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Monad List where
    return x = Cons x Nil
    Nil >>= k = Nil
    Cons x xs >>= k = mplus (k x) (xs >>= k)

instance MonadZero List where
    mzero = Nil

instance MonadPlus List where
    mplus Nil ys = ys
    mplus (Cons x xs) ys = Cons x (mplus xs ys)
8.1 Type definition

The first step is to register the datatype definition with `tycondef`.

```
tycondef 'a·llist = LNil | LCons (lazy 'a) (lazy 'a·llist)
```

The `tycondef` command generates lots of theorems automatically, but there are a few more involving `coerce` and `fmapU` that we still need to prove manually. These proofs could be automated in a later version of `tycondef`.

```
lemma coercel-list-abs [simp]: coercel·(list-abs·x) = list-abs·(coerce·x)
apply (simp add: list-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-llist)
done

lemma coercel-LNil [simp]: coercel·LNil = LNil
unfolding LNil-def by simp

lemma coercel-LCons [simp]: coercel·(LCons·x·xs) = LCons·(coerce·x)·(coerce·xs)
unfolding LCons-def by simp

lemma fmapU-list-simps [simp]:
fmapU·f·(⊥::udom·llist) = ⊥
fmapU·f·LNil = LNil
fmapU·f·(LCons·x·xs) = LCons·(f·x)·(fmapU·f·xs)
unfolding fmapU-list-def llist-map-def
apply (subst fix-eq, simp)
apply (subst fix-eq, simp add: LNil-def)
apply (subst fix-eq, simp add: LCons-def)
done
```

8.2 Class instances

The `tycondef` command defines `fmapU` for us and proves a `prefunctor` class instance automatically. For the `functor` instance we only need to prove the composition law, which we can do by induction.

```
instance llist :: functor
proof
  fix f g and xs :: udom·llist
  show fmapU·f·(fmapU·g·xs) = fmapU·(Λ x. f·(g·x))·xs
    by (induct xs rule: llist.induct) simp-all
qed
```

For the other class instances, we need to provide definitions for a few constants: `returnU`, `bindU` `zeroU`, and `plusU`. We can use ordinary commands like `definition` and `fixrec` for this purpose. Finally we prove the class axioms, along with a few helper lemmas, using ordinary proof procedures like induction.
instantiation llist :: monad-zero-plus
begin

fixrec plusU-llist :: udom·llist → udom·llist → udom·llist
  where plusU-llist·LNil·ys = ys
  | plusU-llist·(LCons·x·xs)·ys = LCons·x·(plusU-llist·xs·ys)

lemma plusU-llist-strict [simp]: plusU·⊥·ys = (⊥::udom·llist)
by fixrec-simp

fixrec bindU-llist :: udom·llist → (udom → udom·llist) → udom·llist
  where bindU-llist·LNil·k = LNil
  | bindU-llist·(LCons·x·xs)·k = plusU·(k·x)·(bindU-llist·xs·k)

lemma bindU-llist-strict [simp]: bindU·⊥·k = (⊥::udom·llist)
by fixrec-simp

definition zeroU-llist-def:
  zeroU = LNil

definition returnU-llist-def:
  returnU = (Λ x. LCons·x·LNil)

lemma plusU-LNil-right: plusU·xs·LNil = xs
by (induct xs rule: llist.induct) simp-all

lemma plusU-llist-assoc:
  fixes xs ys zs :: udom·llist
  shows plusU·(plusU·xs·ys)·zs = plusU·xs·(plusU·ys·zs)
by (induct xs rule: llist.induct) simp-all

lemma bindU-plusU-llist:
  fixes xs ys :: udom·llist shows
  bindU·(plusU·xs·ys)·f = plusU·(bindU·xs·f)·(bindU·ys·f)
by (induct xs rule: llist.induct) (simp-all add: plusU-llist-assoc)

instance proof
  fix x :: udom
  fix f :: udom → udom
  fix h k :: udom → udom·llist
  fix xs ys zs :: udom·llist
show fmapU·f·xs = bindU·xs·(Λ x. returnU·(f·x))
  by (induct xs rule: llist.induct, simp-all add: returnU-llist-def)
show bindU·(returnU·x)·k = k·x
  by (simp add: returnU-llist-def plusU-LNil-right)
show bindU·(bindU·xs·h)·k = bindU·xs·(Λ x. bindU·(h·x)·k)
  by (induct xs rule: llist.induct)
    (simp-all add: bindU-plusU-llist)
show bindU·(plusU·xs·ys)·k = plusU·(bindU·xs·k)·(bindU·ys·k)
by (induct zs rule: llist.induct)
(simp-all add: plusU-list-assoc)
show plusU·(plusU·xs·ys)·zs = plusU·xs·(plusU·ys·zs)
by (rule plusU-list-assoc)
show bindU·zeroU·k = zeroU
by (simp add: zeroU-llist-def)
show fmapU·f·(plusU·xs·ys) = plusU·(fmapU·f·xs)·(fmapU·f·ys)
by (induct xs rule: llist.induct) simp-all
show fmapU·f·zeroU = (zeroU :: udom-llist)
by (simp add: zeroU-llist-def)
show plusU·zeroU·xs = xs
by (simp add: zeroU-llist-def)
show plusU·xs·zeroU = xs
by (simp add: zeroU-llist-def plusU-LNil-right)
qed

end

8.3 Transfer properties to polymorphic versions

After proving the class instances, there is still one more step: We must transfer all the list-specific lemmas about the monomorphic constants (e.g., fmapU and bindU) to the corresponding polymorphic constants (fmap and bind). These lemmas primarily consist of the defining equations for each constant. The polymorphic constants are defined using coerce, so the proofs proceed by unfolding the definitions and simplifying with the coerce-simp rules.

lemma fmap-llist-simps [simp]:
fmap·f·⊥·a·llist = ⊥
fmap·f·LNil = LNil
fmap·f·(LCons·x·xs) = LCons·(f·x)·(fmap·f·xs)

unfolding fmap-def by simp-all

lemma mplus-llist-simps [simp]:
mplus·(⊥·a·llist)·ys = ⊥
mplus·LNil·ys = ys
mplus·(LCons·x·xs)·ys = LCons·x·(mplus·xs·ys)

unfolding mplus-def by simp-all

lemma bind-llist-simps [simp]:
bind·(⊥·a·llist)·f = ⊥
bind·LNil·f = LNil
bind·(LCons·x·xs)·f = mplus·(f·x)·(bind·xs·f)

unfolding bind-def mplus-def
by (simp-all add: coerce-simp)

lemma return-llist-def:
return = (Λ x. LCons·x·LNil)
unfolding return-def returnU-list-def
by (simp add: coerce-simp)

lemma mzero-list-def:
mzero = LNil
unfolding mzero-def zeroU-list-def
by simp

lemma join-list-simps [simp]:
join·(⊥·'a·list·list) = ⊥
join·LNil = LNil
join·(LCons·xs·xss) = maplus·xs·(join·xss)
unfolding join-def by simp-all

end

9 Maybe monad

theory Maybe-Monad
imports Monad-Zero-Plus
begin

9.1 Type definition

tycondef 'a·maybe = Nothing | Just (lazy 'a)

lemma coerce-maybe-abs [simp]:
coerce·(maybe-abs·x) = maybe-abs·(coerce·x)
apply (simp add: maybe-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-maybe)
done

lemma coerce-Nothing [simp]:
coerce·Nothing = Nothing
unfolding Nothing-def by simp

lemma coerce-Just [simp]:
coerce·(Just·x) = Just·(coerce·x)
unfolding Just-def by simp

lemma fmapU-maybe-simps [simp]:
fmapU·f·(⊥·udom-maybe) = ⊥
fmapU·f·Nothing = Nothing
fmapU·f·(Just·x) = Just·(f·x)
unfolding fmapU-maybe-def maybe-map-def fix-const
apply simp
apply (simp add: Nothing-def)
apply (simp add: Just-def)
done
9.2 Class instance proofs

instance maybe :: functor
apply standard
apply (induct-tac xs rule: maybe.induct, simp-all)
done

instatiation maybe :: {functor-zero-plus, monad-zero}
begin

fixrec plusU-maybe :: udom·maybe → udom·maybe → udom·maybe
  where plusU-maybe·Nothing·ys = ys
         | plusU-maybe·(Just·x)·ys = Just·x

lemma plusU-maybe-strict [simp]: plusU·⊥·ys = (⊥::udom·maybe)
by fixrec-simp

fixrec bindU-maybe :: udom·maybe → (udom → udom·maybe) → udom·maybe
  where bindU-maybe·Nothing·k = Nothing
         | bindU-maybe·(Just·x)·k = k·x

lemma bindU-maybe-strict [simp]: bindU·⊥·k = (⊥::udom·maybe)
by fixrec-simp

definition zeroU-maybe-def:
  zeroU = Nothing

definition returnU-maybe-def:
  returnU = Just

lemma plusU-Nothing-right: plusU·xs·Nothing = xs
by (induct xs rule: maybe.induct) simp-all

lemma bindU-plusU-maybe:
  fixes xs ys :: udom·maybe shows
  bindU·(plusU·xs·ys)·f = plusU·(bindU·xs·f)·(bindU·ys·f)
apply (induct xs rule: maybe.induct)
apply simp-all
oops

instance proof
  fix x :: udom
  fix f :: udom → udom
  fix h k :: udom → udom·maybe
  fix xs ys zs :: udom·maybe
  show fmapU·f·xs = bindU·xs·(Λ x. returnU·(f·x))
    by (induct xs rule: maybe.induct, simp-all add: returnU-maybe-def)
  show bindU·(returnU·x)·k = k·x
    by (simp add: returnU-maybe-def plusU-Nothing-right)
  show bindU·(bindU·xs·h)·k = bindU·xs·(Λ x. bindU·(h·x)·k)
by (induct zs rule: maybe.induct) simp-all
show plusU·(plusU·xs·ys)·zs = plusU·xs·(plusU·ys·zs)
  by (induct xs rule: maybe.induct) simp-all
show bindU·zeroU·k = zeroU
  by (simp add: zeroU-maybe-def)
show fmapU·f·(plusU·xs·ys) = plusU·(fmapU·f·xs)·(fmapU·f·ys)
  by (induct xs rule: maybe.induct) simp-all
show fmapU·f·zeroU = (zeroU :: udom-maybe)
  by (simp add: zeroU-maybe-def)
show plusU·zeroU·xs = xs
  by (simp add: zeroU-maybe-def)
show plusU·xs·zeroU = xs
  by (simp add: zeroU-maybe-def plusU-Nothing-right)
qed

end

9.3 Transfer properties to polymorphic versions

lemma fmap-maybe-simps [simp]:
fmap·f·⊥ = ⊥
fmap·f·Nothing = Nothing
fmap·f·(Just·x) = Just·(f·x)
unfolding fmap-def by simp-all

lemma fplus-maybe-simps [simp]:
fplus·⊥ = ⊥
fplus·Nothing·ys = ys
fplus·(Just·x)·ys = Just·x
unfolding fplus-def by simp-all

lemma fplus-Nothing-right [simp]:
fplus·m·Nothing = m
by (simp add: fplus-def plusU-Nothing-right)

lemma bind-maybe-simps [simp]:
bind·⊥ = ⊥
bind·Nothing = Nothing
bind·(Just·x)·f = f·x
unfolding bind-def fplus-def by simp-all

lemma return-maybe-def: return = Just
unfolding return-def returnU-maybe-def
by (simp add: coerce-cfun cfcomp1 eta-cfun)

lemma mzero-maybe-def: mzero = Nothing
unfolding mzero-def zeroU-maybe-def
by simp
lemma join-maybe-simps [simp]:
join(⊥·a·maybe·maybe) = ⊥
join-Nothing = Nothing
join(Just·xs) = xs
unfolding join-def by simp

9.4 Maybe is not in monad-plus

The maybe type does not satisfy the law bind-mplus.

lemma maybe-counterexample1:
[a = Just·x; b = ⊥; k·x = Nothing]
⇒ fplus·a·b \cong k ≠ fplus·(a \cong k)·(b \cong k)
by simp

lemma maybe-counterexample2:
[a = Just·x; b = Just·y; k·x = Nothing; k·y = Just·z]
⇒ fplus·a·b \cong k ≠ fplus·(a \cong k)·(b \cong k)
by simp

end

10 Error monad

theory Error-Monad
imports Monad-Plus
begin

10.1 Type definition
tycondef 'a·e error = Err (lazy 'e) | Ok (lazy 'a)

lemma coerce-error-abs [simp]:
coerce·(error-abs·x) = error-abs·(coerce·x)
apply (simp add: error-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-error)
done

lemma coerce-Err [simp]:
coerce·(Err·x) = Err·(coerce·x)
unfolding Err-def by simp

lemma coerce-Ok [simp]:
coerce·(Ok·m) = Ok·(coerce·m)
unfolding Ok-def by simp

lemma fmapU-error-simps [simp]:
fmapU·f·(⊥·udom·'a error) = ⊥
fmapU·f·(Err·e) = Err·e
fmapU·f·(Ok·x) = Ok·(f·x)
unfolding fmapU-error-def error-map-def fix-const
apply simp
apply (simp add: Err-def)
apply (simp add: Ok-def)
done

10.2 Monad class instance

instantiation error :: (domain) {monad, functor-plus}
begin

definition returnU = Ok

fixrec bindU-error :: udom 'a error → (udom → udom 'a error) → udom 'a error
  where bindU-error (Err·e)·f = Err·e
       | bindU-error (Ok·x)·f = f·x

lemma bindU-error-strict [simp]: bindU·⊥·k = (⊥::udom 'a error)
by fixrec-simp

fixrec plusU-error :: udom 'a error → udom 'a error → udom 'a error
  where plusU-error (Err·e)·f = f
       | plusU-error (Ok·x)·f = Ok·x

lemma plusU-error-strict [simp]: plusU·(⊥::udom 'a error) = ⊥
by fixrec-simp

instance proof

  fix f g :: udom → udom and r :: udom 'a error
  show fmapU·f·(fmapU·g·r) = fmapU·(Λ x. f·(g·x))·r
    by (induct r rule: error.induct) simp-all

next

  fix f :: udom → udom and r :: udom 'a error
  show fmapU·f·r = bindU·r·(Λ x. returnU·(f·x))
    by (induct r rule: error.induct)
    (simp-all add: returnU-error-def)

next

  fix f :: udom → udom 'a error and x :: udom
  show bindU·(returnU·x)·f = f·x
    by (simp add: returnU-error-def)

next

  fix r :: udom 'a error and f g :: udom → udom 'a error
  show bindU·(bindU·r·f)·g = bindU·r·(Λ x. bindU·(f·x)·g)
    by (induct r rule: error.induct)
    simp-all

next

  fix f :: udom → udom and a b :: udom 'a error
  show fmapU·f·(plusU·a·b) = plusU·(fmapU·f·a)·(fmapU·f·b)
    by (induct a rule: error.induct) simp-all
next
fix a b c :: udom · 'a error
show plusU · (plusU · a · b) · c = plusU · a · (plusU · b · c)
by (induct a rule: error.induct) simp-all
qed
end

10.3 Transfer properties to polymorphic versions

lemma fmap-error-simps [simp]:
  fmap f (⊥ :: 'a · 'e error) = ⊥
  fmap f (Err · e :: 'a · 'e error) = Err · e
  fmap f (Ok · x :: 'a · 'e error) = Ok · (f · x)
unfolding fmap-def [where 'f = 'e error]
by (simp-all add: coerce-simp)

lemma return-error-def: return = Ok
unfolding return-def returnU-error-def
by (simp add: coerce-simp eta-cfun)

lemma bind-error-simps [simp]:
  bind (⊥ :: 'a · 'e error) · f = ⊥
  bind (Err · e :: 'a · 'e error) · f = Err · e
  bind (Ok · x :: 'a · 'e error) · f = f · x
unfolding bind-def
by (simp-all add: coerce-simp)

lemma join-error-simps [simp]:
  join ⊥ = (⊥ :: 'a · 'e error)
  join (Err · e) = Err · e
  join (Ok · x) = x
unfolding join-def by simp-all

lemma fplus-error-simps [simp]:
  fplus · ⊥ · r = (⊥ :: 'a · 'e error)
  fplus · (Err · e) · r = r
  fplus · (Ok · x) · r = Ok · x
unfolding fplus-def
by (simp-all add: coerce-simp)

end

11 Writer monad

theory Writer-Monad
imports Monad
begin
11.1  Monoid class

```haskell
class monoid = domain +
fixes mempty :: 'a
fixes mappend :: 'a → 'a → 'a
assumes mempty-left: ∀ys. mappend · mempty · ys = ys
assumes mempty-right: ∀xs. mappend · xs · mempty = xs
assumes mappend-assoc:
  ∀xs ys zs. mappend · (mappend · xs · ys) · zs = mappend · xs · (mappend · ys · zs)
```

11.2  Writer monad type

Below is the standard Haskell definition of a writer monad type; it is an isomorphic copy of the lazy pair type (a, w).

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }
```

Since HOLCF does not have a pre-defined lazy pair type, we will base this formalization on an equivalent, more direct definition:

```haskell
data Writer w a = Writer w a
```

We can directly translate the above Haskell type definition using `tycondef`.

```haskell
tycondef 'a → 'w writer = Writer (lazy 'w) (lazy 'a)
```

```haskell
lemma coerce-writer-abs [simp]: coerce · (writer-abs · x) = writer-abs · (coerce · x)
apply (simp add: writer-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-writer)
done
```

```haskell
lemma coerce-Writer [simp]:
  coerce · (Reader · w · x) = Reader · (coerce · w) · (coerce · x)
unfolding Writer-def by simp
```

```haskell
lemma fmapU-writer-simps [simp]:
  fmapU · f · (⊥ :: udom · 'w writer) = ⊥
  fmapU · f · (Writer · w · x) = Writer · w · (f · x)
unfolding fmapU-writer-def writer-map-def fix-const
apply simp
apply (simp add: Writer-def)
done
```

11.3  Class instance proofs

```haskell
instance writer :: (domain) functor
proof
  fix f g :: udom → udom and xs :: udom · 'a writer
```
\[ \text{show } \text{fmapU} \cdot f \cdot (\text{fmapU} \cdot g \cdot \text{xs}) = \text{fmapU} \cdot (\lambda x. f \cdot (g \cdot x)) \cdot \text{xs} \]
\[ \text{by (induct xs rule: writer.induct) simp-all} \]
\[ \text{qed} \]

**instantiation** **writer** :: (monoid) **Monad**

**begin**

**fixrec** **bindU-writer** ::
\[ \text{udom} \cdot 'a \text{ writer} \rightarrow (\text{udom} \rightarrow \text{udom} \cdot 'a \text{ writer}) \rightarrow \text{udom} \cdot 'a \text{ writer} \]

**where** **bindU-writer** \((\text{Writer} \cdot w \cdot x) \cdot f = \)
\[ \text{(case } f \cdot x \text{ of } \text{Writer} \cdot w' \cdot y \Rightarrow \text{Writer} \cdot (\text{mappend} \cdot w \cdot w') \cdot y) \]

**lemma** **bindU-writer-strict** [simp]:
\[ \text{bindU} \cdot \bot \cdot k = (\bot :: \text{udom} \cdot 'a \text{ writer}) \]

**by fixrec-simp**

**definition**
\[ \text{returnU} = \text{Writer} \cdot \text{mempty} \]

**instance proof**

**fix** \( f :: \text{udom} \rightarrow \text{udom} \) and \( m :: \text{udom} \cdot 'a \text{ writer} \)

**show** \( \text{fmapU} \cdot f \cdot m = \text{bindU} \cdot m \cdot (\lambda x. \text{returnU} \cdot (f \cdot x)) \)

**by (induct m rule: writer.induct) (simp-all add: returnU-writer-def mempty-right)**

**next**

**fix** \( f :: \text{udom} \rightarrow \text{udom} \cdot 'a \text{ writer} \) and \( x :: \text{udom} \)

**show** \( \text{bindU} \cdot (\text{returnU} \cdot x) \cdot f = f \cdot x \)

**by (cases x rule: writer.exhaust) (simp-all add: returnU-writer-def mempty-left)**

**next**

**fix** \( m :: \text{udom} \cdot 'a \text{ writer} \) and \( f \cdot g :: \text{udom} \rightarrow \text{udom} \cdot 'a \text{ writer} \)

**show** \( \text{bindU} \cdot (\text{bindU} \cdot f \cdot m) \cdot g = \text{bindU} \cdot m \cdot (\lambda x. \text{bindU} \cdot (f \cdot x) \cdot g) \)

**apply (induct m rule: writer.induct, simp)**

**apply (case-tac f \cdot a rule: writer.exhaust, simp)**

**apply (case-tac g \cdot aa rule: writer.exhaust, simp)**

**apply (simp add: mappend-assoc)**

**done**

**qed**

**end**

---

### 11.4 Transfer properties to polymorphic versions

**lemma** **fmap-writer-simps** [simp]:
\[ \text{fmapU} \cdot f \cdot (\bot :: 'a \cdot 'w \text{ writer}) = \bot \]
\[ \text{fmapU} \cdot (\text{Writer} \cdot w \cdot x :: 'a \cdot 'w \text{ writer}) = \text{Writer} \cdot (w \cdot (f \cdot x)) \]

**unfolding** **fmap-def** [where \( f = 'w \text{ writer} \)]

**by (simp-all add: coerce-simp)**

**lemma** **return-writer-def**: \( \text{return} = \text{Writer} \cdot \text{mempty} \)
unfolding return-def returnU-writer-def
by (simp add: coerce-simp eta-cfun)

lemma bind-writer-simps [simp];
bind-(⊥ :: 'a::monoid writer)f = ⊥
bind-(Writer·w·x :: 'a::monoid writer)k =
(case k·x of Writer·w·y ⇒ Writer·(mappend·w·w')·y)

unfolding bind-def
apply (simp add: coerce-simp)
apply (cases k·x rule: writer.exhaust)
apply (simp-all add: coerce-simp)
done

lemma join-writer-simps [simp];
join·⊥ = (⊥ :: 'a::monoid writer)
join·(Writer·w·(Writer·w·x)) = Writer·(mappend·w·w')·x

unfolding join-def by simp-all

11.5 Extra operations

definition tell :: 'w ⇒ unit·('w::monoid writer)
where tell = (Λ w. Writer·w·())

end

12 Binary tree monad

theory Binary-Tree-Monad
imports Monad
begin

12.1 Type definition

tycondef 'a-btree =
Leaf (lazy 'a) | Node (lazy 'a-btree) (lazy 'a-btree)

lemma coerce-btree-abs [simp]: coerce·(btree-abs·x) = btree-abs·(coerce·x)
apply (simp add: btree-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-btree)
done

lemma coerce-Leaf [simp]: coerce·(Leaf·x) = Leaf·(coerce·x)
unfolding Leaf-def by simp

lemma coerce-Node [simp]: coerce·(Node·xs·ys) = Node·(coerce·xs)·(coerce·ys)
unfolding Node-def by simp

lemma fmapU-btree-simps [simp]:
fmapU·f·(⊥::udom-btree) = ⊥
\[
\begin{align*}
\text{fmapU} \cdot f \cdot (\text{Leaf} \cdot x) &= \text{Leaf} \cdot (f \cdot x) \\
\text{fmapU} \cdot f \cdot (\text{Node} \cdot xs \cdot ys) &= \text{Node} \cdot (\text{fmapU} \cdot f \cdot xs) \cdot (\text{fmapU} \cdot f \cdot ys)
\end{align*}
\]

unfolding \text{fmapU-btree-def} \text{ btree-map-def}

apply (subst fix-eq, simp)
apply (subst fix-eq, simp add: Leaf-def)
apply (subst fix-eq, simp add: Node-def)
done

12.2 Class instance proofs

instance \text{btree} :: \text{functor}
apply standard
apply (induct-tac xs rule: \text{btree.induct}, simp-all)
done

instantiation \text{btree} :: \text{monad}
begin

definition \text{returnU} = \text{Leaf}

fixrec \text{bindU-btree} :: \text{udom} \cdot \text{btree} \rightarrow (\text{udom} \rightarrow \text{udom} \cdot \text{btree}) \rightarrow \text{udom} \cdot \text{btree}
where \text{bindU-btree} \cdot (\text{Leaf} \cdot x) \cdot k = k \cdot x
\mid \text{bindU-btree} \cdot (\text{Node} \cdot xs \cdot ys) \cdot k =
\quad \text{Node} \cdot (\text{bindU-btree} \cdot xs \cdot k) \cdot (\text{bindU-btree} \cdot ys \cdot k)

lemma \text{bindU-btree-strict [simp]: bindU} \cdot \bot \cdot k = (\bot :: \text{udom} \cdot \text{btree})
by \text{fixrec-simp}

instance proof
\fix x :: \text{udom}
\fix f :: \text{udom} \rightarrow \text{udom}
\fix h k :: \text{udom} \rightarrow \text{udom} \cdot \text{btree}
\fix xs :: \text{udom} \cdot \text{btree}
show \text{fmapU} \cdot f \cdot xs = \text{bindU} \cdot xs \cdot (\Lambda x. \text{returnU} \cdot (f \cdot x))
\quad by (induct xs rule: \text{btree.induct}, simp-all add: \text{returnU-btree-def})
show \text{bindU} \cdot (\text{returnU} \cdot x) \cdot k = k \cdot x
\quad by (simp add: \text{returnU-btree-def})
show \text{bindU} \cdot (\text{bindU} \cdot xs \cdot h) \cdot k = \text{bindU} \cdot xs \cdot (\Lambda x. \text{bindU} \cdot (h \cdot x) \cdot k)
\quad by (induct xs rule: \text{btree.induct}) simp-all
qed

end

12.3 Transfer properties to polymorphic versions

lemma \text{fmap-btree-simps [simp]};
\text{fmap} \cdot f \cdot (\bot :: 'a \cdot \text{btree}) = \bot
\text{fmap} \cdot f \cdot (\text{Leaf} \cdot x) = \text{Leaf} \cdot (f \cdot x)
\text{fmap} \cdot f \cdot (\text{Node} \cdot xs \cdot ys) = \text{Node} \cdot (\text{fmap} \cdot f \cdot xs) \cdot (\text{fmap} \cdot f \cdot ys)
unfolding \textit{fmap-def} by simp-all

\textbf{lemma} \textit{bind-btree-simps} [simp]:
\begin{align*}
bind(\bot \cdot 'a \cdot \text{btree}) \cdot k &= \bot \\
bind(\text{Leaf} \cdot x) \cdot k &= k \cdot x \\
bind(\text{Node} \cdot xs \cdot ys) \cdot k &= \text{Node}(\text{bind} \cdot xs \cdot k) \cdot (\text{bind} \cdot ys \cdot k)
\end{align*}

\textbf{unfolding} \textit{bind-def}

by (simp-all add: \textit{coerce-simp})

\textbf{lemma} \textit{return-btree-def}:
\begin{align*}
\text{return} &= \text{Leaf}
\end{align*}

\textbf{unfolding} \textit{return-def returnU-btree-def}

by (simp add: \textit{coerce-simp eta-cfun})

\textbf{lemma} \textit{join-btree-simps} [simp]:
\begin{align*}
\text{join}(\bot \cdot 'a \cdot \text{btree} \cdot \text{btree}) &= \bot \\
\text{join}(\text{Leaf} \cdot xs) &= xs \\
\text{join}(\text{Node} \cdot xss \cdot yss) &= \text{Node}(\text{join} \cdot xss) \cdot (\text{join} \cdot yss)
\end{align*}

\textbf{unfolding} \textit{join-def} by simp-all

end

13 Lift monad

theory Lift-Monad

imports Monad

begin

13.1 Type definition

\textbf{tycondef} 'a-lifted = Lifted (lazy 'a)

\textbf{lemma} \textit{coerce-lifted-abs} [simp]: \textit{coerce} \cdot (\textit{lifted-abs} \cdot x) = \textit{lifted-abs} \cdot (\textit{coerce} \cdot x)

apply (simp add: \textit{lifted-abs-def coerce-def})

apply (simp add: \textit{emb-prj-emb prj-emb-prj DEFL-eq-lifted})

done

\textbf{lemma} \textit{coerce-Lifted} [simp]: \textit{coerce} \cdot (\textit{Lifted} \cdot x) = \textit{Lifted} \cdot (\textit{coerce} \cdot x)

\textbf{unfolding} \textit{Lifted-def} by simp

\textbf{lemma} \textit{fmapU-lifted-simps} [simp]:
\begin{align*}
\text{fmapU} \cdot f \cdot (\bot \cdot \text{udom-lifted}) &= \bot \\
\text{fmapU} \cdot f \cdot (\text{Lifted} \cdot x) &= \text{Lifted} \cdot (f \cdot x)
\end{align*}

\textbf{unfolding} \textit{fmapU-lifted-def lifted-map-def fix-const}

apply simp

apply (simp add: Lifted-def)

done
13.2 Class instance proofs

instance lifted :: functor
  by standard (induct-tac xs rule: lifted.induct, simp-all)

instantiation lifted :: monad
begin

fixrec bindU-lifted :: udom·lifted ⇒ (udom·lifted) ⇒ udom·lifted
  where bindU-lifted·(Lifted·x)·k = k·x

lemma bindU-lifted-strict [simp]: bindU·⊥·k = (⊥·udom·lifted)
  by fixrec-simp

definition returnU-lifted-def:
  returnU = Lifted

instance proof
  fix x :: udom
  fix f :: udom ⇒ udom
  fix h k :: udom·udom·lifted
  fix xs :: udom·lifted
  show fmapU·f·xs = bindU·xs·(Λ x. returnU·(f·x))
    by (induct xs rule: lifted.induct, simp-all add: returnU-lifted-def)
  show bindU·(returnU·x)·k = k·x
    by (simp add: returnU-lifted-def)
  show bindU·(bindU·xs·h)·k = bindU·xs·(Λ x. bindU·(h·x)·k)
    by (induct xs rule: lifted.induct) simp-all
qed
end

13.3 Transfer properties to polymorphic versions

lemma fmap-lifted-simps [simp]:
  fmap·f·(⊥·′a·lifted) = ⊥
  fmap·f·(Lifted·x) = Lifted·(f·x)
unfolding fmap-def by simp-all

lemma bind-lifted-simps [simp]:
  bind·(⊥·′a·lifted)·f = ⊥
  bind·(Lifted·x)·f = f·x
unfolding bind-def by simp-all

lemma return-lifted-def: return = Lifted
unfolding return-def returnU-lifted-def
by (simp add: coerce-cfun ccompI eta-cfun)

lemma join-lifted-simps [simp]:
  join·(⊥·′a·lifted·lifted) = ⊥
\texttt{join\cdot(Lifted\cdot xs) = xs}

unfolding \texttt{join-def by simp-all}

end

14 Resumption monad transformer

theory Resumption-Transformer
imports Monad-Plus
begin

14.1 Type definition

The standard Haskell libraries do not include a resumption monad transformer type; below is the Haskell definition for the one we will use here.

\texttt{data ResT m a = Done a | More (m (ResT m a))}

The above datatype definition can be translated directly into HOLCF using \texttt{tycondef}.

\texttt{tycondef 'a\cdot(f::functor) resT =}
\texttt{ Done (lazy 'a) | More (lazy ('a\cdot f resT)\cdot f)}

\texttt{lemma coerce-resT-abs [simp]: coerce\cdot(resT-abs\cdot x) = resT-abs\cdot(coerce\cdot x)}
\texttt{apply (simp add: resT-abs-def coerce-def)}
\texttt{done}

\texttt{lemma coerce-Done [simp]: coerce\cdot(Done\cdot x) = Done\cdot(coerce\cdot x)}
\texttt{unfolding Done-def by simp}

\texttt{lemma coerce-More [simp]: coerce\cdot(More\cdot m) = More\cdot(coerce\cdot m)}
\texttt{unfolding More-def by simp}

\texttt{lemma resT-induct [case-names adm bottom Done More]:}
\texttt{fixes P :: 'a\cdot f::functor resT ⇒ bool}
\texttt{assumes adm: adm P}
\texttt{assumes bottom: P ⊥}
\texttt{assumes Done: \(\Lambda x. P (Done\cdot x)\)}
\texttt{assumes More: \(\Lambda m f. (\Lambda (r::'a\cdot f resT). P (f\cdot r)) \implies P (More\cdot(fmap\cdot f\cdot m))\)}
\texttt{shows P r}
\texttt{proof (induct r rule: resT.take-induct [OF adm])}
\texttt{fix n show P (resT-take n\cdot r)}
\texttt{apply (induct n arbitrary: r)}
\texttt{apply (simp add: bottom)}
\texttt{apply (case-tac r rule: resT.exhaust)}
\texttt{apply (simp add: bottom)}
apply (simp add: Done)
done

14.2 Class instance proofs

lemma fmapU-resT-simps [simp]:
  fmapU.f(⊥::udom →'a resT) = ⊥
fmapU.f(Done·x) = Done·(f·x)
fmapU.f(More·m) = More·(fmap·(fmapU·f)·m)

unfolding fmapU-resT-def resT-map-def
apply (subst fix-eq, simp)
apply (subst fix-eq, simp add: Done-def)
apply (subst fix-eq, simp add: More-def)
done

instance resT :: (functor) functor
proof
  fix f g :: udom → udom and xs :: udom →'a resT
  show fmapU·f·(fmapU·g·xs) = fmapU·(f·g)·xs
    by (induct xs rule: resT-induct, simp-all add: fmap-fmap)
qed

instantiation resT :: (functor) monad
begin

fixrec bindU-resT :: udom →'a resT → (udom → udom →'a resT) → udom →'a resT
  where bindU-resT·(Done·x) = f·x
  | bindU-resT·(More·m) = More·(fmap·(Λ·r. bindU-resT·r·f)·m)

lemma bindU-resT-strict [simp]: bindU·⊥·k = (⊥::udom →'a resT)
by (fixrec-simp)

definition returnU = Done

instance proof
  fix f :: udom → udom and xs :: udom →'a resT
  show fmapU·f·xs = bindU·xs·(Λ·x. returnU·(f·x))
    by (induct xs rule: resT-induct)
    (simp-all add: fmap-fmap returnU-resT-def)

next
  fix f :: udom → udom →'a resT and x :: udom
  show bindU·(returnU·x)·f = f·x
    by (simp add: returnU-resT-def)

next
  fix xs :: udom →'a resT and h k :: udom → udom →'a resT
  show bindU·(bindU·xs·h)·k = bindU·xs·(Λ·x. bindU·(h·x)·k)
by (induct zs rule: resT-induct)
(simp-all add: fmap-fmap)
qed
end

14.3 Transfer properties to polymorphic versions

lemma fmap-resT-simps [simp]:
\[ \text{fmap} \cdot \text{f} \cdot (\bot :: \text{functor resT}) = \bot \]
\[ \text{fmap} \cdot \text{f} \cdot (\text{Done} \cdot x :: \text{functor resT}) = \text{Done} \cdot (f \cdot x) \]
\[ \text{fmap} \cdot \text{f} \cdot (\text{More} \cdot m :: \text{functor resT}) = \text{More} \cdot (\text{fmap} \cdot (fmap \cdot f) \cdot m) \]
unfolding fmap-def [where f=’f resT]
by (simp-all add: coerce-simp)

lemma return-resT-def: return = Done
unfolding return-def returnU-resT-def
by (simp add: coerce-simp eta-cfun)

lemma bind-resT-simps [simp]:
\[ \text{bind} \cdot (\bot :: \text{functor resT}) \cdot f = \bot \]
\[ \text{bind} \cdot (\text{Done} \cdot x :: \text{functor resT}) \cdot f = f \cdot x \]
\[ \text{bind} \cdot (\text{More} \cdot m :: \text{functor resT}) \cdot f = \text{More} \cdot (\text{fmap} \cdot (\Lambda \cdot r. \text{bind} \cdot r \cdot f) \cdot m) \]
unfolding bind-def
by (simp-all add: coerce-simp)

lemma join-resT-simps [simp]:
\[ \text{join} \cdot \bot = (\bot :: \text{functor resT}) \]
\[ \text{join} \cdot (\text{Done} \cdot x) = x \]
\[ \text{join} \cdot (\text{More} \cdot m) = \text{More} \cdot (\text{fmap} \cdot \text{join} \cdot m) \]
unfolding join-def by simp-all

14.4 Nondeterministic interleaving

In this section we present a more general formalization of the nondeterministic interleaving operation presented in Chapter 7 of the author’s PhD thesis [2]. If both arguments are Done, then zipRT combines the results with the function f and terminates. While either argument is More, zipRT nondeterministically chooses one such argument, runs it for one step, and then calls itself recursively.

fixrec zipRT ::
\[ (a \rightarrow b \rightarrow c) \rightarrow (a :: \text{functor-plus}) \text{resT} \rightarrow b :: \text{resT} \rightarrow c :: \text{resT} \]
where zipRT-Done-Done:
\[ \text{zipRT} \cdot f \cdot (\text{Done} \cdot x) \cdot (\text{Done} \cdot y) = \text{Done} \cdot (f \cdot x \cdot y) \]
| zipRT-Done-More:
\[ \text{zipRT} \cdot f \cdot (\text{Done} \cdot x) \cdot (\text{More} \cdot b) = \text{More} \cdot (\text{fmap} \cdot (\Lambda \cdot r. \text{zipRT} \cdot f \cdot (\text{Done} \cdot x) \cdot r) \cdot b) \]
| zipRT-More-Done:
zipRT\cdot f\cdot (More\cdot a)\cdot (Done\cdot y) = \\
           More\cdot (fmap\cdot (\lambda r.\ zipRT\cdot f\cdot r\cdot (Done\cdot y))\cdot a)
| zipRT\cdot More\cdot More:
  zipRT\cdot f\cdot (More\cdot a)\cdot (More\cdot b) = \\
           More\cdot (fplus\cdot (fmap\cdot (\lambda r.\ zipRT\cdot f\cdot r\cdot (More\cdot a))\cdot b) \\
            \cdot (fmap\cdot (\lambda r.\ zipRT\cdot f\cdot r\cdot (More\cdot b))\cdot a))

**Lemma** \(\text{zipRT-strict1} [\text{simp}]: \text{zipRT}\cdot f:\bot\cdot r = \bot\)
by \text{fixrec-simp}

**Lemma** \(\text{zipRT-strict2} [\text{simp}]: \text{zipRT}\cdot f\cdot r\cdot \bot = \bot\)
by \((\text{fixrec-simp, cases } r, \text{simp-all})\)

**Abbreviation** \(\text{apR} (\text{infixl} \odot 70)\)
where \(a \odot b \equiv \text{zipRT}\cdot \text{ID} \cdot a \cdot b\)

Proofs that \text{zipRT} satisfies the applicative functor laws:

**Lemma** \(\text{zipRT-homomorphism}: \text{Done}\cdot f \odot \text{Done}\cdot x = \text{Done}\cdot (f\cdot x)\)
bysimp

**Lemma** \(\text{zipRT-identity}: \text{Done}\cdot \text{ID} \odot r = r\)
by \((\text{induct } r \text{ rule: resT-induct, simp-all add: fmap-fmap eta-cfun})\)

**Lemma** \(\text{zipRT-interchange}: r \odot \text{Done}\cdot x = \text{Done}\cdot (\lambda f. f\cdot x) \odot r\)
by \((\text{induct } r \text{ rule: resT-induct, simp-all add: fmap-fmap})\)

The associativity rule is the hard one!

**Lemma** \(\text{zipRT-associativity}: \text{Done}\cdot \text{cfcomp} \odot r1 \odot r2 \odot r3 = r1 \odot (r2 \odot r3)\)

**Proof** \((\text{induct } r1 \text{ arbitrary: } r2 \cdot r3 \text{ rule: resT-induct})\)
   \text{case} \((\text{Done } x1)\) \text{ thus } ?\text{case}
   \text{proof} \((\text{induct } r2 \text{ arbitrary: } r3 \text{ rule: resT-induct})\)
      \text{case} \((\text{Done } x2)\) \text{ thus } ?\text{case}
      \text{proof} \((\text{induct } r3 \text{ rule: resT-induct})\)
         \text{case} \((\text{More } p3 \cdot c3)\) \text{ thus } ?\text{case}
            \text{by} \((\text{simp add: fmap-fmap})\)
      \text{qed simp-all}
   \text{next}
   \text{case} \((\text{More } p2 \cdot c2)\) \text{ thus } ?\text{case}
   \text{proof} \((\text{induct } r3 \text{ rule: resT-induct})\)
      \text{case} \((\text{Done } x3)\) \text{ thus } ?\text{case}
          \text{by} \((\text{simp add: fmap-fmap})\)
   \text{next}
   \text{case} \((\text{More } p3 \cdot c3)\) \text{ thus } ?\text{case}
       \text{by} \((\text{simp add: fmap-fmap fmap-fplus})\)
  \text{qed simp-all}
\text{next}
\text{case} \((\text{More } p1 \cdot c1)\) \text{ thus } ?\text{case}
\text{proof} \((\text{induct } r2 \text{ arbitrary: } r3 \text{ rule: resT-induct})\)

40
case \( (\text{Done } y) \) thus \(?\text{case}\)
proof (induct \( r^3 \) rule: \text{resT-induct})
case \( (\text{Done } x^3) \) thus \(?\text{case}\)
  by (simp add: \text{fmap-fmap})
next
case \( (\text{More } p^3 c^3) \) thus \(?\text{case}\)
  by (simp add: \text{fmap-fmap})
qed simp-all
next
case \( (\text{More } p^2 c^2) \) thus \(?\text{case}\)
proof (induct \( r^3 \) rule: \text{resT-induct})
case \( (\text{Done } x^3) \) thus \(?\text{case}\)
  by (simp add: \text{fmap-fmap} \text{fmap-fplus})
next
case \( (\text{More } p^3 c^3) \) thus \(?\text{case}\)
  by (simp add: \text{fmap-fmap} \text{fmap-fplus} \text{fplus-assoc})
qed simp-all
qed simp-all
qed simp-all

end

15 State monad transformer

theory \text{State-Transformer}
imports \text{Monad-Zero-Plus}
begin

This version has non-lifted product, and a non-lifted function space.

\text{tycondef} \( 'a \cdot (f::\text{functor}, 's) \text{stateT} = \text{StateT} (\text{runStateT} :: 's \to ('a \times 's) \cdot f) \)

\text{lemma} \text{coerce-stateT-abs} [simp]: \text{coerce-(stateT-abs \cdot x) = stateT-abs-(coerce-x)}
\text{apply} (\text{simp add: stateT-abs-def coerce-def})
\text{apply} (\text{simp add: emb-prj-emb prj-emb-prj DEFL-eq-stateT})
done

\text{lemma} \text{coerce-StateT} [simp]: \text{coerce-(StateT \cdot k) = StateT-(coerce \cdot k)}
\text{unfolding} \text{StateT-def by simp}

\text{lemma} \text{stateT-cases} [case-names \text{StateT}]:
  \text{obtains } k \text{ where } y = \text{StateT} \cdot k
\text{proof}
  \text{show } y = \text{StateT} \cdot (\text{runStateT} \cdot y)
  \text{by (cases y, simp-all)}
\text{qed}

\text{lemma} \text{stateT-induct} [case-names \text{StateT}]:
  \text{fixes } P :: 'a\cdot (f::\text{functor},'s) \text{stateT} \Rightarrow \text{bool}
assumes $\forall k. P\ (\text{State}T\cdot k)$
shows $P\ y$
by (cases $y$ rule: stateT-cases, simp add: assms)

lemma stateT-eqI:
$(\forall s. \text{runState}T\cdot a\cdot s = \text{runState}T\cdot b\cdot s) \implies a = b$
apply (cases $a$ rule: stateT-cases)
apply (cases $b$ rule: stateT-cases)
apply (simp add: cfun-eq-iff)
done

lemma runStateT-coerce [simp]:
$\text{runState}T\cdot (\text{coerce}\cdot k)\cdot s = \text{coerce}\cdot (\text{runState}T\cdot k\cdot s)$
by (induct $k$ rule: stateT-induct, simp)

15.1 Functor class instance

lemma fmapU-StateT [simp]:
$fmapU\cdot f\cdot (\text{State}T\cdot k) = \text{State}T\cdot (\lambda s. (fmap\cdot (\lambda (x, s'). (f\cdot x, s')))\cdot (k\cdot s))$
unfolding fmapU-stateT-def stateT-map-def StateT-def
by (subst fix-eq, simp add: cfun-map-def csplit-def prod-map-def)

lemma runStateT-fmapU [simp]:
$\text{runState}T\cdot (fmapU\cdot f\cdot m)\cdot s = (fmap\cdot (\lambda (x, s'). (f\cdot x, s')))\cdot (\text{runState}T\cdot m\cdot s)$
by (cases $m$ rule: stateT-cases, simp)

instantiation stateT :: (functor, domain) functor
begin

instance
apply standard
apply (induct-tac xs rule: stateT-induct)
apply (simp-all add: fmap-fmap ID-def csplit-def)
done

end

15.2 Monad class instance

instantiation stateT :: (monad, domain) monad
begin

definition returnU-stateT-def:
$\text{return}U = (\lambda x. \text{State}T\cdot (\lambda s. \text{return}\cdot (x, s)))$

definition bindU-stateT-def:
$\text{bind}U = (\lambda m\ k. \text{State}T\cdot (\lambda s. \text{runState}T\cdot m\cdot s) \gg= (\lambda (x, s'). \text{runState}T\cdot (k\cdot x)\cdot s'))$
lemma bindU-stateT-StateT [simp]:
bindU·(StateT·f)·k =
StateT·(Λ s. f·s) ≌ (Λ (x, s'). runStateT·(k·x)·s')
unfolding bindU-stateT-def by simp

lemma runStateT-bindU [simp]:
runStateT·(bindU·m·k)·s = runStateT·m·s ≌ (Λ (x, s'). runStateT·(k·x)·s')
unfolding bindU-stateT-def by simp

instance proof
  fix f :: udom → udom and r :: udom·('a,'b) stateT
  show fmapU·f·r = bindU·r·(Λ x. returnU·(f·x))
    by (rule stateT-eqI)
      (simp add: returnU-stateT-def monad-fmap prod-map-def csplit-def)
next
  fix f :: udom → udom·('a,'b) stateT and x :: udom
  show bindU·(returnU·x)·f = f·x
    by (rule stateT-eqI)
      (simp add: returnU-stateT-def eta-cfun)
next
  fix r :: udom·('a,'b) stateT and f g :: udom→ udom·('a,'b) stateT
  show bindU·(bindU·f·g)·r = bindU·r·(Λ x. bindU·(f·x)·g)
    by (rule stateT-eqI)
      (simp add: bind-bind csplit-def)
qed

end

15.3 Monad zero instance

instantiation stateT :: (monad-zero, domain) monad-zero begin

definition zeroU-stateT-def:
zeroU = StateT·(Λ s. mzero)

lemma runStateT-zeroU [simp]:
runStateT·zeroU·s = mzero
unfolding zeroU-stateT-def by simp

instance proof
  fix k :: udom → udom·('a,'b) stateT
  show bindU·zeroU·k = zeroU
    by (rule stateT-eqI, simp add: bind-mzero)
qed

end
15.4 Monad plus instance

instantiation stateT :: (monad-plus, domain) monad-plus
begin

definition plusU-stateT-def:
    plusU = (\Lambda a b. StateT. (\Lambda s. mplus. (runStateT a s) (runStateT b s)))

lemma runStateT-plusU [simp]:
    runStateT.(plusU a b) s = mplus.(runStateT a s) (runStateT b s)
unfolding plusU-stateT-def by simp

instance proof
fix a b :: udom.(\'a, \'b) stateT and k :: udom \rightarrow udom.(\'a, \'b) stateT
show bindU.(plusU a b) k = plusU.(bindU a k) (bindU b k)
    by (rule stateT-eqI, simp add: bind-mplus)
next
fix a b c :: udom.(\'a, \'b) stateT
show plusU.(plusU a b) c = plusU a.(plusU b c)
    by (rule stateT-eqI, simp add: mplus-assoc)
qed

end

15.5 Monad zero plus instance

instance stateT :: (monad-zero-plus, domain) monad-zero-plus
proof
fix m :: udom.(\'a, \'b) stateT
show plusU.zeroU m = m
    by (rule stateT-eqI, simp add: mplus-mzero-left)
next
fix m :: udom.(\'a, \'b) stateT
show plusU.m.zeroU = m
    by (rule stateT-eqI, simp add: mplus-mzero-right)
qed

15.6 Transfer properties to polymorphic versions

lemma coerce-csplit [coerce-simp]:
    shows coerce.(csplit.f p) = csplit.(\Lambda x y. coerce.(f x y)) p
unfolding csplit-def by simp

lemma csplit-coerce [coerce-simp]:
fixes p :: \'a \times \'b
shows csplit.f.(COERCE(\'a \times \'b, \'c \times \'d) p) = csplit.(\Lambda x y. f.(COERCE(\'a, \'c) x) (COERCE(\'b, \'d) y)) p
unfolding coerce-prod csplit-def prod-map-def by simp
lemma fmap-stateT-simps [simp]:
\[ \text{fmap}\cdot f \cdot (\text{StateT}\cdot m \colon \text{a} \cdot (f \colon \text{functor}, \text{g} \colon \text{stateT}) = \text{StateT}\cdot (\Lambda \ s. \ \text{fmap}\cdot (\Lambda \ (x, \ s'). \ (f \cdot x, \ s'))\cdot (m\cdot s)) \]

unfolding fmap-def [where \( f = (f, \ \text{g}) \cdot \text{stateT} \)]
by (simp add: coerce-simp eta-cfun)

lemma runStateT-fmap [simp]:
\[ \text{runStateT}\cdot (\text{fmap}\cdot f \cdot m)\cdot s = \text{fmap}\cdot (\Lambda \ (x, \ s'). \ (f \cdot x, \ s'))\cdot (\text{runStateT}\cdot m\cdot s) \]
by (induct m rule: stateT-induct, simp)

lemma return-stateT-def:
\[ (\text{return} :: \text{a} \cdot (m \colon \text{monad}, \text{g} \colon \text{stateT}) = \text{StateT}\cdot (\Lambda \ x. \ \text{return}\cdot (x, \ s))) \]
unfolding return-def [where \( m = (m, \ \text{g}) \cdot \text{stateT} \)]
by (simp add: coerce-simp)

lemma bind-stateT-def:
\[ \text{bind} = (\Lambda \ m \ k. \ \text{StateT}\cdot (\Lambda \ s. \ \text{runStateT}\cdot m\cdot s \gg (\Lambda \ (x, \ s'). \ \text{runStateT}\cdot (k\cdot x)\cdot s'))) \]
apply (subst bind-def, subst bindU-stateT-def)
apply (simp add: coerce-simp)
apply (simp add: coerce-idem domain-defl-simps monofun-cfun)
done

TODO: add coerce-idem to coerce-simps, along with monotonicity rules for DEFL.

lemma bind-stateT-simps [simp]:
\[ \text{bind}\cdot (\text{StateT}\cdot m \colon \text{a} \cdot (m \colon \text{monad}, \text{g} \colon \text{stateT})\cdot k = \text{StateT}\cdot (\Lambda \ s. \ m\cdot s \gg (\Lambda \ (x, \ s'). \ \text{runStateT}\cdot (k\cdot x)\cdot s')) \]

unfolding bind-stateT-def by simp

lemma runStateT-bind [simp]:
\[ \text{runStateT}\cdot (m \gg k)\cdot s = \text{runStateT}\cdot m\cdot s \gg (\Lambda \ (x, \ s'). \ \text{runStateT}\cdot (k\cdot x)\cdot s') \]

unfolding bind-stateT-def by simp

end

16  Error monad transformer

theory Error-Transformer
imports Error-Monad
begin

16.1  Type definition

The error monad transformer is defined in Haskell by composing the given monad with a standard error monad:
data Error e a = Err e | Ok a
newtype ErrorT e m a = ErrorT { runErrorT :: m (Error e a) }

We can formalize this definition directly using `tycondef`.

```
tycondef 'a ('f::functor,'e::domain) errorT = ErrorT (runErrorT :: ('a::e error)'f)
```

```
lemma coerce-errorT-abs [simp]: coerce-(errorT-abs-x) = errorT-abs-(coerce-x)
apply (simp add: errorT-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-errorT)
done
```

```
lemma coerce-ErrorT [simp]: coerce-(ErrorT-k) = ErrorT-(coerce-k)
unfolding ErrorT-def by simp
```

```
lemma errorT-cases [case-names ErrorT]:
  obtains k where y = ErrorT·k
proof
  show y = ErrorT·(runErrorT·y)
  by (cases y, simp-all)
qed
```

```
lemma ErrorT-runErrorT [simp]: ErrorT·(runErrorT·m) = m
by (cases m rule: errorT-cases, simp)
```

```
lemma errorT-induct [case-names ErrorT]:
  fixes P :: 'a ('f::functor,'e) errorT ⇒ bool
  assumes P k (ErrorT·k)
  shows P y
by (cases y rule: errorT-cases, simp add: assms)
```

```
lemma errorT-eq-iff:
  a = b ⇔ runErrorT·a = runErrorT·b
apply (cases a rule: errorT-cases)
apply (cases b rule: errorT-cases)
apply simp
done
```

```
lemma errorT-eqI:
  runErrorT·a = runErrorT·b ⇒ a = b
by (simp add: errorT-eq-iff)
```

```
lemma runErrorT-coerce [simp]:
  runErrorT·(coerce-k) = coerce·(runErrorT·k)
by (induct k rule: errorT-induct, simp)
```

### 16.2 Functor class instance

```
lemma fmap-error-def: fmap = error-map-ID
```
apply (rule cfun-eqI, rename-tac f)
apply (rule cfun-eqI, rename-tac x)
apply (case-tac x rule: error.exhaust, simp-all)
apply (simp add: error-map-def fix-const)
apply (simp add: error-map-def fix-const Err-def)
apply (simp add: error-map-def fix-const Ok-def)
done

lemma fmapU-ErrorT [simp]:
  fmapU·f·(ErrorT·m) = ErrorT·(fmap·(fmap·f)·m)
unfolding fmapU-errorT-def errorT-map-def fmap-error-def fix-const ErrorT-def
by simp

lemma runErrorT-fmapU [simp]:
  runErrorT·(fmapU·f·m) = fmap·(fmap·f)·(runErrorT·m)
by (induct m rule: errorT-induct) simp

instance errorT :: (functor, domain) functor
proof
  fix f g and xs :: udom·('a, 'b) errorT
  show fmapU·f·(fmapU·g·xs) = fmapU·(Λ x. f·(g·x))·xs
    apply (induct xs rule: errorT-induct)
    apply (simp add: fmap-fmap eta-cfun)
done

qed

16.3 Transfer properties to polymorphic versions

lemma fmap-ErrorT [simp]:
  fixes f :: 'a ⇒ 'b and m :: 'a·('m::functor, 'e errorT)
  shows fmap·f·(ErrorT·m) = ErrorT·(fmap·(fmap·f)·m)
unfolding fmap-def [where 'f=("m","e") errorT]
by (simp-all add: coerce-simp eta-cfun)

lemma runErrorT-fmap [simp]:
  fixes f :: 'a ⇒ 'b and m :: 'a·('m::functor,'e) errorT
  shows runErrorT·(fmap·f·m) = fmap·(fmap·f)·(runErrorT·m)
using fmap-ErrorT [of f runErrorT·m]
by simp

lemma errorT-fmap-strict [simp]:
  shows fmap·f·⊥·('m::monad,'e errorT) = ⊥
by (simp add: errorT-eq-iff fmap-strict)

16.4 Monad operations

The error monad transformer does not yield a monad in the usual sense: We cannot prove a monad class instance, because type 'a·('m,'e) errorT contains values that break the monad laws. However, it turns out that
such values are inaccessible: The monad laws are satisfied by all values constructible from the abstract operations.

To explore the properties of the error monad transformer operations, we define them all as non-overloaded functions.

definition unitET :: 'a ⇒ 'a · ('m::monad,'e) errorT
  where unitET = (Λ x. ErrorT·(return·(Ok·x)))

definition bindET :: 'a·('m::monad,'e) errorT ⇒ ('a ⇒ 'b·('m,'e) errorT) ⇒ 'b·('m,'e) errorT
  where bindET = (Λ m k. ErrorT·(bind·(runErrorT·m)·(Λ n. case n of Err·e ⇒ return·(Err·e) | Ok·x ⇒ runErrorT·(k·x))))

definition liftET :: 'a·'m::monad ⇒ 'a·('m,'e) errorT
  where liftET = (Λ m. ErrorT·(fmap·Ok·m))

definition throwET :: 'e ⇒ 'a·('m::monad,'e) errorT
  where throwET = (Λ e. ErrorT·(return·(Err·e)))

definition catchET :: 'a·('m::monad,'e) errorT ⇒ ('a ⇒ 'b·('m,'e) errorT) ⇒ 'b·('m,'e) errorT
  where catchET = (Λ m h. ErrorT·(bind·(runErrorT·m)·(Λ n. case n of Err·e ⇒ runErrorT·(h·e) | Ok·x ⇒ return·(Ok·x))))

definition fmapET :: ('a ⇒ 'b) ⇒ 'a·('m::monad,'e) errorT ⇒ 'b·('m,'e) errorT
  where fmapET = (Λ f m. bindET·m·(Λ x. unitET·(f·x)))

lemma runErrorT-unitET [simp]:
  runErrorT·(unitET·x) = return·(Ok·x)

unfolding unitET-def by simp

lemma runErrorT-bindET [simp]:
  runErrorT·(bindET·m·k) = bind·(runErrorT·m)·
  (Λ n. case n of Err·e ⇒ return·(Err·e) | Ok·x ⇒ runErrorT·(k·x))

unfolding bindET-def by simp

lemma runErrorT-liftET [simp]:
  runErrorT·(liftET·m·k) = fmap·Ok·m

unfolding liftET-def by simp

lemma runErrorT-throwET [simp]:
  runErrorT·(throwET·e) = return·(Err·e)

unfolding throwET-def by simp

lemma runErrorT-catchET [simp]:
  runErrorT·(catchET·m·k) =
  bind·(runErrorT·m)·(Λ n. case n of
Err·e ⇒ runErrorT·(h·e) | Ok·x ⇒ return·(Ok·x))

unfolding catchET-def by simp

lemma runErrorT-fmapET [simp]:
  runErrorT·(fmapET·f·m) =
    bind·(runErrorT·m)·(Λ n. case n of
      Err·e ⇒ return·(Err·e) | Ok·x ⇒ return·(Ok·(f·x)))

unfolding fmapET-def by simp

16.5 Laws

lemma bindET-unitET [simp]:
  bindET·(unitET·x)·k = k·x
by (rule errorT-eqI, simp)

lemma catchET-unitET [simp]:
  catchET·(unitET·x)·h = unitET·x
by (rule errorT-eqI, simp)

lemma catchET-throwET [simp]:
  catchET·(throwET·e)·h = h·e
by (rule errorT-eqI, simp)

lemma liftET-return:
  liftET·(return·x) = unitET·x
by (rule errorT-eqI, simp add: fmap-return)

lemma liftET-bind:
  liftET·(bind·m·k) = bindET·(liftET·m)·(liftET oo k)
by (rule errorT-eqI, simp add: fmap-bind bind-fmap)

lemma bindET-throwET:
  bindET·(throwET·e)·k = throwET·e
by (rule errorT-eqI, simp)

lemma bindET-bindET:
  bindET·(bindET·m·h)·k = bindET·m·(Λ x. bindET·(h·x)·k)
apply (rule errorT-eqI)
apply simp
apply (simp add: bind-bind)
apply (rule cfun-ary-cong)
apply (rule cfun-eqI, simp)
apply (case_tac x)
apply (simp add: bind-strict)
apply simp
apply simp
done

lemma fmapET-fmapET:
\( \text{fmapET} \cdot f \cdot (\text{fmapET} \cdot g \cdot m) = \text{fmapET} \cdot (\lambda x. f \cdot (g \cdot x)) \cdot m \)

by (simp add: \text{fmapET-def} \ bindET\cdot\text{bindET})

Right unit monad law is not satisfied in general.

**Lemma** bindET\cdot\text{unitET-right-counterexample}:

- **fixes** \( m :: 'a\cdot('m::monad,'e) \text{errorT} \)
- **assumes** \( m = \text{ErrorT}\cdot(\text{return}\cdot\bot) \)
- **shows** \( \text{bindET}\cdot m\cdot \text{unitET} = m \)

by (simp add: \text{errorT-eq-iff} \ assms)

Right unit is satisfied for inner monads with strict return.

**Lemma** bindET\cdot\text{unitET-right-restricted}:

- **fixes** \( m :: 'a\cdot('m::monad,'e) \text{errorT} \)
- **assumes** \( \text{return}\cdot\bot = (\bot :: ('a::'e \text{error})\cdot\text{m}) \)
- **shows** \( \text{bindET}\cdot m\cdot \text{unitET} = m \)

unfolding \text{errorT-eq-iff}
apply simp
apply (rule trans \[OF - \text{monad-right-unit}])
apply (rule cfun-arg-cong)
apply (rule cfun-eqI)
apply (case-tac \( x \), simp-all add: \ assms)
done

**16.6 Error monad transformer invariant**

This inductively-defined invariant is supposed to represent the set of all values constructible using the standard \text{errorT} operations.

**Inductive** \text{invar} :: 'a\cdot('m::monad,'e) \text{errorT} \Rightarrow \text{bool}

where \text{invar-bottom}: \text{invar} \bot
| \text{invar-lub}: \lambda Y. \ [\text{chain} Y; \lambda i. \text{invar} (Y\ i)] \Rightarrow \text{invar} (\bigl\lceil i.\ Y\ i\bigr\rceil)
| \text{invar-unitET}: \lambda x. \text{invar} (\text{unitET}\cdot x)
| \text{invar-bindET}: \lambda m\ k. [\text{invar} m; \lambda x. \text{invar} (k\cdot x)] \Rightarrow \text{invar} (\text{bindET}\cdot m\cdot k)
| \text{invar-throwET}: \lambda e. \text{invar} (\text{throwET}\cdot e)
| \text{invar-catchET}: \lambda m\ h. [\text{invar} m; \lambda e. \text{invar} (h\cdot e)] \Rightarrow \text{invar} (\text{catchET}\cdot m\cdot h)
| \text{invar-liftET}: \lambda m. \text{invar} (\text{liftET}\cdot m)

Right unit is satisfied for arguments built from standard functions.

**Lemma** bindET\cdot\text{unitET-right-invar}:

- **assumes** \text{invar} \( m \)
- **shows** \( \text{bindET}\cdot m\cdot \text{unitET} = m \)

using assms
apply (induct set: \text{invar})
apply (rule \text{errorT-eqI}, simp add: \text{bind-strict})
apply (rule admD, simp, assumption, assumption)
apply (rule \text{errorT-eqI}, simp)
apply (simp add: \text{errorT-eq-iff} \ text{bind-bind})

50
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x, simp add: bind-strict, simp, simp)
apply (rule errorT-eqI, simp)
apply (simp add: errorT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x, simp add: bind-strict, simp, simp)
apply (rule errorT-eqI, simp add: monad-fmap bind-bind)
done

Monad-fmap is satisfied for arguments built from standard functions.

lemma errorT-monad-fmap-invar:
  fixes f :: 'a ⇒ 'b and m :: 'a → ('m :: monad, errorT)
  assumes invar m
  shows fmap · f · m = bindET · m · (Λ x. unitET · (f · x))
using assms
apply (induct set: invar)
apply (rule errorT-eqI, simp add: bind-strict fmap-strict)
apply (rule admD, simp, assumption, assumption)
apply (rule errorT-eqI, simp add: fmap-return)
apply (simp add: errorT-eq-iff bind-bind fmap-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x)
apply (simp add: bind-strict fmap-strict)
apply (simp add: fmap-return)
apply simp
apply (rule errorT-eqI, simp add: fmap-return)
apply (simp add: errorT-eq-iff bind-bind fmap-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x)
apply (simp add: bind-strict fmap-strict)
apply simp
apply (simp add: fmap-return)
apply (rule errorT-eqI, simp add: monad-fmap bind-bind return-error-def)
done

16.7 Invariant expressed as a deflation

We can also define an invariant in a more semantic way, as the set of fixed-points of a deflation.

definition invar' :: 'a · ('m :: monad, errorT) ⇒ bool
  where invar' m ←→ fmapET · ID · m = m

All standard operations preserve the invariant.

lemma invar'-unitET; invar' (unitET · x)
  unfolding invar'-def by (simp add: fmapET-def)

lemma invar'-fmapET; invar' m ⇒ invar' (fmapET · f · m)
  unfolding invar'-def
by (erule subst, simp add: fmapET-def bindET-bindET eta-cfun)

**lemma invar′-bindET**: 

\[ \text{invar} \prime \ m; \ \forall x. \ \text{invar} \prime \ (k \cdot x) \implies \text{invar} \prime \ (\text{bindET} \cdot m \cdot k) \]

**unfolding invar′-def**

by (simp add: fmapET-def bindET-bindET eta-cfun)

**lemma invar′-throwET**: 

\[ \text{invar} \prime \ (\text{throwET} \cdot e) \]

**unfolding invar′-def** by (simp add: fmapET-def bindET-throwET eta-cfun)

**lemma invar′-catchET**: 

\[ \text{invar} \prime \ m; \ \forall e. \ \text{invar} \prime \ (h \cdot e) \]

**unfolding invar′-def**

apply (simp add: fmapET-def eta-cfun)

apply (rule errorT-eqI)

apply (rule cfun-eqI)

apply (rule cfun-arg-cong)

apply (rule bindET-fmapET-unitET)

apply simp

apply (erule-tac t = h \cdot e in subst)

apply simp

apply simp

done

**lemma invar′-liftET**: 

\[ \text{invar} \prime \ (\text{liftET} \cdot m) \]

**unfolding invar′-def**

apply (simp add: fmapET-def errorT-eq-iff)

apply simp

apply simp

done

**lemma invar′-bottom**: 

\[ \text{invar} \prime \ \bot \]

**unfolding invar′-def**

by (simp add: errorT-eq-iff bind-strict)

**lemma adm-invar′**: 

\[ \text{adm invar} \prime \]

**unfolding invar′-def** [abs-def] by simp

All monad laws are preserved by values satisfying the invariant.

**lemma bindET-fmapET-unitET**: 

**shows** 

\[ \text{bindET} \cdot (\text{fmapET} \cdot f \cdot m) \cdot \text{unitET} = \text{fmapET} \cdot f \cdot m \]

by (simp add: fmapET-def bindET-bindET)

**lemma invar′-right-unit**: 

\[ \text{invar} \prime \ m \implies \text{bindET} \cdot m \cdot \text{unitET} = m \]

**unfolding invar′-def** by (erule subst, rule bindET-fmapET-unitET)

**lemma invar′-monad-fmap**: 

\[ \text{invar} \prime \ m \implies \text{fmapET} \cdot f \cdot m = \text{bindET} \cdot m \cdot (\Lambda x. \ \text{unitET} \cdot (f \cdot x)) \]

**unfolding invar′-def** by (erule subst, simp add: errorT-eq-iff)
lemma invar′-bind-assoc:
\[
\begin{align*}
\text{\text{invar}'} \ m; \ \& x. \ \text{invar}'(f \cdot x); \ \& y. \ \text{invar}'(g \cdot y) \\
\implies \text{bindET'(bindET-m \cdot f)} \cdot g = \text{bindET-m(\Lambda \ x. \ \text{bindET-}(f \cdot x) \cdot g)}
\end{align*}
\]
by (rule bindET-bindET)
end

17 Writer monad transformer

theory Writer-Transformer
imports Writer-Monad
begin

17.1 Type definition

Below is the standard Haskell definition of a writer monad transformer:

newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }

In this development, since a lazy pair type is not pre-defined in HOLCF, we will use an equivalent formulation in terms of our previous Writer type:

data Writer w a = Writer w a
newtype WriterT w m a = WriterT { runWriterT :: m (Writer w a) }

We can translate this definition directly into HOLCF using tycondef.

tycondef 'a·('m::functor,'w) writerT =
    WriterT (runWriterT :: ('a·'w writer)·'m)

lemma coerce-writerT-abs [simp]:
    coerce-(writerT-abs·x) = writerT-abs·(coerce·x)
apply (simp add: writerT-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-writerT)
done

lemma coerce-WriterT [simp]:
    coerce·(WriterT·k) = WriterT·(coerce·k)
unfolding WriterT-def by simp

lemma writerT-cases [case-names WriterT]:
    obtains k where y = WriterT·k
proof
    show y = WriterT·(runWriterT·y)
    by (cases y, simp-all)
qed

53
lemma \( \text{WriterT-runWriterT} \) [simp]: \( \text{WriterT}(\text{runWriterT}\cdot m) = m \)
by (cases \( m \) rule: \text{writerT-cases}, simp)

lemma \( \text{writerT-induct} \) [case-names WriterT];
fixes \( P :: 'a::\text{functor} \cdot \text{e} \) \( \text{writerT} \Rightarrow \text{bool} \)
assumes \( \forall k. P (\text{WriterT}\cdot k) \)
shows \( P' \)
by (cases \( y \) rule: \text{writerT-cases}, simp add: assms)

lemma \( \text{writerT-eq-iff} \):
\( a = b \Longleftrightarrow \text{runWriterT}\cdot a = \text{runWriterT}\cdot b \)
apply (cases \( a \) rule: \text{writerT-cases})
apply (cases \( b \) rule: \text{writerT-cases})
apply simp
done

lemma \( \text{writerT-below-iff} \):
\( a \sqsubseteq b \Longleftrightarrow \text{runWriterT}\cdot a \sqsubseteq \text{runWriterT}\cdot b \)
apply (cases \( a \) rule: \text{writerT-cases})
apply (cases \( b \) rule: \text{writerT-cases})
apply simp
done

lemma \( \text{writerT-eqI} \):
\( \text{runWriterT}\cdot a = \text{runWriterT}\cdot b \Longrightarrow a = b \)
by (simp add: \text{writerT-eq-iff})

lemma \( \text{writerT-belowI} \):
\( \text{runWriterT}\cdot a \sqsubseteq \text{runWriterT}\cdot b \Longrightarrow a \sqsubseteq b \)
by (simp add: \text{writerT-below-iff})

lemma \( \text{runWriterT-coerce} \) [simp]:
\( \text{runWriterT}\cdot(\text{coerce}\cdot k) = \text{coerce}\cdot(\text{runWriterT}\cdot k) \)
by (induct \( k \) rule: \text{writerT-induct}, simp)

\section{17.2 Functor class instance}

lemma \( \text{fmap-writer-def} \): \( \text{fmap} = \text{writer-map}\cdot \text{ID} \)
apply (rule \text{cfun-eqI}, rename-tac \( f \))
apply (rule \text{cfun-eqI}, rename-tac \( x \))
apply (case-tac \( x \) rule: \text{writer-exhaust}, simp-all)
apply (simp add: \text{writer-map-def fix-const})
apply (simp add: \text{writer-map-def fix-const Writer-def})
done

lemma \( \text{fmapU-WriterT} \) [simp]:
\( \text{fmapU}\cdot f\cdot (\text{WriterT}\cdot m) = \text{WriterT}\cdot (\text{fmap}\cdot (\text{fmap}\cdot f)\cdot m) \)
unfolding \( \text{fmapU-writerT-def} \) \( \text{writerT-map-def} \) \( \text{fmap-writer-def fix-const WriterT-def} \) by simp

54
The writer monad transformer does not yield a monad in the usual sense: We cannot prove a monad class instance, because type 'a=('m,'w) writerT contains values that break the monad laws. However, it turns out that such values are inaccessible: The monad laws are satisfied by all values constructible from the abstract operations.

To explore the properties of the writer monad transformer operations, we define them all as non-overloaded functions.

**definition** unitWT :: 'a → 'a-(('m::monad,'w::monoid) writerT
where unitWT = (Λ x. WriterT.(return(Writer-mempty-x)))

**definition** bindWT :: 'a-(('m::monad,'w::monoid) writerT) → ('a → 'b-('m,'w) writerT) → 'b-('m,'w) writerT
where bindWT = (Λ m k. WriterT.(bind-(runWriterT·m).
(Λ(Writer·w·x). bind-(runWriterT·(k·x))·(Λ(Writer·w·y). return-(Writer·(mappend·w·w·f·y))))))

**definition** liftWT :: 'a→'a-(('m::monad,'w::monoid) writerT
where liftWT = (Λ m. WriterT·(fmap·(Writer-mempty·m)))

**definition** tellWT :: 'a → 'w → 'a-(('m::monad,'w::monoid) writerT
where tellWT = (Λ x w. WriterT·(return·(Writer·w·x)))

**definition** fmapWT :: ('a → 'b) → ('a-(('m::monad,'w::monoid) writerT) → 'b-('m,'w) writerT
where fmapWT = (Λ f m. bindWT·m·(Λ x. unitWT·(f·x)))

**lemma** runWriterT-fmap [simp]:
runWriterT·(fmap·f·m) = fmap·(fmap·f)·(runWriterT·m)
by (subst fmap-def, simp add: coerce-simp eta-cfun)

**lemma** runWriterT-unitWT [simp]:
runWriterT\cdot (unitWT\cdot x) = return\cdot (Writer\cdot mempty\cdot x)

unfolding unitWT-def by simp

lemma runWriterT-bindWT [simp]:
runWriterT\cdot (bindWT\cdot m\cdot k) = bind\cdot (runWriterT\cdot m).
(\Lambda (Writer\cdot w\cdot x). \ bind\cdot (runWriterT\cdot (k\cdot x)))\cdot (\Lambda (Writer\cdot w'\cdot y). \ return\cdot (Writer\cdot (mappend\cdot w\cdot w')\cdot y)))

unfolding bindWT-def by simp

lemma runWriterT-liftWT [simp]:
runWriterT\cdot (liftWT\cdot m) = fmap\cdot (Writer\cdot mempty)\cdot m

unfolding liftWT-def by simp

lemma runWriterT-tellWT [simp]:
runWriterT\cdot (tellWT\cdot x\cdot w) = return\cdot (Writer\cdot w\cdot x)

unfolding tellWT-def by simp

lemma runWriterT-fmapWT [simp]:
runWriterT\cdot (fmapWT\cdot f\cdot m) = runWriterT\cdot (\Lambda x. f\cdot (g\cdot x))\cdot m

by (simp add: fmapWT-def bindWT-def mempty-right)

17.4 Laws

The liftWT function maps return and bind on the inner monad to unitWT and bindWT, as expected.

lemma liftWT-return:
    liftWT\cdot (return\cdot x) = unitWT\cdot x

by (rule writerT-eqI, simp add: fmap-return)

lemma liftWT-bind:
    liftWT\cdot (bind\cdot m\cdot k) = bindWT\cdot (liftWT\cdot m)\cdot (liftWT oo k)

by (rule writerT-eqI)
  (simp add: monad-fmap bind-bind mempty-left)

The composition rule holds unconditionally for fmap. The fmap function also interacts as expected with unit and bind.

lemma fmapWT-fmapWT:
    fmapWT\cdot f\cdot (fmapWT\cdot g\cdot m) = fmapWT\cdot (\Lambda x. f\cdot (g\cdot x))\cdot m

apply (simp add: writerT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x, simp add: bind-strict, simp add: mempty-right)
done

lemma fmapWT-unitWT:
    fmapWT\cdot f\cdot (unitWT\cdot x) = unitWT\cdot (f\cdot x)

by (simp add: writerT-eq-iff mempty-right)

56
lemma \textit{fmapWT\textbullet\textit{bindWT}}:
\[
\text{fmapWT}\cdot f \cdot (\text{bindWT}\cdot m\cdot k) = \text{bindWT}\cdot m\cdot (\Lambda x. \text{fmapWT}\cdot f\cdot (k\cdot x))
\]
apply (simp add: writerT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, rename-tac x, simp)
apply (case-tac x, simp add: bind-strict, simp add: bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, rename-tac y, simp)
apply (case-tac y, simp add: bind-strict, simp add: mempty-right)
done

lemma \textit{bindWT\textbullet\textit{fmapWT}}:
\[
\text{bindWT}\cdot (\text{fmapWT}\cdot f\cdot m)\cdot k = \text{bindWT}\cdot m\cdot (\Lambda x. k\cdot (f\cdot x))
\]
apply (simp add: writerT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, rename-tac x, simp)
apply (case-tac x, simp add: bind-strict, simp add: mempty-right)
done

The left unit monad law is not satisfied in general.

lemma \textit{bindWT\textbullet\textit{unitWT-counterexample}}:
fixes \(k : 'a \rightarrow 'b \cdot ('m::monad, 'w::monoid)\) writerT
assumes \(1: k\cdot x = \text{WriterT}(\text{return}\cdot \bot)\)
assumes \(2: \text{return}\cdot \bot \neq (\bot :: ('b\cdot 'w\cdot w)\cdot 'm::monad)\)
shows \(\text{bindWT}\cdot (\text{unitWT}\cdot x)\cdot k \neq k\cdot x\)
by (simp add: writerT-eq-iff mempty-left assms)

However, left unit is satisfied for inner monads with a strict \textit{return} function.

lemma \textit{bindWT\textbullet\textit{unitWT-restricted}}:
fixes \(k : 'a \rightarrow 'b \cdot ('m::monad, 'w::monoid)\) writerT
assumes return\cdot \bot = (\bot :: ('b\cdot 'w\cdot writer)\cdot 'm)
shows \(\text{bindWT}\cdot (\text{unitWT}\cdot x)\cdot k = k\cdot x\)
unfolding writerT-eq-iff
apply (simp add: mempty-left)
apply (rule trans [OF - monad-right-unit])
apply (rule cfun-arg-cong)
apply (rule cfun-eqI)
apply (case-tac x, simp-all add: assms)
done

The associativity of \textit{bindWT} holds unconditionally.

lemma \textit{bindWT\textbullet\textit{bindWT}}:
\[
\text{bindWT}\cdot (\text{bindWT}\cdot m\cdot h)\cdot k = \text{bindWT}\cdot m\cdot (\Lambda x. \text{bindWT}\cdot (h\cdot x)\cdot k)
\]
apply (rule writerT-eqI)
apply simp
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, simp)
apply (case-tac x)
apply (simp add: bind-strict)
apply (simp add: bind-bind)
apply (rule cfun-ary-cong)
apply (rule cfun-eqI, simp, rename-tac y)
apply (case-tac y)
apply (simp add: bind-strict)
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, simp, rename-tac z)
apply (case-tac z)
apply (simp add: bind-strict)
apply (simp add: mappend-assoc)
done

The right unit monad law is not satisfied in general.

lemma bindWT-unitWT-right-counterexample:
  fixes m :: 'a::{m::monad,'w::monoid} writerT
  assumes m = WriterT·(return·⊥)
  assumes return·⊥̸ = (⊥:: ('a·'w writer)·m)
  shows bindWT·m·unitWT̸ = m
by (simp add: writerT-eq-iff assms)

Right unit is satisfied for inner monads with a strict return function.

lemma bindWT-unitWT-right-restricted:
  fixes m :: 'a::{m::monad,'w::monoid} writerT
  assumes return·⊥ = (⊥:: ('a·'w writer)·m)
  shows bindWT·m·unitWT = m
unfolding writerT-eq-iff
apply simp
apply (rule trans [OF - monad-right-unit])
apply (rule cfun-ary-cong)
apply (rule cfun-eqI)
apply (case-tac x, simp-all add: assms mempty-right)
done

17.5 Writer monad transformer invariant

We inductively define a predicate that includes all values that can be con-
structed from the standard writerT operations.

inductive invar :: 'a::{m::monad, 'w::monoid} writerT ⇒ bool
where invar-bottom: invar ⊥
invar-lub: ∀Y. [chain Y; ∀i. invar (Y i)] ⇒ invar (LUB i. Y i)
invar-unitWT: ∀x. invar (unitWT·x)
invar-bindWT: ∀m k. [invar m; ∀x. invar (k·x)] ⇒ invar (bindWT·m·k)
invar-tellWT: ∀x w. invar (tellWT·x·w)
invar-liftWT: ∀m. invar (liftWT·m)
Right unit is satisfied for arguments built from standard functions.

**Lemma** \[\text{bindWT-unitWT-right-invar}\]:

- **Fixes** \[m :: 'a \cdot ('m::monad,'w::monoid) \text{writerT}\]
- **Assumes** \[\text{invar} \ m\]
- **Shows** \[\text{bindWT} \cdot m \cdot \text{unitWT} = m\]
- **Using** \[\text{assms} \text{ proof (induct set: invar)}\]
  - **Case** \[\text{invar-bottom} \text{ thus } ?\text{case}\]
    - by \[\text{(rule writerT-eqI, simp add: bind-strict)}\]
  next
  - **Case** \[\text{invar-lub} \text{ thus } ?\text{case}\]
    - by \[\text{− (rule admD, simp, assumption, assumption)}\]
  next
  - **Case** \[\text{invar-unitWT} \text{ thus } ?\text{case}\]
    - by \[\text{(rule writerT-eqI, simp add: bind-bind mempty-left)}\]
  next
  - **Case** \[\text{invar-bindWT} \text{ thus } ?\text{case}\]
    - apply \[\text{(simp add: writerT-eq-iff bind-bind)}\]
    - apply \[\text{(rule cfun-arg-cong, rule cfun-eqI, simp)}\]
    - apply \[\text{(case-tac x, simp add: bind-strict, simp add: bind-bind)}\]
    - apply \[\text{(rule cfun-arg-cong, rule cfun-eqI, simp, rename-tac y)}\]
    - apply \[\text{(case-tac y, simp add: bind-strict, simp add: mempty-right)}\]
  done
  next
  - **Case** \[\text{invar-tellWT} \text{ thus } ?\text{case}\]
    - by \[\text{(simp add: writerT-eq-iff mempty-right)}\]
  next
  - **Case** \[\text{invar-liftWT} \text{ thus } ?\text{case}\]
    - by \[\text{(rule writerT-eqI, simp add: monad-fmap bind-bind mempty-right)}\]
  qed

Left unit is also satisfied for arguments built from standard functions.

**Lemma** \[\text{writerT-left-unit-invar-lemma}\]:

- **Assumes** \[\text{invar} \ m\]
- **Shows** \[\text{runWriterT} \cdot m \gg (\Lambda (\text{Writer} \cdot w \cdot x). \text{return} (\text{Writer} \cdot w \cdot x)) = \text{runWriterT} \cdot m\]
- **Using** \[\text{assms} \text{ proof (induct } m \text{ set: invar)}\]
  - **Case** \[\text{invar-bottom} \text{ thus } ?\text{case}\]
    - by \[\text{(simp add: bind-strict)}\]
  next
  - **Case** \[\text{invar-lub} \text{ thus } ?\text{case}\]
    - by \[\text{− (rule admD, simp, assumption, assumption)}\]
  next
  - **Case** \[\text{invar-unitWT} \text{ thus } ?\text{case}\]
    - by \[\text{simp}\]
  next
  - **Case** \[\text{invar-bindWT} \text{ thus } ?\text{case}\]
    - apply \[\text{(simp add: bind-bind)}\]
    - apply \[\text{(rule cfun-arg-cong)}\]
    - apply \[\text{(rule cfun-eqI, simp, rename-tac n)}\]
apply (case-tac n, simp add: bind-strict)
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, simp, rename-tac p)
apply (case-tac p, simp add: bind-strict)
apply simp
done

next

  case invar-tellWT thus ?case
  by simp

next

  case invar-liftWT thus ?case
  by (simp add: monad-fmap bind-bind)

qed

lemma bindWT-unitWT-invar:
  assumes invar (k·x)
  shows bindWT·(unitWT·x)·k = k·x
apply (simp add: writerT-eq-iff mempty-left)
apply (rule writerT-left-unit-invar-lemma [OF assms])
done

17.6 Invariant expressed as a deflation

definition invar' :: 'a·('m::monad, 'w::monoid) writerT ⇒ bool
where invar' m ⇔ fmapWT·ID·m = m

All standard operations preserve the invariant.

lemma invar'·bottom: invar' ⊥
unfolding invar'·def by (simp add: writerT-eq-iff bind-strict)

lemma adm-invar': adm invar'
unfolding invar'·def [abs-def] by simp

lemma invar'·unitWT: invar' (unitWT·x)
unfolding invar'·def by (simp add: writerT-eq-iff)

lemma invar'·bindWT: [invar' m; ∃x. invar' (k·x)] ⇒ invar' (bindWT·m·k)
unfolding invar'·def
apply (erule subst)
apply (simp add: writerT-eq-iff)
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, case-tac x)
apply (simp add: bind-strict)
apply simp
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, rename-tac x, case-tac x)
apply (simp add: bind-strict)
apply simp
done

lemma invar′-tellWT: invar′ (tellWT·x·w)
  unfolding invar′-def by (simp add: writerT-eq-iff)

lemma invar′-liftWT: invar′ (liftWT·m)
  unfolding invar′-def by (simp add: writerT-eq-iff monad-fmap bind-bind)

Left unit is satisfied for arguments built from fmap.

lemma bindWT-unitWT-fmapWT:
  bindWT·(unitWT·x)·(Λ x. fmapWT·f·(k·x))
  = fmapWT·f·(k·x)
apply (simp add: fmapWT-def writerT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x, simp-all add: bind-strict mempty-left)
done

Right unit is satisfied for arguments built from fmap.

lemma bindWT-fmapWT-unitWT:
  shows bindWT·(fmapWT·f·m)·unitWT = fmapWT·f·m
apply (simp add: bindWT-fmapWT)
apply (simp add: fmapWT-def)
done

All monad laws are preserved by values satisfying the invariant.

lemma invar′-right-unit: invar′ m ⇒ bindWT·m·unitWT = m
unfolding invar′-def by (erule subst, rule bindWT-fmapWT-unitWT)

lemma invar′-monad-fmap:
  invar′ m ⇒ fmapWT·f·m = bindWT·m·(Λ x. unitWT·(f·x))
unfolding invar′-def
by (erule subst, simp add: writerT-eq-iff mempty-right)

lemma invar′-bind-assoc:
  [invar′ m; ∀x. invar′ (f·x); ∀y. invar′ (g·y)]
  ⇒ bindWT·(bindWT·m·f)·g = bindWT·m·(Λ x. bindWT·(f·x)·g)
by (rule bindWT-bindWT)

end

References
