Applicative Lifting

Andreas Lochbihler        Joshua Schneider

October 10, 2017

Abstract

Applicative functors augment computations with effects by lifting function application to types which model the effects [5]. As the structure of the computation cannot depend on the effects, applicative expressions can be analysed statically. This allows us to lift universally quantified equations to the effectful types, as observed by Hinze [3]. Thus, equational reasoning over effectful computations can be reduced to pure types.

This entry provides a package for registering applicative functors and two proof methods for lifting of equations over applicative functors. The first method applicative-nf normalises applicative expressions according to the laws of applicative functors. This way, equations whose two sides contain the same list of variables can be lifted to every applicative functor.

To lift larger classes of equations, the second method applicative-lifting exploits a number of additional properties (e.g., commutativity of effects) provided the properties have been declared for the concrete applicative functor at hand upon registration.

We declare several types from the Isabelle library as applicative functors and illustrate the use of the methods with two examples: the lifting of the arithmetic type class hierarchy to streams and the verification of a relabelling function on binary trees. We also formalise and verify the normalisation algorithm used by the first proof method, as well as the general approach of the second method, which is based on bracket abstraction.

Contents

1 Lifting with applicative functors 3
   1.1 Equality restricted to a set 3
   1.2 Proof automation 3
   1.3 Overloaded applicative operators 5

2 Common applicative functors 5
   2.1 Environment functor 5
   2.2 Option 6
1 Lifting with applicative functors

theory Applicative
imports Main
keywords applicative :: thy-goal and print-applicative :: diag
begin

1.1 Equality restricted to a set

definition eq-on :: 'a set ⇒ 'a ⇒ 'a ⇒ bool
where [simp]: eq-on A = (λx y. x ∈ A ∧ x = y)

lemma rel-fun-eq-onI: (∀x. x ∈ A ⇒ R (f x) (g x)) ⇒ rel-fun (eq-on A) R f g
by auto

1.2 Proof automation

lemma arg1-cong: x = y ⇒ f x z = f y z
by (rule arg-cong)

lemma UNIV-E: x ∈ UNIV ⇒ P ⇒ P .

context begin

private named-theorems combinator-unfold
private named-theorems combinator-repr

private definition B g f x ≡ g (f x)
private definition C f x y ≡ f y x
private definition I x ≡ x
private definition K x y ≡ x
private definition S f g x ≡ (f x) (g x)
private definition T x f ≡ f x
private definition W f x ≡ f x x

lemmas [combinator-repr] = combinator-unfold

private definition cpair ≡ Pair
private definition cuncurry ≡ case-prod

private lemma uncurry-pair: cuncurry f (cpair x y) = f x y
unfolding cpair-def cuncurry-def by simp

ML-file applicative.ML

local-setup ⟨Applicative.setup-combinators
[(B, @{thm B-def}),
(C, @{thm C-def}),
(I, @{thm I-def})],
(K, \{thm K-def\}),
(S, \{thm S-def\}),
(T, \{thm T-def\}),
(W, \{thm W-def\});

private attribute-setup combinator-eq =
⟨Scan.lift (Scan.option (Args.$$ weak |--
  Scan.optional (Args.colon |-- Scan.repeat1 Args.name) [])) >>
  Applicative.combinator-rule-attrib⟩

lemma [combinator-eq]: B ≡ S (K S) K unfolding combinator-unfold .
lemma [combinator-eq]: C ≡ S (S (K S) K) S) (K K) unfolding combinator-unfold .
lemma [combinator-eq]: I ≡ W K unfolding combinator-unfold .
lemma [combinator-eq]: I ≡ C K () unfolding combinator-unfold .
lemma [combinator-eq]: S ≡ B (B W) (B B C) unfolding combinator-unfold .
lemma [combinator-eq]: T ≡ C I unfolding combinator-unfold .
lemma [combinator-eq]: W ≡ S S (S K) unfolding combinator-unfold .

lemma [combinator-eq weak]: C:
  C ≡ C (B B (B W) (C (B C (B B) (C B (cuncurry (K I))))) (cuncurry K))))) cpair
unfolding combinator-unfold uncurry-pair .

end

method-setup applicative-unfold =
⟨Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>
  SIMPLE-METHOD' (Applicative.unfold-wrapper-tac ctxt opt-af)); unfold into an applicative expression

method-setup applicative-fold =
⟨Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>
  SIMPLE-METHOD' (Applicative.fold-wrapper-tac ctxt opt-af)); fold an applicative expression

method-setup applicative-nf =
⟨Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>
  SIMPLE-METHOD' (Applicative.normalize-wrapper-tac ctxt opt-af));
  prove an equation that has been lifted to an applicative functor, using normal forms

method-setup applicative-lifting =
⟨Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>
  SIMPLE-METHOD' (Applicative.lifting-wrapper-tac ctxt opt-af));
  prove an equation that has been lifted to an applicative functor

ML ⟨Outer-Syntax.local-theory-to-proof @\{command-keyword applicative\}
register applicative functors

(Print.binding --
  Scan.optional ([@{keyword (} | -- Parse.list Parse.short-ident -- | @{keyword )}) [] --
  (@{keyword for} | -- Parse.reserved pure | -- @{keyword :} | -- Parse.term)
--
  (Parse.reserved ap | -- @{keyword :} | -- Parse.term) --
Scan.option (Parse.reserved rel | -- @{keyword :} | -- Parse.term) --
Scan.option (Parse.reserved set | -- @{keyword :} | -- Parse.term) >>
  Applicative.applicative-cmd)

ML <Outer-Syntax.command @{command-keyword print-applicative}
  print registered applicative functors
  (Scan.succeed (Toplevel.keep (Applicative.print-afuns o Toplevel.context-of))))

attribute-setup applicative-unfold =
  (Scan.lift (Scan.option Parse.name >> Applicative.add-unfold-attrib))
  register rules for unfolding into applicative expressions

attribute-setup applicative-lifted =
  (Scan.lift (Parse.name >> Applicative.forward-lift-attrib))
  lift an equation to an applicative functor

1.3 Overloaded applicative operators

consts
  pure :: 'a ⇒ 'b
  ap :: 'a ⇒ 'b ⇒ 'c

bundle applicative-syntax
begin
  notation ap (infixl ◦ 70)
end

hide-const (open) ap

end

2 Common applicative functors

2.1 Environment functor

theory Applicative-Environment imports
  Applicative
  HOL-Library.Adhoc-Overloading
begin

definition const x = (λ-. x)
definition apf x y = (λz. x z (y z))
**adhoc-overloading** Applicative.pure const

**adhoc-overloading** Applicative.ap apf

The declaration below demonstrates that applicative functors which lift the reductions for combinators K and W also lift C. However, the interchange law must be supplied in this case.

```plaintext
applicative env (K, W)
for
  pure: const
  ap: apf
  rel: rel-fun op =
  set: range
by(simp-all add: const-def apf-def rel-fun-def)
```

**lemma**
  includes applicative-syntax
  shows const (λf x y. f y x) o f o x o y = f o y o x
by applicative-lifting simp

### 2.2 Option

**theory** Applicative-Option **imports**
  Applicative
  HOL-Library, Adhoc-Overloading
begin

```plaintext
fun ap-option :: ('a ⇒ 'b) option ⇒ 'a option ⇒ 'b option
where
  ap-option (Some f) (Some x) = Some (f x)
  | ap-option - - = None
```

**abbreviation** (input) pure-option :: 'a ⇒ 'a option
where pure-option ≡ Some

**adhoc-overloading** Applicative.pure pure-option

**adhoc-overloading** Applicative.ap ap-option

**lemma** some-ap-option: ap-option (Some f) x = map-option f x
by (cases x) simp-all

**lemma** ap-some-option: ap-option f (Some x) = map-option (λg. g x) f
by (cases f) simp-all

**lemma** ap-option-transfer[transfer-rule]:
  rel-fun (rel-option (rel-fun A B)) (rel-fun (rel-option A) (rel-option B)) ap-option
ap-option
```
by (auto elim!: option.rel-cases simp add: rel-fun-def)

applicative option (C, W)

for
pure: Some
ap: ap-option
rel: rel-option
set: set-option

proof –
include applicative-syntax
{ fix x :: 'a option
  show pure (λx. x) ⋄ x = x by (cases x) simp-all
next
fix g :: ('b ⇒ 'c) option and f :: ('a ⇒ 'b) option and x
  show pure (λg f x. g (f x)) ⋄ g ⋄ f ⋄ x = g ⋄ (f ⋄ x)
    by (cases g f x rule: option.exhaust[case-product option.exhaust, case-product option.exhaust]) simp-all
next
fix f :: ('b ⇒ 'a ⇒ 'c) option and x y
  show pure (λf x y. f x y) ⋄ f ⋄ x ⋄ y = f ⋄ y ⋄ x
    by (cases f x y rule: option.exhaust[case-product option.exhaust, case-product option.exhaust]) simp-all
next
fix R :: 'a ⇒ 'b ⇒ bool
  show rel-fun R (rel-option R) pure pure by transfer-prover
next
fix R and f :: ('a ⇒ 'b) option and g :: ('a ⇒ 'c) option and x
  assume [transfer-rule]: rel-option (rel-fun (eq-on (set-option x)) R) f g
  have [transfer-rule]: rel-option (rel-fun (eq-on (set-option x)) x x by (auto intro: option.rel-refl-strong)
    show rel-option R (f ⋄ x) (g ⋄ x) by transfer-prover
} qed (simp add: some-ap-option ap-some-option)

lemma map-option-ap-conv[applicative-unfold]: map-option f x = ap-option (pure f) x
by (cases x rule: option.exhaust) simp-all

no-adhoc-overloading Applicative.pure pure-option — We do not want to print all occurrences of Some as pure

end
2.3 Sum types

theory Applicative-Sum imports
  Applicative
  HOL-Library.Adhoc-Overloading
begin

There are several ways to define an applicative functor based on sum types. First, we can choose whether the left or the right type is fixed. Both cases are isomorphic, of course. Next, what should happen if two values of the fixed type are combined? The corresponding operator must be associative, or the idiom laws don’t hold true.

We focus on the cases where the right type is fixed. We define two concrete functors: One based on Haskell’s Either datatype, which prefers the value of the left operand, and a generic one using the semigroup-add class. Only the former lifts the $W$ combinator, though.

fun ap-sum :: ('e ⇒ 'e ⇒ 'e) ⇒ ('a ⇒ 'b) + 'e ⇒ 'a + 'b + 'e
where
  ap-sum - (Inl f) (Inl x) = Inl (f x)
| ap-sum - (Inl -) (Inr e) = Inr e
| ap-sum - (Inr e) (Inl -) = Inr e
| ap-sum c (Inr e1) (Inr e2) = Inr (c e1 e2)

abbreviation ap-either ≡ ap-sum (λx -. x)
abbreviation ap-plus ≡ ap-sum (plus :: 'a :: semigroup-add ⇒ -)

abbreviation (input) pure-sum where pure-sum ≡ Inl
adhoc-overloading Applicative.pure pure-sum
adhoc-overloading Applicative.ap ap-either

lemma ap-sum-id: ap-sum c (Inl id) x = x
by (cases x) simp-all

lemma ap-sum-ichng: ap-sum c f (Inl x) = ap-sum c (Inl (λf. f x)) f
by (cases f) simp-all

lemma (in semigroup) ap-sum-comp:
  ap-sum f (ap-sum f (ap-sum f (Inl op a) h) g) x = ap-sum f h (ap-sum f g x)
by (cases h g x rule: sum.exhaust[case-product sum.exhaust, case-product sum.exhaust])
  (simp-all add: local.assoc)

lemma semigroup-const: semigroup (λx y. x)
by unfold-locales simp

locale either-af =
  fixes B :: 'b ⇒ 'b ⇒ bool
  assumes B-refl: reflp B
begin


8
applicative either (W)
for
pure: Inl
ap: ap-either
rel: \( \lambda A. \text{rel-sum} A B \)
proof –
include applicative-syntax
{ fix \( f :: ('c \Rightarrow 'c \Rightarrow 'd) + 'a \text{ and } x \) 
  show pure (\( \lambda f x. f x x \)) \circ f \circ x = f \circ x \circ x 
  by (cases f x rule: sum.exhaust[case-product sum.exhaust]) simp-all 
next 
interpret semigroup \( \lambda x y. x \) by (rule semigroup-const) 
fix \( g :: ('d \Rightarrow 'e) + 'a \text{ and } f :: ('c \Rightarrow 'd) + 'a \text{ and } x \) 
show pure (\( \lambda g f x. g (f x) \)) \circ g \circ f \circ x = g \circ (f \circ x) 
by (rule ap-sum-comp[simplified comp-def[abs-def]]) 
next 
fix \( R \text{ and } f :: ('c \Rightarrow 'd) + 'b \text{ and } g :: ('c \Rightarrow 'e) + 'b \text{ and } x \) 
assume rel-sum (rel-fun (eq-on UNIV) R) B f g 
then show rel-sum R B (f \circ x) (g \circ x) 
  by (cases f g x rule: sum.exhaust[case-product sum.exhaust, case-product sum.exhaust]) 
(auto intro: B-refl[THEN reflpD] elim: rel-funE) 
} 
qed (auto intro: ap-sum-id[simplified id-def] ap-sum-ichng) 
end
interpretation either-af op = by unfold-locales simp

applicative semigroup-sum
for
pure: Inl
ap: ap-plus
using
ap-sum-id[simplified id-def] 
ap-sum-ichng 
add.ap-sum-comp[simplified comp-def[abs-def]]
by auto

no-adhoc-overloading Applicative.pure pure-sum
end

2.4 Set with Cartesian product

theory Applicative-Set imports
Applicative
HOL-Library.Adhoc-Overloading
begin

definition ap-set :: ('a ⇒ 'b) set ⇒ 'a set ⇒ 'b set where ap-set F X = {f x | f x. f ∈ F ∧ x ∈ X}

adhoc-overloading Applicative.ap ap-set

lemma ap-set-transfer[transfer-rule]:
  rel-fun (rel-set (rel-fun A B)) (rel-fun (rel-set A) (rel-set B)) ap-set ap-set
unfolding ap-set-def[abs-def] rel-set-def
by (fastforce elim: rel-funE)

applicative set (C)
for
  pure: λx. {x}
ap: ap-set
rel: rel-set
set: λx. x

proof –
fix R :: 'a ⇒ 'b ⇒ bool
show rel-fun R (rel-set R) (λx. {x}) (λx. {x}) by (auto intro: rel-setI)

next
fix R and f :: ('a ⇒ 'b) set and g :: ('a ⇒ 'c) set and x
assume [transfer-rule]: rel-set (rel-fun (eq-on x) R) f g
have [transfer-rule]: rel-set (eq-on x) x x by (auto intro: rel-setI)
show rel-set R (ap-set f x) (ap-set g x) by transfer-prover
qed (unfold ap-set-def, fast+)

end

2.5 Lists

theory Applicative-List imports
  Applicative
  HOL-Library.Adhoc-Overloading
begin

definition ap-list fs xs = List.bind fs (λf. List.bind xs (λx. [f x]))

adhoc-overloading Applicative.ap ap-list

lemma Nil-ap[simp]: ap-list [] xs = []
unfolding ap-list-def by simp

lemma ap-Nil[simp]: ap-list fs [] = []
unfolding ap-list-def by (induction fs) simp-all

lemma ap-list-transfer[transfer-rule]:
  rel-fun (list-all2 (rel-fun A B)) (rel-fun (list-all2 A) (list-all2 B)) ap-list ap-list
unfolding ap-list-def[abs-def] List.bind-def
by transfer-prover

context includes applicative-syntax
begin

lemma cons-ap-list: (f # fs) ∘ xs = map f xs @ fs ∘ xs
unfolding ap-list-def by (induction xs) simp-all

lemma append-ap-distrib: (fs @ gs) ∘ xs = fs ∘ xs @ gs ∘ xs
unfolding ap-list-def by (induction fs) simp-all

applicative list
for
pure: λx. [x]
ap: ap-list
rel: list-all2
set: set

proof –
fix x :: 'a list
show [λx. x] ∘ x = x unfolding ap-list-def by (induction x) simp-all

next
fix g :: ('b ⇒ 'c) list and f :: ('a ⇒ 'b) list and x
let ?B = λg f. g (f x)
show [?B] ∘ g ∘ f ∘ x = g ∘ (f ∘ x)
proof (induction g)
  case Nil show ?case by simp
next
  case (Cons g gs)
  have g-comp: [?B g] ∘ f ∘ x = [g] ∘ (f ∘ x)
  proof (induction f)
    case Nil show ?case by simp
    next
      case (Cons f fs)
      have [?B g] ∘ (f # fs) ∘ x = [g] ∘ ([f] ∘ x) @ [?B g] ∘ fs ∘ x
        by (simp add: cons-ap-list)
      also have ... = [g] ∘ (f ∘ x) @ [g] ∘ (fs ∘ x) using Cons.IH ..
      also have ... = [g] ∘ (f [fs] ∘ x) by (simp add: cons-ap-list)
      finally show ?case .
    qed
  next
    have [?B] ∘ (g # gs) ∘ f ∘ x = [?B g] ∘ f ∘ x @ [?B g] ∘ gs ∘ f ∘ x
      by (simp add: cons-ap-list append-ap-distrib)
    also have ... = [g] ∘ (f ∘ x) @ gs ∘ (f ∘ x) using g-comp Cons.IH by simp
    also have ... = (g # gs) ∘ (f ∘ x) by (simp add: cons-ap-list)
    finally show ?case .
  qed
next
fix f :: ('a ⇒ 'b) list and x
show f ∘ [x] = [λf. f x] ∘ f unfolding ap-list-def by simp
next
  fix R :: 'a ⇒ 'b ⇒ bool
  show rel-fun R (list-all2 R) (λx. [x]) (λx. [x]) by transfer-prover
next
  fix R and f :: ('a ⇒ 'b) list and g :: ('a ⇒ 'c) list and x
  assume [transfer-rule]: list-all2 (rel-fun (eq-on (set x))) R f g
  have [transfer-rule]: list-all2 (eq-on (set x)) x x by (simp add: list-all2same)
  show list-all2 R (f ⋄ x) (g ⋄ x) by transfer-prover
qed (simp add: cons-ap-list)

lemma map-ap-conv[applicative-unfold]: map f x = [f] ⋄ x
unfolding ap-list-def List.bind-def
by simp

end

end

3 Distinct, non-empty list

theory Applicative-DNEList imports
  Applicative-List
  HOL-Library.List
begin

lemma bind-eq-Nil-iff [simp]: List.bind xs f = [] ←→ (∀x ∈ set xs. f x = [])
by (simp add: List.bind-def)

lemma zip-eq-Nil-iff [simp]: zip xs ys = [] ←→ xs = [] ∨ ys = []
by (cases xs ys rule: list.exhaust[case-product list.exhaust]) simp-all

lemma remdups-append1: remdups (remdups xs @ ys) = remdups (xs @ ys)
by (induction xs) simp-all

lemma remdups-append2: remdups (xs @ remdups ys) = remdups (xs @ ys)
by (induction xs) simp-all

lemma remdups-append1-drop: set xs ⊆ set ys =⇒ remdups (xs @ ys) = remdups ys
by (induction xs) auto

lemma remdups-concat-map: remdups (concat (map remdups xss)) = remdups (concat xss)
by (induction xss) (simp-all add: remdups-append1, metis remdups-append2)

lemma remdups-concat-remdups: remdups (concat (remdups xss)) = remdups (concat xss)
apply (induction xss)
apply (auto simp add: remdups-append1-drop)
apply (subst remdups-append1-drop; auto)
apply (metis remdups-append2)
done

lemma remdups-replicate: remdups (replicate n x) = (if n = 0 then [] else [x])
by (induction n) simp-all

typedef 'a dnelist = {xs::'a list. distinct xs ∧ xs ≠ []}
  morphisms list-of-dnelist Abs-dnelist
proof
  show [x] ∈ ?dnelist for x by simp
qed

setup-lifting type-definition-dnelist

lemma dnelist-subtype-dlist:
  type-definition (λx. Dlist (list-of-dnelist x)) (λx. Abs-dnelist (list-of-dlist x)) {xs. xs ≠ Dlist.empty}
  apply unfold-locales
subgoal by (transfer; auto simp add: dlist-eq-iff)
subgoal by (simp add: distinct-remdups-id dnelist.list-of-dnelist[simplified] list-of-dnelist-inverse)
subgoal by (simp add: dlist-eq-iff Abs-dnelist-inverse)
done

lift-bnf 'a dnelist via dnelsubtype-dlist for map: map by (simp-all add: dlist-eq-iff)
hide-const (open) map

context begin
qualified lemma map-def: Applicative-DNEList.map = map-fun id (map-fun list-of-dnelist Abs-dnelist) (λf xs. remdups (list.map f xs))
unfolding map-def by (simp add: fun-eq-iff distinct-remdups-id list-of-dnelist[simplified])

qualified lemma map-transfer [transfer-rule]:
  rel-fun op = (rel-fun (pcr-dnelist op =) (pcr-dnelist op =)) (λf xs. remdups (map f xs)) Applicative-DNEList.map
by (simp add: map-def rel-fun-def dnelist.pcr-cr-eq cr-dnelist-def list-of-dnelist[simplified] Abs-dnelist-inverse)

qualified lift-definition single :: 'a ⇒ 'a dnelist is λx. [x] by simp
qualified lift-definition insert :: 'a ⇒ 'a dnelist ⇒ 'a dnelist is λx xs. if x ∈ set xs then xs else x # xs by auto
qualified lift-definition append :: 'a dnelist ⇒ 'a dnelist ⇒ 'a dnelist is λxs ys. remdups (xs @ ys) by auto
qualified lift-definition bind :: 'a dnelist ⇒ ('a ⇒ 'b dnelist) ⇒ 'b dnelist is λxs f. remdups (List.bind xs f) by auto

abbreviation (input) pure-dnelist :: 'a ⇒ 'a dnelist
where pure-dnelist ≡ single
lift-definition \texttt{ap-dnelist} :: ('a \Rightarrow 'b) dnelist ⇒ 'a dnelist ⇒ 'b dnelist
is \lambda f x. \texttt{remdups (ap-list \ f \ x)}
by (auto simp add: \texttt{ap-list-def})

adhoc-overloading \textit{Applicative}.\texttt{ap \ ap-dnelist}

\textbf{lemma} \texttt{ap-pure-list} [simp]: \texttt{ap-list \ f \ xs} = \texttt{map \ f \ xs}
by (simp add: \texttt{ap-list-def \ List.\ bind-def})

\textbf{context includes} \textit{applicative-syntax}
\begin{proof}
\textbf{lemma} \texttt{ap-pure-dnelist}:
\texttt{pure-dnelist \ (λ x. x)} \texttt{⋄ x} = \texttt{Applicative-\ DNEList.\ map \ f \ x}
by transfer simp
\end{proof}

\begin{proof}
\textbf{proof} –
\textbf{show} \texttt{pure-dnelist \ (λ x. x)} \texttt{⋄ x} = \texttt{x} for \texttt{x} :: 'a dnelist
by transfer simp
\begin{enumerate}
\item \texttt{have *: \texttt{remdups (remdups (remdups ([λg f x. g (f x)] \texttt{}) \texttt{}) \texttt{})} = \texttt{remdups (g \texttt{) \texttt{)})}
\texttt{(is \texttt{?rhs = ?lhs) for g :: ('b ⇒ 'c) list \texttt{ and f :: ('a ⇒ 'b) list \texttt{ and x)}}}
\texttt{by (auto simp add: \texttt{ap-list-def \ List.\ bind-def})}
\item \texttt{unfolding \texttt{ap-list-def \ List.\ bind-def}}
\texttt{by (simp add: \texttt{o-def \ remdups-map-remdups \ remdups-concat-remdups)}}
\item \texttt{also have \texttt{... = \texttt{remdups (concat (map (\texttt{λf. map f x}) (concat (map (\texttt{λx. map \ (\texttt{λy. x (f y)}) \texttt{) g})}))})}}
\texttt{by (simp add: \texttt{remdups-map-remdups \ remdups-concat-remdups \ remdups-concat-map \ map (\texttt{λf. map f x}) f}))})
\texttt{using \texttt{list.pure-B-conv[of g f x] \texttt{unfolding \texttt{remdups-concat-map \}}}}
\texttt{by (simp add: \texttt{ap-list-def \ List.\ bind-def \ o-def})}
\item \texttt{also have \texttt{... = \texttt{?rhs unfolding \texttt{ap-list-def \ List.\ bind-def \}}}}
\texttt{by (simp add: \texttt{remdups-map-remdups \ remdups-concat-map \ remdups-concat-map \ map (\texttt{λf. map f x}) f}))}
\texttt{using \texttt{list.pure-B-conv[of g f x]} \texttt{unfolding \texttt{remdups-concat-map \}}}
\texttt{by (simp add: \texttt{ap-list-def \ List.\ bind-def \ o-def})}
\item \texttt{also have \texttt{... = \texttt{?rhs unfolding \texttt{ap-list-def \ List.\ bind-def \}}}}
\texttt{by (simp add: \texttt{remdups-concat-map \ map (\texttt{λf. map f x}) f}))}
\texttt{finally show \texttt{?thesis \ .}}
\end{enumerate}
\textbf{qed}
\textbf{show} \texttt{pure-dnelist \ (λg f x. g (f x)) \texttt{) ⋄ g \texttt{) \texttt{)}} \texttt{=} \texttt{g \texttt{)}}
\texttt{by transfer\texttt{\ (rule * )}}
\end{proof}
show pure-dnelist f o pure-dnelist x = pure-dnelist (f x) for f :: 'a ⇒ 'b and x

by transfer simp

show f o pure-dnelist x = pure-dnelist (λf. f x) o f for f :: ('a ⇒ 'b) dnelist and x

by transfer(simp add: list.interchange)

have *: remdups (remdups ([λx y. x] o x) o y) = x if x: distinct x and y: distinct y y ≠ []

for x :: 'b list and y :: 'a list

proof –

have remdups (map (λ(x :: 'b) (y :: 'a). x) x) = map (λ(x :: 'b) (y :: 'a). x) x

using that by(simp add: distinct-map inj-on-def fun-eq-iff)

hence remdups (remdups ([λx y. x] o x) o y) = remdups (concat (map (λf. map f y) (map (λx y. x) x)))

by(simp add: ap-list-def List.bind-def del: remdups-id-iff-distinct)

also have ... = x using that

by(simp add: o-def map-replicate-const)(subst remdups-concat-map[symmetric],

simp add: o-def remdups-replicate)

finally show ?thesis .

qed

show pure-dnelist (λx y. x) o x o y = x

for x :: 'b dnelist and y :: 'a dnelist

by transfer(rule *; simp)

qed

- dnelist does not have combinator C, so it cannot have W either.

context begin

private lift-definition x :: int dnelist is [2,3] by simp

private lift-definition y :: int dnelist is [5,7] by simp

private lemma pure-dnelist (λf x. f y x) o pure-dnelist (op *) o x o y ≠ pure-dnelist (op *) o y o x

by transfer(simp add: ap-list-def fun-eq-iff)

end

end

3.1 Monoid

theory Applicative-Monoid imports
Applicative
HOL-Library.Adhoc-Overloading

begin

datatype ('a, 'b) monoid-ap = Monoid-ap 'a 'b

definition (in zero) pure-monoid-add :: 'b ⇒ ('a, 'b) monoid-ap

where pure-monoid-add = Monoid-ap 0

15
fun (in plus) ap-monoid-add :: ('a, 'b ⇒ 'c) monoid-ap ⇒ ('a, 'b) monoid-ap ⇒ ('a, 'c) monoid-ap
where ap-monoid-add (Monoid-ap a1 f) (Monoid-ap a2 x) = Monoid-ap (a1 + a2) (f x)

setup ⟨
fold Sign.add-const-constraint
[@{const-name pure-monoid-add}, SOME (@{typ 'b ⇒ ('a :: monoid-add, 'b) monoid-ap})],
(@{const-name ap-monoid-add}, SOME (@{typ ('a :: monoid-add, 'b ⇒ 'c) monoid-ap ⇒ ('a, 'b) monoid-ap ⇒ ('a, 'c) monoid-ap}))⟩

adhoc-overloading Applicative.pure pure-monoid-add
adhoc-overloading Applicative.ap ap-monoid-add

applicative monoid-add
  for pure: pure-monoid-add
    ap: ap-monoid-add

subgoal by(simp add: pure-monoid-add-def)

subgoal for g f x by(cases g f x rule: monoid-ap.exhaust[case-product monoid-ap.exhaust, case-product monoid-ap.exhaust])(simp add: pure-monoid-add-def add.assoc)

subgoal for x by(cases x)(simp add: pure-monoid-add-def)
subgoal for f x y by(cases f)(simp add: pure-monoid-add-def)
done

applicative comm-monoid-add (C)
  for pure: pure-monoid-add :: - ⇒ (- :: comm-monoid-add, -) monoid-ap
    ap: ap-monoid-add :: (- :: comm-monoid-add, -) monoid-ap ⇒ -
apply(rule monoid-add.homomorphism monoid-add.pure-B-conv monoid-add.interchange)+

subgoal for f x y by(cases f x y rule: monoid-ap.exhaust[case-product monoid-ap.exhaust, case-product monoid-ap.exhaust])(simp add: pure-monoid-add-def add-ac)
apply(rule monoid-add.pure-I-conv)
done

class idemp-monoid-add = monoid-add +
  assumes add-idemp: x + x = x

applicative idemp-monoid-add (W)
  for pure: pure-monoid-add :: - ⇒ (- :: idemp-monoid-add, -) monoid-ap
    ap: ap-monoid-add :: (- :: idemp-monoid-add, -) monoid-ap ⇒ -
apply(rule monoid-add.homomorphism monoid-add.pure-B-conv monoid-add.pure-I-conv)+

subgoal for f x y by(cases f x y rule: monoid-ap.exhaust[case-product monoid-ap.exhaust])(simp add: pure-monoid-add-def add assoc add-idemp)
apply(rule monoid-add.interchange)
done

Test case
lemma
  includes applicative-syntax
  shows pure-monoid-add op + (x :: (nat, int) monoid-ap) ⋄ y = pure op + ⋄ y ⋄ x
  by (applicative-lifting comm-monoid-add) simp
end

3.2 State monad

theory Applicative-State
imports
  Applicative
  HOL‐Library.State‐Monad
begin

applicative state for
pure: State‐Monad.return
ap: State‐Monad.ap
unfolding State‐Monad.return‐def State‐Monad.ap‐def
by (auto split: prod.splits)
end

3.3 Streams as an applicative functor

theory Applicative‐Stream imports
  Applicative
  HOL‐Library.Stream
  HOL‐Library.Adhoc‐Overloading
begin

primcorec (transfer) ap‐stream :: ('a ⇒ 'b) stream ⇒ 'a stream ⇒ 'b stream
where
  shd (ap‐stream f x) = shd f (shd x)
| stl (ap‐stream f x) = ap‐stream (stl f) (stl x)

adhoc‐overloading Applicative.pure sconst
adhoc‐overloading Applicative.ap ap‐stream

context includes lifting‐syntax applicative‐syntax
begin

lemma ap‐stream‐id: pure (λx. x) ⋄ x = x
  by (coinduction arbitrary: x) simp

lemma ap‐stream‐homo: pure f ⋄ pure x = pure (f x)
  by coinduction simp

lemma ap‐stream‐interchange: f ⋄ pure x = pure (λf ′. f x) ⋄ f

17
by (coinduction arbitrary: \( f \)) auto

**lemma** ap-stream-composition: pure \((\lambda g \ x. \ g \ (f \ x)) \circ g \circ f \circ x = g \circ (f \circ x)\)
by (coinduction arbitrary: \( g \ f \ x \)) auto

**applicative stream** \((S, K)\)
for
- pure: sconst
- ap: ap-stream
- rel: stream-all2
- set: sset

**proof** –
fix \( g :: ('b \Rightarrow 'a \Rightarrow 'c) \) stream and \( f x \)
show pure \((\lambda g \ x. \ g \ (f \ x)) \circ g \circ f \circ x = g \circ (f \circ x)\)
by (coinduction arbitrary: \( g \ f \ x \)) auto

**next**
fix \( x :: 'b \) stream and \( y :: 'a \) stream
show pure \((\lambda x\ y. \ x \circ y = x)\)
by (coinduction arbitrary: \( x \ y \)) auto

**next**
fix \( R :: 'a \Rightarrow 'b \Rightarrow bool\)
show \((R \Longrightarrow stream-all2 \ R) \) pure pure
**proof**
fix \( x \ y \)
assume \( R \ x \ y \)
then show \( stream-all2 \ R \ (pure \ x) \ (pure \ y)\)
by coinduction simp
**qed**

**next**
fix \( R \) and \( f :: ('a \Rightarrow 'b) \) stream and \( g :: ('a \Rightarrow 'c) \) stream and \( x \)
assume [transfer-rule]: \( stream-all2 \ (eq-on \ (sset \ x)) \Longrightarrow R) \ f \ g \)
have [transfer-rule]: \( stream-all2 \ (eq-on \ (sset \ x)) \ x \ x \) by (simp add: stream.rel-refl-strong)
show \( stream-all2 \ R \ (f \circ x) \ (g \circ x)\) by transfer-prover
**qed** (rule ap-stream-homo)

**lemma** smap-applicative[applicative-unfold]: smap \( f \ x = pure \ f \circ x\)
**unfolding** ap-stream-def by (coinduction arbitrary: \( x \)) auto

**lemma** smap2-applicative[applicative-unfold]: smap2 \( f \ x \ y = pure \ f \circ x \circ y\)
**unfolding** ap-stream-def by (coinduction arbitrary: \( x \ y \)) auto

end

end

**3.4 Open state monad**

**theory** Applicative-Open-State **imports**
Applicative
HOL-Library.Adhoc-Overloading

begin

type-synonym ('a, 's) state = 's ⇒ 'a × 's

definition ap-state f x = (λs. case f s of (g, s') ⇒ case x s' of (y, s'') ⇒ (g y, s''))

abbreviation (input) pure-state ≡ Pair

adhoc-overloading Applicative.ap ap-state

applicative state
for
  pure: pure-state
  ap: ap-state :: ('a ⇒ 'b, 's) state ⇒ ('a, 's) state ⇒ ('b, 's) state
unfolding ap-state-def
by (auto split: prod.split)
end

3.5 Probability mass functions

theory Applicative-PMF imports
  Applicative
  Probability
  HOL-Library.Adhoc-Overloading
begin

abbreviation (input) pure-pmf :: 'a ⇒ 'a pmf
where pure-pmf ≡ return-pmf

definition ap-pmf :: ('a ⇒ 'b) pmf ⇒ 'a pmf ⇒ 'b pmf
where ap-pmf f x = map-pmf (λ(f, x). f x) (pair-pmf f x)

adhoc-overloading Applicative.ap ap-pmf

context includes applicative-syntax
begin

lemma ap-pmf-id: pure-pmf (λx. x) ◦ x = x
by(simp add: ap-pmf-def pair-return-pmf1 pmf.map-comp o-def)

lemma ap-pmf-comp: pure-pmf op ◦ u ◦ v ◦ w = u ◦ (v ◦ w)
by(simp add: ap-pmf-def pair-return-pmf1 pair-map-pmf1 pair-map-pmf2 pmf.map-comp o-def split-def pair-map-pmf)

lemma ap-pmf-homo: pure-pmf f ◦ pure-pmf x = pure-pmf (f x)
by(simp add: ap-pmf-def pair-return-pmf1)
lemma ap-pmf-interchange: \( u \odot \text{pure-pmf} \ x = \text{pure-pmf} \ (\lambda f. f \ x) \odot u \)
by (simp add: ap-pmf-def pair-return-pmf1 pair-return-pmf2 pmf.map-comp o-def)

lemma ap-pmf-K: \( \text{return-pmf} \ (\lambda x -. x) \odot x \odot y = x \)
by (simp add: ap-pmf-def pair-map-pmf1 pmf.map-comp pair-return-pmf1 o-def split-def map-fst-pair-pmf)

lemma ap-pmf-C: \( \text{return-pmf} \ (\lambda f x y. f y x) \odot f \odot x \odot y = f \odot y \odot x \)
apply (simp add: ap-pmf-def pair-map-pmf1 pmf.map-comp pair-return-pmf1 pair-pair-pmf o-def split-def)
apply (subst (2) pair-commute-pmf)
apply (simp add: pair-map-pmf2 pmf.map-comp o-def split-def)
done

lemma ap-pmf-transfer[transfer-rule]:
rel-fun (rel-pmf (rel-fun A B)) (rel-fun (rel-pmf A) (rel-pmf B)) ap-pmf ap-pmf
unfolding ap-pmf-def[abs-def] pair-pmf-def
by transfer-prover

applicative pmf (C, K)
for
pure: pure-pmf
ap: ap-pmf
rel: rel-pmf
set: set-pmf
proof
  fix R :: 'a ⇒ 'b ⇒ bool
  show rel-fun R (rel-pmf R) pure-pmf pure-pmf by transfer-prover
next
  fix R and f :: ('a ⇒ 'b) pmf and g :: ('a ⇒ 'c) pmf and x
  assume [transfer-rule]: rel-pmf (rel-fun (eq-on (set-pmf x)) R) f g
  have [transfer-rule]: rel-pmf (eq-on (set-pmf x)) x x by (simp add: pmf.rel-refl-strong)
  show rel-pmf R (ap-pmf f x) (ap-pmf g x) by transfer-prover
qed

end

3.6 Probability mass functions implemented as lists with duplicates

theory Applicative-Probability-List imports
  Applicative-List
  Complex-Main
begin

lemma sum-list-concat-map: \( \text{sum-list} \ (\text{concat} \ (\text{map} \ f \ xs)) = \text{sum-list} \ (\text{map} \ (\lambda x. \)


sum-list (f x) xs
by (induction xs) simp-all

context includes applicative-syntax begin

lemma set-ap-list [simp]: set (f x) = (λ(f, x). f x) ' (set f × set x)
by (auto simp add: ap-list-def List.bind-def)

We call the implementation type pfp because it is the basis for the Haskell library Probability by Martin Erwig and Steve Kollmansberger (Probabilistic Functional Programming).

typedef 'a pfp = {xs :: ('a × real) list. (∀ (-, p) ∈ set xs. p > 0) ∧ sum-list (map snd xs) = 1}
proof
  show [(x, 1)] ∈ ?pfp for x by simp
qed

setup-lifting type-definition-pfp

lift-definition pure-pfp :: 'a ⇒ 'a pfp is λx. [(x, 1)]
by simp

lift-definition ap-pfp :: ('a ⇒ 'b) pfp ⇒ 'a pfp ⇒ 'b pfp
is λfs xs. [λ(f, p) (x, q). (f x, p * q)] o fs o xs

proof safe
  fix xs :: (('a ⇒ 'b) × real) list and ys :: ('a × real) list
  assume xs: ∀ (x, y) ∈ set xs. 0 < y sum-list (map snd xs) = 1
  and ys: ∀ (x, y) ∈ set ys. 0 < y sum-list (map snd ys) = 1
  let ?ap = [λ(f, p) (x, q). (f x, p * q)] o xs o ys
  show 0 < b if (a, b) ∈ set ?ap for a b using that xs ys
    by (auto intro!: mult-pos-pos)
  show sum-list (map snd ?ap) = 1 using xs ys
    by (simp add: ap-list-def List.bind-def map-concat o-def split-beta sum-list-concat-map
        sum-list-const-mult)
qed

adhoc-overloading Applicative.ap ap-pfp

applicative pfp
for pure: pure-pfp
  ap: ap-pfp

proof −
  show pure-pfp (λx. x) o x = x for x :: 'a pfp
    by transfer (simp add: ap-list-def List.bind-def)
  show pure-pfp f o pure-pfp x = pure-pfp (f x) for f :: 'a ⇒ 'b and x
    by transfer (applicative-lifting; simp)
  show pure-pfp (λa f x. g (f x)) o g o f o x = g o (f o x)
    for g :: ('b ⇒ 'c) pfp and f :: ('a ⇒ 'b) pfp and x
    by transfer (applicative-lifting; clarsimp)
  show f o pure-pfp x = pure-pfp (λf. f x) o f for f :: ('a ⇒ 'b) pfp and x
by transfer(applicative-lifting; clarsimp)
qed
end

3.7 Ultrafilter

theory Applicative-Star imports
  Applicative
  HOL-Nonstandard-Analysis.StarDef
begin

applicative star (C, K, W)
for
  pure: star-of
  ap: Ifun

proof –
  show star-of f * star-of x = star-of (f x) for f x by(fact Ifun-star-of)
qed(transfer; rule refl)+
end

theory Applicative-Vector imports
  Applicative
  HOL-Analysis.Finite-Cartesian-Product
  HOL-Library.Adhoc-Overloading
begin

definition pure-vec :: 'a ⇒ ('a, 'b :: finite) vec
where pure-vec x = (χ - . x)

definition ap-vec :: ('a ⇒ 'b, 'c :: finite) vec ⇒ ('a, 'c) vec ⇒ ('b, 'c) vec
where ap-vec f x = (χ i. (f $ i) (x $ i))

adhoc-overloading Applicative.ap ap-vec

applicative vec (K, W)
for
  pure: pure-vec
  ap: ap-vec
by(auto simp add: pure-vec-def ap-vec-def vec-nth-inverse)

lemma pure-vec-nth [simp]: pure-vec x $ i = x
by(simp add: pure-vec-def)

lemma ap-vec-nth [simp]: ap-vec f x $ i = (f $ i) (x $ i)
by \((\text{simp add: ap-vec-def})\)

end


print-applicative

end

4 Examples of applicative lifting

4.1 Algebraic operations for the environment functor

theory Applicative-Environment-Algebra imports Applicative-Environment HOL-Library.Function-Division begin

Link between applicative instance of the environment functor with the pointwise operations for the algebraic type classes

context includes applicative-syntax begin

lemma plus-fun-af [applicative-unfold]: \(f + g = \text{pure} \circ \circ f \circ g\)

unfolding plus-fun-def const-def apf-def ..

lemma zero-fun-af [applicative-unfold]: \(0 = \text{pure} 0\)

unfolding zero-fun-def const-def ..

lemma times-fun-af [applicative-unfold]: \(f \times g = \text{pure} \circ \circ f \circ g\)

unfolding times-fun-def const-def apf-def ..
lemma one-fun-af [applicative-unfold]: \(1 = \text{pure } 1\)
unfolding one-fun-def const-def ..

lemma of-nat-fun-af [applicative-unfold]: \(\text{of-nat } n = \text{pure } (\text{of-nat } n)\)
unfolding of-nat-fun const-def ..

lemma inverse-fun-af [applicative-unfold]: \(\text{inverse } f = \text{pure } \text{inverse} \circ f\)
unfolding inverse-fun-def o-def const-def apf-def ..

lemma divide-fun-af [applicative-unfold]: \(\text{divide } f g = \text{pure } \text{divide} \circ f \circ g\)
unfolding divide-fun-def const-def apf-def ..

end
end

4.2 Pointwise arithmetic on streams

theory Stream-Algebra
imports Applicative-Stream
begin

instantiation stream :: (zero) zero begin
definition [applicative-unfold]: \(0 = \text{sconst } 0\)
instance ..
end

instantiation stream :: (one) one begin
definition [applicative-unfold]: \(1 = \text{sconst } 1\)
instance ..
end

instantiation stream :: (plus) plus begin
context includes applicative-syntax begin
definition [applicative-unfold]: \(x + y = \text{pure } \text{op} \circ x \circ (y :: \text{'a stream})\)
end
instance ..
end

instantiation stream :: (minus) minus begin
context includes applicative-syntax begin
definition [applicative-unfold]: \(x - y = \text{pure } \text{op} \circ x \circ (y :: \text{'a stream})\)
end
instance ..
end

instantiation stream :: (uminus) uminus begin
context includes applicative-syntax begin
definition [applicative-unfold stream]: \(\text{uminus} = (\text{op} \circ \text{pure } \text{uminus}) :: \text{'a stream}\)
\( \Rightarrow \text{a stream} \)
end
instance .. end

\textbf{instanceation stream :: (times) times begin}
\textbf{context includes} applicative-syntax \textbf{begin}
definition [applicative-unfold]: \( x \ast y = \text{pure op} \odot x \odot (y :: \text{a stream}) \)
end
instance .. end

\textbf{instance stream :: (Rings dvd) Rings dvd ..}

\textbf{instanceation stream :: (modulo) modulo begin}
\textbf{context includes} applicative-syntax \textbf{begin}
definition [applicative-unfold]: \( x \div y = \text{pure op div} \odot x \odot (y :: \text{a stream}) \)
definition [applicative-unfold]: \( x \mod y = \text{pure op mod} \odot x \odot (y :: \text{a stream}) \)
end
instance .. end

\textbf{instance stream :: (semigroup-add) semigroup-add}
\textbf{using} add.assoc \textbf{by intro-classes applicative-lifting}

\textbf{instance stream :: (ab-semigroup-add) ab-semigroup-add}
\textbf{using} add.commute \textbf{by intro-classes applicative-lifting}

\textbf{instance stream :: (semigroup-mult) semigroup-mult}
\textbf{using} mult.assoc \textbf{by intro-classes applicative-lifting}

\textbf{instance stream :: (ab-semigroup-mult) ab-semigroup-mult}
\textbf{using} mult.commute \textbf{by intro-classes applicative-lifting}

\textbf{instance stream :: (monoid-add) monoid-add}
\textbf{by intro-classes} (applicative-lifting, simp)+

\textbf{instance stream :: (comm-monoid-add) comm-monoid-add}
\textbf{by intro-classes} (applicative-lifting, simp)

\textbf{instance stream :: (comm-monoid-diff) comm-monoid-diff}
\textbf{by intro-classes} (applicative-lifting, simp add: diff-diff-add)+

\textbf{instance stream :: (monoid-mult) monoid-mult}
\textbf{by intro-classes} (applicative-lifting, simp)+

\textbf{instance stream :: (comm-monoid-mult) comm-monoid-mult}
\textbf{by intro-classes} (applicative-lifting, simp)
lemma plus-stream-shd: shd (x + y) = shd x + shd y
unfolding plus-stream-def by simp

lemma plus-stream-stl: stl (x + y) = stl x + stl y
unfolding plus-stream-def by simp

instance stream :: (cancel-semigroup-add) cancel-semigroup-add
proof
  fix a b c :: 'a stream
  assume a + b = a + c
  thus b = c proof (coinduction arbitrary: a b c)
    case (Eq-stream a b c)
    hence shd (a + b) = shd (a + c) stl (a + b) = stl (a + c) by simp-all
    thus ?case by (auto simp add: plus-stream-shd plus-stream-stl)
  qed
next
  fix a b c :: 'a stream
  assume b + a = c + a
  thus b = c proof (coinduction arbitrary: a b c)
    case (Eq-stream a b c)
    hence shd (b + a) = shd (c + a) stl (b + a) = stl (c + a) by simp-all
    thus ?case by (auto simp add: plus-stream-shd plus-stream-stl)
  qed
qed

instance stream :: (cancel-associative-cancel) cancel-associative-cancel
by intro-classes (applicative-lifting, simp add: diff-diff-eq)+

instance stream :: (cancel-comm-monoid-add) cancel-comm-monoid-add ..

instance stream :: (group-add) group-add
by intro-classes (applicative-lifting, simp)+

instance stream :: (ab-group-add) ab-group-add
by intro-classes simp-all

instance stream :: (semiring) semiring
by intro-classes (applicative-lifting, simp add: ring-distribs)+

instance stream :: (mult-zero) mult-zero
by intro-classes (applicative-lifting, simp)+

instance stream :: (semiring-0) semiring-0 ..

instance stream :: (semiring-0-cancel) semiring-0-cancel ..

instance stream :: (comm-semiring) comm-semiring
by intro-classes(rule distrib-right)

instance stream :: (comm-semiring-0) comm-semiring-0 ..

instance stream :: (comm-semiring-0-cancel) comm-semiring-0-cancel ..

lemma pure-stream-inject [simp]: sconst x = sconst y ⟷ x = y
proof
  assume sconst x = sconst y
  hence shd (sconst x) = shd (sconst y) by simp
  thus x = y by simp
qed auto

instance stream :: (zero-neq-one) zero-neq-one
by intro-classes (applicative-unfold stream)

instance stream :: (semiring-1) semiring-1 ..

instance stream :: (comm-semiring-1) comm-semiring-1 ..

instance stream :: (semiring-1-cancel) semiring-1-cancel ..

instance stream :: (comm-semiring-1-cancel) comm-semiring-1-cancel
by(intro-classes; applicative-lifting, rule right-diff-distrib')

instance stream :: (ring) ring ..

instance stream :: (comm-ring) comm-ring ..

instance stream :: (ring-1) ring-1 ..

instance stream :: (comm-ring-1) comm-ring-1 ..

instance stream :: (numeral) numeral ..

instance stream :: (neg-numeral) neg-numeral ..

instance stream :: (semiring-numeral) semiring-numeral ..

lemma of-nat-stream [applicative-unfold]: of-nat n = sconst (of-nat n)
proof (induction n)
  case 0 show ?case by (simp add: zero-stream-def del: id-apply)
next
  case (Suc n)
  have 1 + pure (of-nat n) = pure (1 + of-nat n) by applicative-nf rule
  with Suc.IH show ?case by (simp del: id-apply)
qed

instance stream :: (semiring-char-0) semiring-char-0

27
by intro-classes (simp add: inj-on-def of-nat-stream)

lemma pure-stream-numeral [applicative-unfold]: numeral n = pure (numeral n)
by(induction n)(simp-all only: numeral.simps one-stream-def plus-stream-def ap-stream-homo)

instance stream :: (ring-char-0) ring-char-0 ..

4.3 Tree relabelling

theory Tree-Relabelling imports
Applicative-State
Applicative-Option
Applicative-PMF
HOL-Library.Stream
begin

unbundle applicative-syntax
adhoc-overloading Applicative.pure pure-option
adhoc-overloading Applicative.pure State-Monad.return
adhoc-overloading Applicative.ap State-Monad.ap

Hutton and Fulger [4] suggested the following tree relabelling problem as an example for reasoning about effects. Given a binary tree with labels at the leaves, the relabelling assigns a unique number to every leaf. Their correctness property states that the list of labels in the obtained tree is distinct. As observed by Gibbons and Bird [1], this breaks the abstraction of the state monad, because the relabeling function must be run. Although Hutton and Fulger are careful to reason in point-free style, they nevertheless unfold the implementation of the state monad operations. Gibbons and Hinze [2] suggest to state the correctness in an effectful way using an exception-state monad. Thereby, they lose the applicative structure and have to resort to a full monad.

Here, we model the tree relabelling function three times. First, we state correctness in pure terms following Hutton and Fulger. Second, we take Gibbons’ and Bird’s approach of considering traversals. Third, we state correctness effectfully, but only using the applicative functors.

datatype 'a tree = Leaf 'a | Node 'a tree 'a tree

primrec fold-tree :: ('a ⇒ 'b) ⇒ ('b ⇒ 'b ⇒ 'b) ⇒ 'a tree ⇒ 'b
where
fold-tree f g (Leaf a) = f a
| fold-tree f g (Node l r) = g (fold-tree f g l) (fold-tree f g r)

definition leaves :: 'a tree ⇒ nat
where leaves = fold-tree (λ-. 1) (op +)
lemma leaves-simps [simp]:
leaves (Leaf x) = Suc 0
leaves (Node l r) = leaves l + leaves r
by(simp-all add: leaves-def)

4.3.1 Pure correctness statement

definition labels :: 'a tree ⇒ 'a list
where labels = fold-tree (λx. [x]) append

lemma labels-simps [simp]:
labels (Leaf x) = [x]
labels (Node l r) = labels l @ labels r
by(simp-all add: labels-def)

locale labelling =
  fixes fresh :: ('s, 'x) state
begin

declare [[show-variants]]

definition label-tree :: 'a tree ⇒ ('s, 'x tree) state
where label-tree = fold-tree (λ- :: 'a. pure Leaf ⊗ fresh) (λl r, pure Node ⊗ l ⊗ r)

lemma label-tree-simps [simp]:
label-tree (Leaf x) = pure Leaf ⊗ fresh
label-tree (Node l r) = pure Node ⊗ label-tree l ⊗ label-tree r
by(simp-all add: label-tree-def)

primrec label-list :: 'a list ⇒ ('s, 'x list) state
where
  label-list [] = pure []
| label-list (x # xs) = pure (op #) ⊗ fresh ⊗ label-list xs

lemma label-append: label-list (a @ b) = pure (op @) ⊗ label-list a ⊗ label-list b
— The proof lifts the defining equations of op @ to the state monad.
proof (induction a)
case Nil
  show ?case
    unfolding append.simps label-list.simps
    by applicative-nf simp
next
case (Cons a1 a2)
  show ?case
    unfolding append.simps label-list.simps Cons.IH
    by applicative-nf simp
qed
lemma label-tree-list: pure labels ◁ label-tree t = label-list (labels t)
proof (induction t)
  case Leaf show ?case unfolding label-tree-simps labels-simps label-list.simps
    by applicative-nf simp
next
  case Node show ?case unfolding label-tree-simps labels-simps label-append Node.IH[symmetric]
    by applicative-nf simp
qed

We directly show correctness without going via streams like Hutton and Fulger [4].

lemma correctness-pure:
  fixes t :: 'a tree
  assumes distinct: ∀xs :: 'a list. distinct (fst (run-state (label-list xs) s))
  shows distinct (labels (fst (run-state (label-tree t) s)))
using label-tree-list[of t, THEN arg-cong, of λf. run-state f s] assms[of labels t]
by (cases run-state (label-list (labels t)) s)(simp add: State-Monad.ap-def split-beta)

end

4.3.2 Correctness via monadic traversals

Dual version of an applicative functor with effects composed in the opposite order

typedef 'a dual = UNIV :: 'a set morphisms un-B B by blast
setup-lifting type-definition-dual

lift-definition pure-dual :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b dual
is λpure. pure .

lift-definition ap-dual :: (('a ⇒ ('a ⇒ 'b) ⇒ 'b) ⇒ 'a f1 ⇒ 'a f2 ⇒ 'a f) ⇒ 'a f2 dual ⇒ 'a f3 dual ⇒ 'a f dual
is λpure ap1 ap2 f x. ap2 (ap1 (pure (λx f x)) x) f .

type-synonym ('s, 'a) state-rev = ('s, 'a) state dual

definition pure-state-rev :: 'a ⇒ ('s, 'a) state-rev
where pure-state-rev = pure-dual State-Monad.return

definition ap-state-rev :: ('s, 'a ⇒ 'b) state-rev ⇒ ('s, 'a) state-rev ⇒ ('s, 'b) state-rev

adhoc-overloading Applicative.pure pure-state-rev
adhoc-overloading Applicative.ap ap-state-rev

applicative state-rev

for
pure: pure-state-rev
ap: ap-state-rev

unfolding pure-state-rev-def ap-state-rev-def by (transfer, applicative-nf, rule refl) +

type-synonym (s, a) state-rev-rev = (s, a) state-rev dual

definition pure-state-rev : a ⇒ (s, a) state-rev
where pure-state-rev-rev = pure-dual pure-state-rev

definition ap-state-rev : (s, a ⇒ b) state-rev-rev ⇒ (s, a) state-rev-rev ⇒ (s, b) state-rev-rev

adhoc-overloading Applicative.pure pure-state-rev-rev
adhoc-overloading Applicative.ap ap-state-rev-rev

applicative state-rev-rev
for
  pure: pure-state-rev-rev
  ap: ap-state-rev-rev

unfolding pure-state-rev-rev-def ap-state-rev-rev-def by (transfer, applicative-nf, rule refl) +

lemma ap-state-rev-B: B f o B x = B (State-Monad.return (λx. f x) o x o f)
unfolding ap-state-rev-def by (fact ap-dual.abs-eq)

lemma ap-state-rev-pure-B: pure f o B x = B (State-Monad.return f o x)
unfolding ap-state-rev-def pure-state-rev-def
by transfer(applicative-nf, rule refl)

lemma ap-state-rev-rev-B: B f o B x = B (pure-state-rev (λx. f x) o x o f)
unfolding ap-state-rev-rev-def by (fact ap-dual.abs-eq)

lemma ap-state-rev-rev-pure-B: pure f o B x = B (pure-state-rev f o x)
unfolding ap-state-rev-rev-def pure-state-rev-rev-def
by transfer(applicative-nf, rule refl)

The formulation by Gibbons and Bird [1] crucially depends on Kleisli composition, so we need the state monad rather than the applicative functor only.

lemma ap-cone-bind-state: State-Monad.ap f x = State-Monad.bind f (λf. State-Monad.bind x (State-Monad.return o f))
by (simp add: State-Monad.ap-def State-Monad.bind-def Let-def split-def o-def fun-eq-iff)

lemma ap-pure-bind-state: pure x o State-Monad.bind y f = State-Monad.bind y (ap o (pure x) o f)
by (simp add: ap-cone-bind-state o-def)
definition kleisli-state :: ('b ⇒ ('s, 'c) state) ⇒ ('a ⇒ ('s, 'b) state) ⇒ 'a ⇒ ('s, 'c) state (infixl · 55)
where [simp]: kleisli-state g f a = State-Monad.bind (f a) g

definition fetch :: ('a stream, 'a) state
where fetch = State-Monad.bind State-Monad.get (λs. State-Monad.bind (State-Monad.set (shd s)) (λ_. State-Monad.return (shd s)))

primrec traverse :: ('a ⇒ ('s, 'b) state) ⇒ 'a tree ⇒ ('s, 'b tree) state
where
  traverse f (Leaf x) = pure Leaf ○ f x
| traverse f (Node l r) = pure Node ○ traverse f l ○ traverse f r

As we cannot abstract over the applicative functor in definitions, we define traversal on the transformed applicative function once again.

primrec traverse-rev :: ('a ⇒ ('s, 'b) state-rev) ⇒ 'a tree ⇒ ('s, 'b tree) state-rev
where
  traverse-rev f (Leaf x) = pure Leaf ○ f x
| traverse-rev f (Node l r) = pure Node ○ traverse-rev f l ○ traverse-rev f r

definition recurse :: ('a ⇒ ('s, 'b) state) ⇒ 'a tree ⇒ ('s, 'b tree) state
where recurse f = un-B ○ traverse-rev (B ○ f)

lemma recurse-Leaf: recurse f (Leaf x) = pure Leaf ○ f x
unfolding recurse-def traverse-rev-simps o-def ap-state-rev-pure-B
by(simp add: B-inverse)

lemma recurse-Node:
  recurse f (Node l r) = pure (λr l. Node l r) ○ recurse f l ○ recurse f l
proof –
  have recurse f (Node l r) = un-B (pure Node ○ traverse-rev (B ○ f) l ○ traverse-rev (B ○ f) r)
  by(simp add: recurse-def)
  also have ... = un-B (B (pure Node) ○ B (recurse f l) ○ B (recurse f r))
  by(simp add: un-B-inverse recurse-def pure-state-rev-def pure-dual-def)
  also have ... = pure (λx f. f x) ○ recurse f r ○ (pure (λx f. f x) ○ recurse f l ○ pure Node)
  by(simp add: ap-state-rev-B B-inverse)
  also have ... = pure (λr l. Node l r) ○ recurse f r ○ recurse f l
  also have ... = pure (λr l. Node l r) ○ recurse f r ○ recurse f l
  finally show ?thesis .
  — This step expands to 13 steps in [1]
  by(applicative-nf) simp
qed

lemma traverse-pure: traverse pure t = pure t
proof(induction t)
  { case Leaf show ?case unfolding traverse.simps by applicative-nf simp }
  { case Node show ?case unfolding traverse.simps Node.IH by applicative-nf simp }

32
qed

$B \circ B$ is an idiom morphism

**Lemma** $B$-pure: $pure \ x = B \ (\text{State-Monad.return} \ x)$

**Unfolding** $pure$-state-rev-def by transfer simp

**Lemma** $BB$-pure: $pure \ x = B \ (B \ (pure \ x))$

**Unfolding** $pure$-state-rev-rev-def $B$-pure [symmetric] by transfer (rule refl)

**Lemma** $BB$-ap: $B \ (B \ f) \circ B \ (B \ x) = B \ (B \ (f \circ x))$

**Proof**
- have $B \ (B \ f) \circ B \ (B \ x) = B \ (B \ (pure \ (\lambda x \ f. f) \circ f \circ \ (pure \ (\lambda x \ f. f)) \circ x \circ \ pure \ (\lambda x \ f. f)))$
  - unfolding $ap$-state-rev-rev-B $B$-pure $ap$-state-rev-B ..
  - also have $?exp = f \circ x$ — This step takes 15 steps in [1].
    - by (applicative-nf) (rule refl)
  - finally show $?thesis$

**Qed**

**Primrec** $traverse$-rev-rev :: $(‘a ⇒ (‘s, ‘b) state-rev-rev) ⇒ ‘a tree ⇒ (‘s, ‘b tree) state-rev-rev$

**Where**
- $traverse$-rev-rev $f$ (Leaf $x$) = pure Leaf $\circ f \ x$
- $traverse$-rev-rev $f$ (Node $l \ r$) = pure Node $\circ traverse$-rev-rev $f \ l \circ traverse$-rev-rev $f \ r$

**Definition** $recurse$-rev :: $(‘a ⇒ (‘s, ‘b) state-rev) ⇒ ‘a tree ⇒ (‘s, ‘b tree) state-rev$

**Where**
- $recurse$-rev $f$ = $un-B \circ traverse$-rev-rev ($B \circ f$)

**Lemma** $traverse-B$-B: $traverse$-rev-rev ($B \circ B \circ f$) = $B \circ B \circ traverse$ $f$ (is $?lhs = ?rhs$

**Proof**
- fix $t$
  - show $?lhs \ t = ?rhs \ t$ by (induction $t$) (simp-all add: $BB$-pure $BB$-ap)

**Qed**

**Lemma** $traverse$-recurse: $traverse$ $f$ = $un-B \circ$ $recurse$-rev ($B \circ f$) (is $?lhs = ?rhs$

**Proof**
- have $?lhs = un-B \circ un-B \circ B \circ B \circ traverse$ $f$ by (simp add: $a$-def $B$-inverse)
  - also have $… = un-B \circ un-B \circ traverse$-rev-rev ($B \circ B \circ f$) unfolding $traverse$-B-B by (simp add: $a$-assoc)
  - also have $… = ?rhs$ by (simp add: $recurse$-rev-def $a$-assoc)
  - finally show $?thesis$

**Qed**

**Lemma** $recurse$-traverse:
- assumes $f \cdot g = pure$
- shows $recurse$ $f \cdot traverse$ $g = pure$
— Gibbons and Bird impose this as an additional requirement on traversals, but they write that they have not found a way to derive this fact from other axioms. So we prove it directly.

proof
fix t
from assms have *: \( \forall x. \text{State-Monad.bind} \ (g \ x) \ f = \text{State-Monad.return} \ x \)
by(simp add: fun-eq-iff)
hence **: \( \forall x. \text{State-Monad.bind} \ (g \ x) \ (\lambda x. \text{State-Monad.bind} \ (f \ x) \ h) = h \ x \)
by(fold State-Monad.bind-assoc(simp))
show (recurse \( f \cdot \text{traverse} \ g \)) \( t = \text{pure} \ t \)
unfolding kleisli-state-def
proof(induction \( t \))
case (Leaf \( x \))
show ?case
by(simp add: ap-conv-bind-state recurse-Leaf **)
next
case (Node \( l \ r \))
show ?case
Node.IH)
qed
qed

Apply traversals to labelling

**definition** strip :: \( \alpha \times \beta \Rightarrow (\beta \text{ stream}, \alpha \text{ stream}) \text{ state} \)
where strip = \((\lambda (a, b). \text{State-Monad.bind} \ (\text{State-Monad.update} \ (\text{SCons} \ b)) \ (\lambda-. \text{State-Monad.return} \ a))\)

**definition** adorn :: \( \alpha \Rightarrow (\beta \text{ stream}, \alpha \times \beta \text{ state}) \)
where adorn \( a \) = pure \((\text{Pair} \ a) \circ \text{fetch}\)

**abbreviation** label :: \( \alpha \text{ tree} \Rightarrow (\beta \text{ stream}, (\alpha \times \beta) \text{ tree}) \text{ state} \)
where label \equiv \text{traverse} \text{ adorn}

**abbreviation** unlabel :: \( (\alpha \times \beta) \text{ tree} \Rightarrow (\beta \text{ stream}, \alpha \text{ tree}) \text{ state} \)
where unlabel \equiv \text{recurse} \text{ strip}

**lemma** strip-adorn: strip \( \cdot \) adorn = pure

**lemma** correctness-monadic: unlabel \( \cdot \) label = pure
by(rule recurse-traverse)(rule strip-adorn)

**4.3.3 Applicative correctness statement**

Repeating an effect

**primrec** repeatM :: nat \Rightarrow (\'s, \'x \text{ stream}) \Rightarrow (\'s, \'x \text{ list} \text{ state})
where
repeatM 0 \( f \) = State-Monad.return \[
| \text{repeatM} \ (\text{Suc} \ n) \ f = \text{pure} \ \text{op} \# \circ f \circ \text{repeatM} \ n \ f \]
lemma repeatM-plus: repeatM (n + m) f = pure append ∘ repeatM n f ∘ repeatM m f
by(induction n)(simp; applicative-nf; simp)+

abbreviation (input) fail :: 'a option where fail ≡ None

definition lift-state :: ('s, 'a state ⇒ ('s, 'a option) state)
where [applicative-unfold]: lift-state x = pure pure ∘ x

definition lift-option :: 'a option ⇒ ('s, 'a option) state
where [applicative-unfold]: lift-option x = pure x

fun assert :: ('a ⇒ bool) ⇒ 'a option ⇒ 'a option
where
  assert-fail: assert P fail = fail
| assert-pure: assert P (pure x) = (if P x then pure x else fail)

context labelling begin

abbreviation symbols :: nat ⇒ ('s, 'a list option) state
where symbols n ≡ lift-state (repeatM n fresh)

abbreviation (input) disjoint :: 'x list ⇒ 'x list ⇒ bool
where disjoint xs ys ≡ set xs ∩ set ys = {}

definition dlabels :: 'x tree ⇒ 'x list option
where
dlabels = fold-tree (λx. pure [x])
  (λl r. pure (case-prod append) ∘ (assert (case-prod disjoint) (pure Pair ∘ l ∘ r)))

lemma dlabels-simps [simp]:
dlabels (Leaf x) = pure [x]
dlabels (Node l r) = pure (case-prod append) ∘ (assert (case-prod disjoint) (pure Pair ∘ dlabels l ∘ dlabels r))
by(simp-all add: dlabels-def)

lemma correctness-applicative:
  assumes distinct: ∀n. pure (assert distinct) ∘ symbols n = symbols n
  shows State-Monad.return dlabels ∘ label-tree t = symbols (leaves t)
proof(induction t)
  show pure dlabels ∘ label-tree (Leaf x) = symbols (leaves (Leaf x)) for x :: 'a
  unfolding label-tree-simps leaves-simps repeatM.simps by applicative-nf simp
next
  fix l r :: 'a tree
  assume IH: pure dlabels ∘ label-tree l = symbols (leaves l) pure dlabels ∘ label-tree r = symbols (leaves r)
  let ?cat = case-prod append and ?disj = case-prod disjoint
let \( f = \lambda l r. \) pure ?cat \( \circ (\text{assert} \ \text{disj} \ (\text{pure} \ l \circ r)) \)

have State-Monad.return dlabels \( \circ \) label-tree (Node l r) = 
  pure \( f \circ (\text{pure} \ dlabels \circ \text{label-tree} \ l) \circ (\text{pure} \ dlabels \circ \text{label-tree} \ r) \)

unfolding label-tree-simps by applicative-nf simp

also have \( \ldots = \text{pure} \ f \circ (\text{pure} \ (\text{assert} \ \text{distinct}) \circ \text{symbols} \ (\text{leaves} \ l)) \circ (\text{pure} \ (\text{assert} \ \text{distinct}) \circ \text{symbols} \ (\text{leaves} \ r)) \)

unfolding IH distinct ..

also have \( \ldots = \text{symbols} \ (\text{leaves} \ (\text{Node} \ l \ r)) \)

unfolding leaves-simps repeatM-plus by applicative-nf simp

finally show pure dlabels \( \circ \) label-tree (Node l r) = symbols (leaves (Node l r)) .

qed

definitions

4.3.4 Probabilistic tree relabelling

primrec mirror :: 'a tree \Rightarrow 'a tree
where
  mirror (Leaf x) = Leaf x
| mirror (Node l r) = Node (mirror r) (mirror l)

datatype dir = Left | Right

hide-const (open) path

function (sequential) subtree :: dir list \Rightarrow 'a tree \Rightarrow 'a tree
where
  subtree (Left # path) (Node l r) = subtree path l
| subtree (Right # path) (Node l r) = subtree path r
| subtree [] (Leaf x) = Leaf x
| subtree [] t = t
by pat-completeness auto
termination by lexicographic-order

adhoc-overloading Applicative.pure pure-pmf

class context fixes p :: 'a \Rightarrow 'b pmf begin

primrec plabel :: 'a tree \Rightarrow 'b tree pmf
where
  plabel (Leaf x) = pure Leaf \( \circ \) p x
| plabel (Node l r) = pure Node \( \circ \) plabel l \( \circ \) plabel r

lemma plabel-mirror: plabel (mirror t) = pure mirror \( \circ \) plabel t
proof (induction t)
  case (Leaf x)
  show ?case unfolding plabel.simps mirror.simps by (applicative-lifting) simp
next

end
case (Node t1 t2)
show ?case unfolding plabel.simps mirror.simps Node.IH by (applicative-lifting) simp
qed

lemma plabel-subtree: plabel (subtree path t) = pure (subtree path) ◦ plabel t
proof (induction path t rule: subtree.induct)
case Left: (1 path l r)
  show ?case unfolding plabel.simps subtree.simps Left.IH by (applicative-lifting) simp
next
case Right: (2 path l r)
  show ?case unfolding plabel.simps subtree.simps Right.IH by (applicative-lifting) simp
next
case (3 uu x)
  show ?case unfolding plabel.simps subtree.simps by (applicative-lifting) simp
next
case (4 v va)
  show ?case unfolding plabel.simps subtree.simps by (applicative-lifting) simp
qed
end
end

theory Applicative-Examples imports
  Applicative-Environment-Algebra
  Stream-Algebra
  Tree-Relabelling
begin
end

5 Formalisation of idiomatic terms and lifting

5.1 Immediate joinability under a relation

theory Joinable imports Main begin

5.1.1 Definition and basic properties
definition joinable :: ('a × 'b) set ⇒ ('a × 'a) set
where joinable R = {(x, y). ∃ z. (x, z) ∈ R ∧ (y, z) ∈ R}

lemma joinable-simp: (x, y) ∈ joinable R =⇒ (∃ z. (x, z) ∈ R ∧ (y, z) ∈ R)
unfolding joinable-def by simp

lemma joinableI: \((x, z) \in R \implies (y, z) \in R \implies (x, y) \in \text{joinable } R\)
unfolding joinable-simp by blast

lemma joinableD: \((x, y) \in \text{joinable } R \implies \exists z. (x, z) \in R \land (y, z) \in R\)
unfolding joinable-simp .

lemma joinableE:
  assumes \((x, y) \in \text{joinable } R\)
  obtains \(z\) where \((x, z) \in R \land (y, z) \in R\)
using assms unfolding joinable-simp by blast

lemma refl-on-joinable: refl-on \(\{x. \exists y. (x, y) \in R\}\) (joinable R)
by (auto intro!: refl-onI simp only: joinable-simp)

lemma refl-joinable-iff: \((\forall x. \exists y. (x, y) \in R) = \text{refl} (\text{joinable } R)\)
by (auto intro!: refl-onI dest: refl-onD simp add: joinable-simp)

lemma refl-joinable: refl R \subseteq \text{joinable } R
using refl-joinable-iff by (blast dest: refl-onD)

lemma sym-joinable: sym (joinable R)
by (auto intro!: symI simp only: joinable-simp)

lemma joinable-sym: \((x, y) \in \text{joinable } R \implies (y, x) \in \text{joinable } R\)
using sym-joinable by (rule symD)

lemma joinable-mono: \(R \subseteq S \implies \text{joinable } R \subseteq \text{joinable } S\)
by (rule subrelI) (auto simp only: joinable-simp)

lemma refl-le-joinable:
  assumes refl R
  shows \(R \subseteq \text{joinable } R\)
proof (rule subrelI)
  fix x y
  assume \((x, y) \in R\)
  moreover from refl R have \((y, y) \in R\) by (blast dest: refl-onD)
  ultimately show \((x, y) \in \text{joinable } R\) by (rule joinableI)
qed

lemma joinable-subst:
  assumes \(R\)-subst: \(\forall x. (x, y) \in R \implies (P x, P y) \in R\)
  assumes joinable: \((x, y) \in \text{joinable } R\)
  shows \((P x, P y) \in \text{joinable } R\)
proof –
5.1.2 Confluence

definition confluent :: 'a rel ⇒ bool
where
confluent R ≜ (∀x y y'. (x, y) ∈ R ∧ (x, y') ∈ R ↦ (y, y') ∈ joinable R)

lemma confluentI:
(∀x y y'. (x, y) ∈ R ⇒ (x, y') ∈ R ⇒ ∃z. (y, z) ∈ R ∧ (y', z) ∈ R) ⇒
confluent R
unfolding confluent-def by (blast intro: joinableI)

lemma confluentD:
confluent R ⇒ (x, y) ∈ R ⇒ (x, y') ∈ R ⇒ (y, y') ∈ joinable R
unfolding confluent-def by blast

lemma confluentE:
assumes confluent R and (x, y) ∈ R and (x, y') ∈ R
obtains z where (y, z) ∈ R and (y', z) ∈ R
using assms unfolding confluent-def by (blast elim: joinableE)

lemma trans-joinable:
assumes trans R and confluent R
shows trans (joinable R)
proof (rule transI)
fix x y z
assume (x, y) ∈ joinable R
then obtain u where xu: (x, u) ∈ R and yu: (y, u) ∈ R by (rule joinableE)
assume (y, z) ∈ joinable R
then obtain v where yv: (y, v) ∈ R and zv: (z, v) ∈ R by (rule joinableE)
from yu yv (confluent R) obtain w where uw: (u, w) ∈ R and vw: (v, w) ∈ R
by (blast elim: confluentE)
from xu uw (trans R) have (x, w) ∈ R by (blast elim: transE)
moreover from zv vw (trans R) have (z, w) ∈ R by (blast elim: transE)
ultimately show (x, z) ∈ joinable R by (rule joinableI)
qed

5.1.3 Relation to reflexive transitive symmetric closure

lemma joinable-le-rtscl: joinable (R\*) ⊆ (R ∪ R\(^{-1}\))\*
proof (rule subrelI)
fix x y
assume (x, y) ∈ joinable (R\*)
then obtain z where xz: (x, z) ∈ R\* and yz: (y, z) ∈ R\* by (rule joinableE)
from \(xz\) have \((x, z) \in (R \cup R^{-1})^*\) by (blast intro: in-rtrancl-UnI)
moreover from \(yz\) have \((z, y) \in (R \cup R^{-1})^*\) by (blast intro: in-rtrancl-UnI in-rtrancl-converseI)
ultimately show \((x, y) \in (R \cup R^{-1})^*\) by (rule rtrancl-trans)
qed

theorem joinable-eq-rtscl:
assumes confluent \((R^*)\)
sows joinable \((R^*) = (R \cup R^{-1})^*\)
proof
show joinable \((R^*) \subseteq (R \cup R^{-1})^*\) using joinable-le-rtscl.
next
show joinable \((R^*) \supseteq (R \cup R^{-1})^*\) proof (rule subrelI)
fix \(x y\)
assume \((x, y) \in (R \cup R^{-1})^*\)
thus \((x, y) \in \text{joinable} \((R^*)\)\) proof (induction set: rtrancl)
case base
show \((x, x) \in \text{joinable} \((R^*)\)\) using joinable-refl refl-rtrancl.
next
case (step \(y z\))
have \(R \subseteq \text{joinable} \((R^*)\)\) using refl-le-joinable refl-rtrancl by fast
with \((y, z) \in R \cup R^{-1}\) have \((y, z) \in \text{joinable} \((R^*)\)\) using joinable-sym by fast
with \((x, y) \in \text{joinable} \((R^*)\)\) show \((x, z) \in \text{joinable} \((R^*)\)\)
using trans-joinable trans-rtrancl (confluent \((R^*)\)\) by (blast dest: transD)
qed
qed

5.1.4 Predicate version

definition joinablep :: \('(a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool\) where joinablep \(P x y \leftrightarrow (\exists z. P x z \land P y z)\)

lemma joinablep-joinable[pred-set-conv]:
  joinablep \((\lambda x y. (x, y) \in R) = (\lambda x y. (x, y) \in \text{joinable} R)\)
by (fastforce simp only: joinablep-def joinable-simp)

lemma reflp-joinablep: reflp \(P \Rightarrow \text{joinablep} \(P)\)
by (blast intro: reflpI joinable-refl[to-pred] refl-onI[to-pred] dest: reflpD)

lemma joinablep-refl: reflp \(P \Rightarrow \text{joinablep} \(P x x\)
using reflp-joinablep by (rule reflpD)

lemma reflp-le-joinablep: reflp \(P \Rightarrow P \leq \text{joinablep} \(P\)
by (blast intro: reflp-joinablep[to-pred] refl-onI[to-pred] dest: reflpD)

end
5.2 Combined beta and eta reduction of lambda terms

theory Beta-Eta
imports HOL - Proofs - Lambda.Eta.Joinable
begin

5.2.1 Auxiliary lemmas

lemma liftn-lift-swap: liftn n (lift t k) k = lift (liftn n t k) k
by (induction n) simp-all

lemma subst-liftn:
  i ≤ n + k ∧ k ≤ i ⇒ (liftn (Suc n) s k)[t/i] = liftn n s k
by (induction s arbitrary: i k t) auto

lemma subst-lift2[simp]: (lift (lift t 0) 0)[x/Suc 0] = lift t 0
proof
  have lift (lift t 0) 0 = lift (lift t 0) (Suc 0) using lift-lift by simp
  thus ?thesis by simp
qed

lemma free-liftn:
  free (liftn n t k) i = (i < k ∧ free t i ∨ k + n ≤ i ∧ free t (i − n))
by (induction t arbitrary: k i) (auto simp add: Suc-diff-le)

5.2.2 Reduction

abbreviation beta-eta :: dB ⇒ dB ⇒ bool (infixl →βη 50)
where beta-eta ≡ sup beta eta

abbreviation beta-eta-reds :: dB ⇒ dB ⇒ bool (infixl →βη∗ 50)
where s →βη∗ t ≡ (beta-eta)∗∗ s t

lemma beta-into-beta-eta-reds: s →β t ⇒ s →βη∗ t
by auto

lemma eta-into-beta-eta-reds: s →η t ⇒ s →βη∗ t
by auto

lemma beta-reds-into-beta-eta-reds: s →β* t ⇒ s →βη* t
by (auto intro: rtranclp-mono[THEN predicate2D])

lemma eta-reds-into-beta-eta-reds: s →η* t ⇒ s →βη* t
by (auto intro: rtranclp-mono[THEN predicate2D])

lemma beta-eta-appL[intro]: s →βη* s' ⇒ s' t →βη* s' t
by (induction set: rtranclp) (auto intro: rtranclp.rtrancl-into-rtrancl)

lemma beta-eta-appR[intro]: t →βη* t' ⇒ s t →βη* s t'
by (induction set: rtranclp) (auto intro: rtranclp.rtrancl-into-rtrancl)
lemma beta-eta-abs[intro]: \( t \rightarrow_{\beta\eta}^* t' \implies Abs \ t \rightarrow_{\beta\eta}^* Abs \ t' \)
by (induction set: rtranclp) (auto intro: rtranclp.rtrancl_into_rtrancl)

lemma beta-eta-lift: \( s \rightarrow_{\beta\eta}^* t \implies lift \ s \ k \rightarrow_{\beta\eta}^* lift \ t \ k \)
proof (induction pred: rtranclp)
  case base show ?case ..
next
  case (step y z)
    hence \( lift \ y \ k \rightarrow_{\beta\eta} lift \ z \ k \) using lift-preserves-beta eta-lift
    by blast
  with \( \text{step.IH} \) show \( lift \ s \ k \rightarrow_{\beta\eta}^* lift \ z \ k \) by iprover
qed

lemma confluent-beta-eta-reds: Joinable.confluent \{ \( s, t \). \( s \rightarrow_{\beta\eta}^* t \}\)
using confluent-beta-eta
unfolding diamond-def commute-def square-def
by (blast intro!: confluentI)

5.2.3 Equivalence

Terms are equivalent iff they can be reduced to a common term.

definition term-equiv :: dB ⇒ dB ⇒ bool (infixl ↔ 50)
where term-equiv = joinablep beta-eta-reds

lemma term-equivI:
  assumes \( s \rightarrow_{\beta\eta}^* u \) and \( t \rightarrow_{\beta\eta}^* u \)
  shows \( s \leftrightarrow t \)
using assms unfolding term-equiv-def by (rule joinableI[to-pred])

lemma term-equivE:
  assumes \( s \leftrightarrow t \)
  obtains \( u \) where \( s \rightarrow_{\beta\eta}^* u \) and \( t \rightarrow_{\beta\eta}^* u \)
using assms unfolding term-equiv-def by (rule joinableE[to-pred])

lemma reds-into-equiv[elim]: \( s \rightarrow_{\beta\eta}^* t \implies s \leftrightarrow t \)
by (blast intro: term-equivI)

lemma beta-into-equiv[elim]: \( s \rightarrow_\beta t \implies s \leftrightarrow t \)
by (rule reds-into-equiv) (rule beta-into-beta-eta-reds)

lemma eta-into-equiv[elim]: \( s \rightarrow_\eta t \implies s \leftrightarrow t \)
by (rule reds-into-equiv) (rule eta-into-beta-eta-reds)

lemma beta-reds-into-equiv[elim]: \( s \rightarrow_\beta^* t \implies s \leftrightarrow t \)
by (rule reds-into-equiv) (rule beta-reds-into-beta-eta-reds)

lemma eta-reds-into-equiv[elim]: \( s \rightarrow_\eta^* t \implies s \leftrightarrow t \)
by (rule reds-into-equiv) (rule eta-reds-into-beta-eta-reds)
lemma term-refl[iff]: t ↔ t
unfolding term-equiv-def by (blast intro: joinablep-refl reflpI)

lemma term-sym[sym]: (s ↔ t) ⟹ (t ↔ s)
unfolding term-equiv-def by (rule joinable-sym[to-pred])

lemma conversep-term [simp]: (s ↔ t) =⇒ (t ↔ s)
by (blast elim: transpE)

lemma term-trans[trans]: s ↔ t =⇒ t ↔ u =⇒ s ↔ u
by (blast elim: transpE)

lemma term-beta-trans[trans]: s ↔ t =⇒ t → β u =⇒ s ↔ u
by (fast dest!: beta-eta-lift elim: term-equivE)

lemma term-eta-trans[trans]: s ↔ t =⇒ t → η u =⇒ s ↔ u
by (fast dest!: eta-into-beta-eta-reds intro: term-trans)

lemma equiv-appL[intro]: s ↔ s' =⇒ s ° t ↔ s' ° t
unfolding term-equiv-def using beta-eta-appL
by (iprover intro: joinable-subst[to-pred])

lemma equiv-appR[intro]: t ↔ t' =⇒ s ° t ↔ s ° t'
unfolding term-equiv-def using beta-eta-appR
by (iprover intro: joinable-subst[to-pred])

lemma equiv-app: s ↔ s' =⇒ t ↔ t' =⇒ s ° t ↔ s' ° t'
by (blast intro: term-trans)

lemma equiv-abs[intro]: t ↔ t' =⇒ Abs t ↔ Abs t'
unfolding term-equiv-def using beta-eta-abs
by (iprover intro: joinable-subst[to-pred])

lemma equiv-lift: s ↔ t =⇒ lift s k ↔ lift t k
by (auto intro: term-equivI beta-eta-lift elim: term-equivE)

lemma equiv-liftn: s ↔ t =⇒ liftn n s k ↔ liftn n t k
by (induction n) (auto intro: equiv-lift)

Our definition is equivalent to the the symmetric and transitive closure of
the reduction relation.

lemma equiv-eq-rtscl-reds: term-equiv = (sup beta-eta beta-eta^-1)^*
unfolding term-equiv-def
using confluent-beta-eta-reds
by (rule joinable-eq-rtscl[to-pred])

end
5.3 Combinators defined as closed lambda terms

theory Combinators
imports Beta-Eta
begin

definition I-def: I = Abs (Var 0)
definition B-def: B = Abs (Abs (Var 2 ° (Var 1 ° Var 0)))
definition T-def: T = Abs (Var 0 ° Var 1) — reverse application

lemma I-eval: I ° x → β
proof
  have I ° x → β Var 0 [x/0] unfolding I-def ..
  then show thesis by simp
qed

lemma I-equiv [iff]: I ° x ↔ x
using I-eval ..

lemma I-closed [simp]: liftn n I k = I
unfolding I-def by simp

lemma B-eval1: B ° g → β (Abs (Abs (lift g 0 ° (Var 1 ° Var 0))))
proof
  have B ° g → β (Abs (Abs (Var 2 ° (Var 1 ° Var 0))) [g/0] unfolding B-def ..
  then show thesis by simp add: numerals
qed

lemma B-eval2: B ° g ° f → β (Abs (lift g 0 ° (Var 0)))
proof
  have B ° g ° f → β (Abs (Abs (lift g 0 ° (Var 1 ° Var 0)))) ° f
    using B-eval1 by blast
  also have ... → β (Abs (lift g 0 ° (Var 1 ° Var 0)) [f/0] ..
  also have ... = Abs (lift g 0 ° (lift f 0 ° Var 0)) by simp
  finally show thesis .
qed

lemma B-eval: B ° g ° f ° x → β (g ° (f ° x))
proof
  have B ° g ° f ° x → β (Abs (lift g 0 ° (lift f 0 ° Var 0))) ° x
    using B-eval2 by blast
  also have ... → β (lift g 0 ° (lift f 0 ° Var 0)) [x/0] ..
  also have ... = g ° (f ° x) by simp
  finally show thesis .
qed

lemma B-equiv [iff]: B ° g ° f ° x ↔ g ° (f ° x)
using B-eval ..

lemma B-closed [simp]: liftn n B k = B
unfolding $B$-def by simp

lemma $T$-eval1: $T \cdot x \to_{\beta} \text{Abs} \ (\text{Var} \ 0 \ \text{lift} \ x \ 0)$
proof
  have $T \cdot x \to_{\beta} \text{Abs} \ (\text{Var} \ 0 \ \text{Var} \ 1) \ [x/0]$ unfolding $T$-def ..
  then show $\text{thesis}$ by simp
qed

lemma $T$-eval: $T \cdot x \cdot f \to_{\beta^*} f \cdot x$
proof
  have $T \cdot x \cdot f \to_{\beta^*} \text{Abs} \ (\text{Var} \ 0 \ \text{lift} \ x \ 0) \cdot f$
    using $T$-eval1 by blast
  also have $... \to_{\beta} \ (\text{Var} \ 0 \ \text{lift} \ x \ 0) \ [f/0]$ ..
  also have $... = f \cdot x$ by simp
  finally show $\text{thesis}$ .
qed

lemma $T$-equiv[iff]: $T \cdot x \cdot f \leftrightarrow f \cdot x$
using $T$-eval ..

lemma $T$-closed[simp]: liftn n $T \ k = T$
unfolding $T$-def by simp

end

5.4 Idiomatic terms – Properties and operations

theory Idiomatic-Terms
imports Combinators
begin

This theory proves the correctness of the normalisation algorithm for arbitrary applicative functors. We generalise the normal form using a framework for bracket abstraction algorithms. Both approaches justify lifting certain classes of equations. We model this as implications of term equivalences, where unlifting of idiomatic terms is expressed syntactically.

5.4.1 Basic definitions

datatype 'a itrm =
  Opaque 'a | Pure dB
| IAp 'a itrm 'a itrm (infixl $\odot$ 150)
primrec opaque :: 'a itrm $\Rightarrow$ 'a list
where
  opaque (Opaque x) = [x]
| opaque (Pure -) = []
| opaque (f $\odot$ x) = opaque f $\odots$ opaque x
abbreviation \( iorder \ x \equiv \text{length (opaque } x \text{)} \)

inductive \( \text{itrm-cong} :: (\text{\textquotesingle} a \ \text{itrm} \Rightarrow \text{\textquotesingle} a \ \text{itrm} \Rightarrow \text{bool}) \Rightarrow (\text{\textquotesingle} a \ \text{itrm} \Rightarrow \text{\textquotesingle} a \ \text{itrm} \Rightarrow \text{bool} \text{ for } R) \)

where

\[ \begin{align*}
\text{into-itrm-cong} : & \quad R \ x \ y \Longrightarrow \text{itrm-cong} \ R \ x \ y \\
\text{pure-cong}[\text{intro}] : & \quad x \leftrightarrow y \Longrightarrow \text{itrm-cong} \ (\text{Pure } x) \ (\text{Pure } y) \\
\text{ap-cong} : & \quad \text{itrm-cong} \ R \ f \ f' \Longrightarrow \text{itrm-cong} \ R \ x \ x' \Longrightarrow \text{itrm-cong} \ R \ (f \circ x) \ (f' \circ x') \\
\text{itrm-refl}[\text{iff}] : & \quad \text{itrm-cong} \ R \ x \ x \\
\text{itrm-sym}[\text{sym}] : & \quad \text{itrm-cong} \ R \ x \ y \Longrightarrow \text{itrm-cong} \ R \ y \ x \\
\text{itrm-trans}[\text{trans}] : & \quad \text{itrm-cong} \ R \ x \ y \Longrightarrow \text{itrm-cong} \ R \ y \ z \Longrightarrow \text{itrm-cong} \ R \ x \ z
\end{align*} \]

lemma \( \text{ap-congL}[\text{intro}] :: \text{itrm-cong} \ R \ f \ f' \Longrightarrow \text{itrm-cong} \ R \ (f \circ x) \ (f' \circ x) \)

by \( \text{blast intro: ap-cong} \)

lemma \( \text{ap-congR}[\text{intro}] :: \text{itrm-cong} \ R \ x \ x' \Longrightarrow \text{itrm-cong} \ R \ (f \circ x) \ (f \circ x') \)

by \( \text{blast intro: ap-cong} \)

Idiomatic terms are \( \text{similar} \) iff they have the same structure, and all contained lambda terms are equivalent.

abbreviation \( \text{similar} :: (\text{\textquotesingle} a \ \text{itrm} \Rightarrow \text{\textquotesingle} a \ \text{itrm} \Rightarrow \text{bool} \ \text{infixl } \sim = 50) \)

where \( x \sim y \equiv \text{itrm-cong} \ (\lambda - - \text{. False}) \ x \ y \)

lemma \( \text{pure-similarE} :: \text{assumes Pure } x \text{' } \sim = y \text{ obtains } y' \text{ where } y = \text{Pure } y' \text{ and } x' \leftrightarrow y' \)

proof

- def \( x \equiv \text{Pure } x' :: (\text{\textquotesingle} a \ \text{itrm} \\text{def} \)

from \text{assms have } x \sim = y \ text{ unfolding } x-def .

then have \( (\forall x''.\ x = \text{Pure } x'' \longrightarrow (\exists y'.\ y = \text{Pure } y' \land x'' \leftrightarrow y')) \land \\
(\forall x''.\ y = \text{Pure } x'' \longrightarrow (\exists y'.\ x = \text{Pure } y' \land x'' \leftrightarrow y')) \)

proof (induction)

- case \text{pure-cong} thus ?case by (auto intro: term-sym)

next

- case \text{itrm-trans} thus ?case by (fastforce intro: term-trans)

qed simp-all

with \text{that show thesis unfolding } x-def \text{ by blast}

qed

lemma \( \text{opaque-similarE} :: \text{assumes Opaque } x' \equiv y \text{ obtains } y' \text{ where } y = \text{Opaque } y' \text{ and } x' = y' \)

proof

- def \( x \equiv \text{Opaque } x' :: (\text{\textquotesingle} a \ \text{itrm} \\text{def} \)

from \text{assms have } x \equiv y \ text{ unfolding } x-def .

then have \( (\forall x''.\ x = \text{Opaque } x'' \longrightarrow (\exists y'.\ y = \text{Opaque } y' \land x'' = y')) \land \\
(\forall x''.\ y = \text{Opaque } x'' \longrightarrow (\exists y'.\ x = \text{Opaque } y' \land x'' = y')) \)

by induction fast+

46
with that show thesis unfolding x-def by blast
qed

lemma ap-similarE:
  assumes x1 ⋄ x2 ≃ y
  obtains y1 y2 where y = y1 ⋄ y2 and x1 ≃ y1 and x2 ≃ y2
proof -
  from assms have (∀x1' x2'. x1 ⋄ x2 = x1' ⋄ x2' → (∃y1 y2. y = y1 ⋄ y2 ∧ x1' ≃ y1 ∧ x2' ≃ y2)) ∧
    (∀x1' x2'. y = x1' ⋄ x2' → (∃y1 y2. x1 ⋄ x2 = y1 ⋄ y2 ∧ x1' ≃ y1 ∧ x2' ≃ y2))
  proof (induction)
    case ap-cong thus ?case by (blast intro: itrm-sym)
  next
case trans: itrm-trans thus ?case by (fastforce intro: itrm-trans)
qed simp-all

with that show thesis by blast
qed

The following relations define semantic equivalence of idiomatic terms. We consider equivalences that hold universally in all idioms, as well as arbitrary specialisations using additional laws.

inductive idiom-rule :: 'a itrm ⇒ 'a itrm ⇒ bool
where
  idiom-id: idiom-rule (Pure T ⋄ x) x
| idiom-comp: idiom-rule (Pure B ⊠ g ⊠ f ⊠ x) (g ⊠ (f ⋄ x))
| idiom-hom: idiom-rule (Pure f † Pure x) (Pure (f° x))
| idiom-xchng: idiom-rule (f ⊠ Pure x) (Pure (T° x) ⋄ f)

abbreviation itrm-equiv :: 'a itrm ⇒ 'a itrm ⇒ bool (infixl ≃ 50)
where x ≃ y ≡ idiom-rule x y

lemma idiom-rule-into-equiv: idiom-rule x y ⇒ x ≃ y ..

lemmas itrm-id = idiom-id[THEN idiom-rule-into-equiv]
lemmas itrm-comp = idiom-comp[THEN idiom-rule-into-equiv]
lemmas itrm-hom = idiom-hom[THEN idiom-rule-into-equiv]
lemmas itrm-xchng = idiom-xchng[THEN idiom-rule-into-equiv]

lemma similar-into-equiv: x ≃ y ⇒ x ≃ y
by (induction pred: itrm-cong) (auto intro: ap-cong itrm-sym itrm-trans)

lemma opaque-equiv: x ≃ y ⇒ opaque x = opaque y
proof (induction pred: itrm-cong)
case (into-itrm-cong x y)
  thus ?case by induction auto
qed simp-all
lemma iorder-equiv: \( x \simeq y \Rightarrow \text{iorder } x = \text{iorder } y \)
by (auto dest: opaque-equiv)

locale special-idiom = 
  fixes extra-rule :: 'a itrm \Rightarrow 'a itrm \Rightarrow bool
begin

definition idiom-ext-rule = sup idiom-rule extra-rule

abbreviation itrm-ext-equiv :: 'a itrm \Rightarrow 'a itrm \Rightarrow bool (infix \ (=\ +\ 50))
where \( x \simeq y \) \( \equiv \) itrm-cong idiom-ext-rule x y

lemma equiv-into-ext-equiv: \( x \simeq y \Rightarrow x \simeq y \)
unfolding idiom-ext-rule-def
by (induction pred: itrm-cong)
  (auto intro: into-itrm-cong ap-cong itrm-sym itrm-trans)

lemmas itrm-ext-id = itrm-id[THEN equiv-into-ext-equiv]
lemmas itrm-ext-comp = itrm-comp[THEN equiv-into-ext-equiv]
lemmas itrm-ext-hom = itrm-hom[THEN equiv-into-ext-equiv]
lemmas itrm-ext-xchng = itrm-xchng[THEN equiv-into-ext-equiv]

end

5.4.2 Syntactic unlifting

With generalisation of variables  

primrec unlift' :: nat \Rightarrow 'a itrm \Rightarrow nat \Rightarrow dB
where
  unlift' \( n \) (Opaque -) i = Var i
  | unlift' \( n \) (Pure x) i = liftn n x 0
  | unlift' \( n \) (f o x) i = unlift' \( n \) f (i + iorder x) ^ unlift' \( n \) x i

abbreviation unlift x \equiv (Abs``iorder x) (unlift' (iorder x) x 0)

lemma funpow-Suc-inside: (f ^^ Suc n) x = (f ^^ n) (f x)
using funpow-Suc-right unfolding comp-def by metis

lemma absn-cong[intro]: s \leftrightarrow t \( \Rightarrow \) (Abs``n) s \leftrightarrow (Abs``n) t
by (induction n) auto

lemma free-unlift: free (unlift' n x i) j \( \Rightarrow \) j \( \geq \) n \( \lor \) (j \( \geq \) i \( \land \) j < i + iorder x)
proof (induction x arbitrary: i)
  case (Opaque x)
  thus ?case by simp
next
case (Pure x)
  thus ?case using free-liftn by simp
next

48
case (IAp x y)
thus ?case by fastforce
qed

lemma unlift-subst: \( j \leq i \land j \leq n \Rightarrow (\text{unlift}' (\text{Suc} \ n) \ t \ (\text{Suc} \ i))[s/j] = \text{unlift}' \ n \ t \ i \)
proof (induction \( t \) arbitrary: \( i \))
  case (Opaque \( x \))
  thus ?case by simp
next
case (Pure \( x \))
  thus ?case using subst-liftn by simp
next
case (IAp x y)
hence \( j \leq i + \text{iorder} \ y \) by simp
with IAp show ?case by auto
qed

lemma unlift'-'equiv: \( x \simeq y \Rightarrow \text{unlift}' \ n \ x \ i \leftrightarrow \text{unlift}' \ n \ y \ i \)
proof (induction arbitrary: \( n \ i \) pred: itrm-cong)
  case (into-itrm-cong \( x \ y \))
  thus ?case
next
case (idiom-id \( x \))
  show ?case using I-equiv [symmetric] by simp
next
case (idiom-comp \( g \ f \ x \))
  let \( \?G = \text{unlift}' \ n \ y \ (i + \text{iorder} \ f + \text{iorder} \ x) \)
  let \( \?F = \text{unlift}' \ n \ f \ (i + \text{iorder} \ x) \)
  let \( \?X = \text{unlift}' \ n \ x \ i \)
  have \( \text{unlift}' \ n \ (\?G \circ (\?F \circ \?X)) i = \?G \circ (\?F \circ \?X) \)
    by (simp add: add.assoc)
  moreover have \( \text{unlift}' \ n \ (\text{Pure} \ ?B \circ \?G \circ \?F \circ \?X) i = \?B \circ \?G \circ \?F \circ \?X \)
    by (simp add: add.commute add.left-commute)
  moreover have \( \?G \circ (\?F \circ \?X) \leftrightarrow \?B \circ \?G \circ \?F \circ \?X \) using B-equiv [symmetric]
  .
    ultimately show ?case by simp
next
case (idiom-hom \( f \ x \))
  show ?case by auto
next
case (idiom-zchng \( f \ x \))
  let \( \?F = \text{unlift}' \ n \ f \ i \)
  let \( \?X = \text{liftn} \ n \ x \ 0 \)
  have \( \text{unlift}' \ n \ (f \circ \text{Pure} \ x) i = \?F \circ \?X \) by simp
  moreover have \( \text{unlift}' \ n \ (\text{Pure} \ (?T \circ \?x) \circ f) i = ?T \circ ?X \circ ?F \) by simp
  moreover have \( \?F \circ ?X \leftrightarrow ?T \circ ?X \circ ?F \) using T-equiv [symmetric]
  .
    ultimately show ?case by simp
qed
next
case pure-cong
thus ?case by (auto intro: equiv-liftn)
next
case (ap-cong f f' x x')
from \(x \simeq x'\) have iorder-eq: iorder x = iorder x' by (rule iorder-equiv)
have unlift' n (f \od x) i = unlift' n f (i + iorder x) \cdot unlift' n x i by simp
moreover have unlift' n (f' \od x') i = unlift' n f' (i + iorder x) \cdot unlift' n x' i
  using iorder-eq by simp
ultimately show ?case using ap-cong.IH by (auto intro: equiv-app)
next
case itrm-refl
thus ?case by simp
next
case itrm-sym
thus ?case using term-sym by simp
next
case itrm-trans
thus ?case using term-trans by blast
qed

lemma unlift-equiv: \(x \simeq y \Rightarrow\) unlift x \leftrightarrow unlift y
proof -
  assume \(x \simeq y\)
  then have unlift' (iorder y) x 0 \leftrightarrow unlift' (iorder y) y 0 by (rule unlift'-equiv)
  moreover from \(x \simeq y\) have iorder x = iorder y by (rule iorder-equiv)
  ultimately show ?thesis by auto
qed

Preserving variables  primrec unlift-vars :: nat \Rightarrow nat \Rightarrow dB
where
  unlift-vars n (Opaque i) = Var i
| unlift-vars n (Pure x) = lifttn n x 0
| unlift-vars n (x \od y) = unlift-vars n x \cdot unlift-vars n y

lemma all-pure-unlift-vars: opaque x = [] = x \simeq Pure (unlift-vars 0 x)
proof (induction x)
  case (Opaque x) then show ?case by simp
next
case (Pure x) then show ?case by simp
next
case (IAp x y)
  then have no-opaque: opaque x = [] opaque y = [] by simp+
  then have unlift-ap: unlift-vars 0 (x \od y) = unlift-vars 0 x \cdot unlift-vars 0 y
    by simp
  from no-opaque IAp.IH have x \od y \simeq Pure (unlift-vars 0 x) \od Pure (unlift-vars 0 y)
    by (blast intro: ap-cong)
  also have ... \simeq Pure (unlift-vars 0 x \cdot unlift-vars 0 y) by (rule itrm-hom)
  also have ... = Pure (unlift-vars 0 (x \od y)) by (simp only: unlift-ap)
finally show ?case.

qed

5.4.3 Canonical forms

inductive-set CF :: 'a itrm set
where
pure-cf[iff]: Pure x ∈ CF
| ap-cf[intro]: f ∈ CF ⇒ f ◦ Opaque x ∈ CF

primrec CF-pure :: 'a itrm ⇒ dB
where
CF-pure (Opaque _) = undefined
| CF-pure (Pure x) = x
| CF-pure (x ◦ _) = CF-pure x

lemma ap-cfD1[dest]: f ◦ x ∈ CF ⇒ f ∈ CF
by (rule CF.cases) auto

lemma ap-cfD2[dest]: f ◦ x ∈ CF ⇒ ∃x'. x = Opaque x'
by (rule CF.cases) auto

lemma opaque-not-cf[simp]: Opaque x ∈ CF ⇒ False
by (rule CF.cases) auto

lemma cf-unlift:
  assumes x ∈ CF
  shows CF-pure x ↔ unlift x
using assms proof (induction set: CF)
case (pure-cf x)
show ?case by simp
next
case (ap-cf f x)
let ?n = iorder f + 1
have unlift (f ◦ Opaque x) = (Absˆˆ?n) (unlift' ?n f 1 · Var 0)
  by simp
also have ... = (Absˆˆiorder f) (Abs (unlift' ?n f 1 · Var 0))
  using funpow-Suc-inside by simp
also have ... ↔ unlift f proof –
  have ¬ free (unlift' ?n f 1) 0 using free-unlift by fastforce
  hence Abs (unlift' ?n f 1 · Var 0) →η (unlift' ?n f 1)[Var 0/0] ..
  also have ... = unlift' (iorder f) f 0
  using unlift-subst by (metis One-nat-def Suc-eq-plus1 le0)
finally show ?thesis
  by (simp add: r-into-rtranclp absn-cong eta-into-equiv)
qed
finally show ?case
  using ap-cf.IH by (auto intro: term-sym term-trans)
qed
lemma cf-similarI:
assumes $x \in CF$ y $\in CF$
  and opaque $x = $ opaque y
  and CF-pure $x \leftrightarrow$ CF-pure y
shows $x \cong y$
using assms proof (induction arbitrary: y)
case (pure-cf $x$)
hence opaque y = [] by auto
with ($y \in CF$) obtain $y'$ where $y = $ Pure $y'$ by cases auto
with pure-cf.prems show ?case by auto
next
case (ap-cf $f$ $x$)
from (opaque $(f \diamond Opaque x) = $ opaque y)
obtain $y1$ $y2$ where opaque y = $y1 \oplus $ y2
  and opaque $f = $ y1 and $[x] = $ y2 by fastforce
from ($[x] = $ y2) obtain $y'$ where y2 = [y'] and x = $y'
  by auto
with ($y \in CF$) and :opaque y = $y1 \oplus $ y2 obtain g
  where opaque g = $y1$ and y-split: y = $g \circ$ Opaque $y'$ g $\in$ CF by cases auto
with ap-cf.prems :opaque f = $y1$)
have opaque $f = $ opaque g CF-pure $f \leftrightarrow$ CF-pure g by auto
with ap-cf.IH ($g \in CF$) have $f \cong$ g by simp
with ap-cf.prems y-split ($x = $ y') show ?case by (auto intro: ap-cong)
qed

lemma cf-similarD:
assumes in-cf: $x \in CF$ y $\in CF$
  and similar: $x \cong y$
shows CF-pure $x \leftrightarrow$ CF-pure y $\land$ opaque $x = $ opaque y
using assms by (blast intro!: similar-into-equiv opaque-equiv cf-unlift unlift-equiv
  intro: term-trans term-sym)

Equivalent idiomatic terms in canonical form are similar. This justifies
speaking of a normal form.

lemma cf-unique:
assumes in-cf: $x \in CF$ y $\in CF$
  and equiv: $x \equiv y$
shows $x \cong y$
using in-cf proof (rule cf-similarI)
from equiv show opaque $x = $ opaque y by (rule opaque-equiv)
next
from equiv have unlift $x \leftrightarrow$ unlift y by (rule unlift-equiv)
thus CF-pure $x \leftrightarrow$ CF-pure y
using cf-unlift[OF in-cf(1)] cf-unlift[OF in-cf(2)]
  by (auto intro: term-sym term-trans)
qed
5.4.4 Normalisation of idiomatic terms

\textbf{primrec} norm-pn :: \( \text{dB} \Rightarrow \text{a itrm} \Rightarrow \text{a itrm} \)

where

\[
\begin{align*}
\text{norm-pn} f (\text{Opaque } x) &= \text{undefined} \\
\text{norm-pn} f (\text{Pure } x) &= \text{Pure } (f \circ x) \\
\text{norm-pn} f (n \circ x) &= \text{norm-pn } (\text{B } \circ f) n \circ x
\end{align*}
\]

\textbf{primrec} norm-nn :: \( \text{a itrm} \Rightarrow \text{a itrm} \Rightarrow \text{a itrm} \)

where

\[
\begin{align*}
\text{norm-nn} n (\text{Opaque } x) &= \text{undefined} \\
\text{norm-nn} n (\text{Pure } x) &= \text{norm-pn } (T \circ x) n \\
\text{norm-nn} n (n' \circ x) &= \text{norm-nn } (\text{norm-pn } B n) n' \circ x
\end{align*}
\]

\textbf{primrec} norm :: \( \text{a itrm} \Rightarrow \text{a itrm} \)

where

\[
\begin{align*}
\text{norm } (\text{Opaque } x) &= \text{Pure } I \circ Opaque x \\
\text{norm } (\text{Pure } x) &= \text{Pure } x \\
\text{norm } (f \circ x) &= \text{norm-nn } (\text{norm } f) (\text{norm } x)
\end{align*}
\]

\textbf{lemma} norm-pn-in-cf:

\textbf{assumes} \( x \in \text{CF} \)

\textbf{shows} \( \text{norm-pn } f x \in \text{CF} \)

\textbf{using} \( \text{assms} \)

\textbf{by} \( \text{(induction } x \text{ arbitrary: } f) \text{ auto} \)

\textbf{lemma} norm-nn-in-cf:

\textbf{assumes} \( n \in \text{CF} \) \( n' \in \text{CF} \)

\textbf{shows} \( \text{norm-nn } n n' \in \text{CF} \)

\textbf{using} \( \text{assms}(2,1) \)

\textbf{by} \( \text{(induction } n' \text{ arbitrary: } n) \text{ (auto intro: norm-pn-in-cf)} \)

\textbf{lemma} norm-in-cf: \( \text{norm } x \in \text{CF} \)

\textbf{by} \( \text{(induction } x \text{) (auto intro: norm-nn-in-cf)} \)

\textbf{lemma} norm-pn-equiv:

\textbf{assumes} \( x \in \text{CF} \)

\textbf{shows} \( \text{norm-pn } f x \simeq \text{Pure } f \circ x \)

\textbf{using} \( \text{assms} \) \textbf{proof} \( \text{(induction } x \text{ arbitrary: } f) \)

\textbf{case} \( \text{pure-cf } x \)

\textbf{have} \( \text{Pure } (f \circ x) \simeq \text{Pure } f \circ \text{Pure } x \) \textbf{using} \( \text{itrn-hom[symmetric]} \).

\textbf{then show} \(?case \text{ by simp}\)

\textbf{next}

\textbf{case} \( \text{ap-cf } n x \)

\textbf{from} \( \text{ap-cf.IH} \) \textbf{have} \( \text{norm-pn } (\text{B } \circ f) n \simeq \text{Pure } (\text{B } \circ f) \circ n \).

\textbf{then have} \( \text{norm-pn } (\text{B } \circ f) n \circ Opaque x \simeq \text{Pure } (\text{B } \circ f) \circ n \circ Opaque x \).

\textbf{also have} \( ... \simeq \text{Pure } \text{B } \circ \text{Pure } f \circ n \circ Opaque x \)

\textbf{using} \( \text{itrn-hom[symmetric]} \) \textbf{by blast}

\text{53}
also have ... \simeq Pure f \circ (n \circ Opaque x) using itrm-comp.

finally show \texttt{?case by simp}

qed

lemma \texttt{norm-nn-equiv}:
assumes \( n \in CF \) \( n' \in CF \)
shows \( \text{norm-nn} n n' \simeq n \circ n' \)
using \texttt{assms(2,1) proof (induction} \( n' \) arbitrary: \( n \))
case \( \text{pure-cf} \ x \)
then have \( \text{norm-pn} (T \circ x) n \simeq Pure \ (T \circ x) \circ n \) by \( \text{rule norm-pn-equiv} \)
also have ... \( \simeq n \circ Pure \ x \) using \texttt{itrm-xchng[symmetric]}.

finally show \texttt{?case by simp}

next
case \( \text{ap-cf} \ n' \ x \)
have \( \text{norm-nn} \ (\text{norm-pn} B \ n) n' \circ Opaque x \simeq Pure \ B \circ n \circ n' \circ Opaque x \)
proof
from \( (n \in CF) \) have \( \text{norm-pn} B \ n \in CF \) by \( \text{rule norm-pn-in-cf} \)
with \texttt{ap-cf.IH} have \( \text{norm-nn} \ (\text{norm-pn} B \ n) n' \simeq \text{norm-pn} B \ n \circ n' \).
also have ... \( \simeq Pure \ B \circ n \circ n' \) using \( \texttt{norm-pn-equiv} \ n \in CF \) by \texttt{blast}
finally show \( \text{norm-nn} \ (\text{norm-pn} B \ n) n' \simeq Pure \ B \circ n \circ n' \).

qed
also have ... \( \simeq n \circ (n' \circ Opaque x) \) using \texttt{itrm-comp}.
finally show \texttt{?case by simp}

qed

lemma \texttt{norm-equiv}:
\( \text{norm} \ x \simeq x \)
proof \( \text{(induction}) \)
case \( \text{Opaque} \ x \)
have \( Pure \ I \circ Opaque \ x \simeq Opaque \ x \) using \texttt{itrm-id}.
then show \texttt{?case by simp}

next
case \( \text{Pure} \ x \)
show \texttt{?case by simp}

next
case \( \text{IAp f} \ x \)
have \( \text{norm} f \in CF \) and \( \text{norm} x \in CF \) by \( \text{rule norm-in-cf} \)+
then have \( \text{norm-nn} \ (\text{norm} f) (\text{norm} x) \simeq \text{norm} f \circ \text{norm} x \)
by \( \text{rule norm-nn-equiv} \)
also have ... \( \simeq f \circ x \) using \texttt{IAp.IH}.
finally show \texttt{?case by simp}

qed

lemma \texttt{normal-form}: obtains \( n \) where \( n \simeq x \) and \( n \in CF \)
using \( \texttt{norm-equiv} \) \( \texttt{norm-in-cf} \).

5.4.5 Lifting with normal forms

lemma \texttt{nf-unlift}:
assumes \( \text{equiv} \ n \simeq x \) and \( \text{cf} \ n \in CF \)
shows $\text{CF-pure } n \leftrightarrow \text{unlift } x$

proof –
from $\text{cf}$ have $\text{CF-pure } n \leftrightarrow \text{unlift } n$ by (rule $\text{cf-unlift}$)
also from $\text{equiv}$ have $\text{unlift } n \leftrightarrow \text{unlift } x$ by (rule $\text{unlift-equiv}$)
finally show $\text{thesis}$ .

qed

theorem $\text{nf-lifting}$:
assumes opaque: $\text{opaque } x = \text{opaque } y$
and base-eq: $\text{unlift } x \leftrightarrow \text{unlift } y$
shows $x \simeq y$

proof –
obtain $n$ where $\text{nf-x}$: $n \simeq x$ $n \in \text{CF}$ by (rule $\text{normal-form}$)
obtain $n'$ where $\text{nf-y}$: $n' \simeq y$ $n' \in \text{CF}$ by (rule $\text{normal-form}$)
from $\text{nf-x(1)}$ have opaque $n$ = opaque $x$ by (rule $\text{opaque-equiv}$)
also note opaque
also from $\text{nf-y(1)}$ have opaque $y$ = opaque $n'$ by (rule $\text{opaque-equiv[THEN sym]}$)
finally have pure-eq: $\text{CF-pure } n \leftrightarrow \text{CF-pure } n'$.

from $\text{nf-x(1)}$ have opaque $n$ = opaque $x$ by (rule $\text{opaque-equiv}$)
also note opaque
also from $\text{nf-y(1)}$ have opaque $y$ = opaque $n'$ by (rule $\text{opaque-equiv[THEN sym]}$)
finally have opaque-eq: opaque $n$ = opaque $n'$.

from $\text{nf-x(1)}$ have $x \simeq n$ ..
also have $n \simeq n'$
using $\text{nf-x nf-y pure-eq opaque-eq}$
by (blast intro: similar-into-eqv cf-similarI)
also from $\text{nf-y(1)}$ have $n' \simeq y$ .
finally show $x \simeq y$ .

qed

5.4.6 Bracket abstraction, twice

Preliminaries: Sequential application of variables
definition $\text{frees :: } dB \Rightarrow \text{nat set}$
where $\text{[simp]}$: $\text{frees } t = \{i. \text{ free } t \ i\}$

definition $\text{var-dist :: } \text{nat list} \Rightarrow dB \Rightarrow dB$
where $\text{var-dist = fold (}\lambda i t. t \cdot \text{ Var } i\)$

lemma $\text{var-dist-Nil[simp]}$: $\text{var-dist } [] t = t$
unfolding $\text{var-dist-def by simp}$

lemma $\text{var-dist-Cons[simp]}$: $\text{var-dist } (v \# vs) t = \text{var-dist } vs (t \cdot \text{ Var } v)$
unfolding $\text{var-dist-def by simp}$
lemma var-dist-append1: var-dist (vs @ [v]) t = var-dist vs t · Var v
unfolding var-dist-def by simp

lemma var-dist-frees: frees (var-dist vs t) = frees t ∪ set vs
by (induction vs arbitrary: t) auto

lemma var-dist-subst-lt:
∀ v ∈ set vs. i < v ⇒ (var-dist vs s)[t/i] = var-dist (map (λv. v - 1) vs) (s[t/i])
by (induction vs arbitrary: s) simp-all

lemma var-dist-subst-gt:
∀ v ∈ set vs. v < i ⇒ (var-dist vs s)[t/i] = var-dist vs (s[t/i])
by (induction vs arbitrary: s) simp-all

definition vsubst :: nat ⇒ nat ⇒ nat ⇒ nat
where vsubst u v w = (if u < w then u else if u = w then v else u - 1)

lemma vsubst-subst: (Var u)[Var v/w] = Var (vsubst u v w)
unfolding vsubst-def by simp

lemma vsubst-subst-lt: u < w ⇒ vsubst u v w = u
unfolding vsubst-def by simp

lemma var-dist-cong: s ↔ t =⇒ var-dist vs s ↔ var-dist vs t
by (induction vs arbitrary: s t) auto

Preliminaries: Eta reductions with permuted variables

lemma absn-subst:
((Abs^n) s)[t/k] = (Abs^n) (s[liftn n t 0/k+n])
by (induction n arbitrary: t k) (simp-all add: liftn-lift-swap)

lemma absn-beta-equiv: (Abs^n Suc n) s ° t ↔ (Abs^n) (s[liftn n t 0/n])
proof –
  have (Abs^n Suc n) s ° t = Abs ((Abs^n) s) ° t by simp
  also have ... ↔ ((Abs^n) s)[t/0] by (rule beta-into-equiv) (rule beta.beta)
  also have ... = (Abs^n) (s[liftn n t 0/n]) by (simp add: absn-subst)
  finally show ?thesis .
qed

lemma absn-dist-eta: (Abs^n) (var-dist (rev [0..<n])) (liftn n t 0) ↔ t
proof (induction n)
case 0 show ?case by simp
next
case (Suc n)
  let ?dist-range = λa k. var-dist (rev [a..<k]) (liftn k t 0)
  have append: rev [0..<Suc n] = rev [1..<Suc n] @ [0] by (simp add: upt-rec)

56
have \( \text{dist-last}: \text{dist-range } 0 \ (\text{Suc } n) = \text{dist-range } 1 \ (\text{Suc } n) \ 	ext{Var } 0 \)

unfolding append \( \text{var-dist-append1} \).

have \( \neg \text{free (dist-range } 1 \ (\text{Suc } n)) \ 0 \) proof

  have \( \text{frees (dist-range } 1 \ (\text{Suc } n)) = \text{frees (liftn } (\text{Suc } n) \ 0) \cup \{1..n\} \)

unfolding \( \text{var-dist-frees} \) by fastforce

then have \( 0 \notin \text{frees (dist-range } 1 \ (\text{Suc } n)) \) by simp

then show \( \text{thesis} \) by simp

qed

then have \( \text{Abs (dist-range } 0 \ (\text{Suc } n)) \rightarrow \text{eta (dist-range } 1 \ (\text{Suc } n)) \[\text{Var } 0/0\] \)

unfolding dist-last by (rule eta)

also have \( \ldots = \text{var-dist (rev } [0..<n]) \ (\text{liftn } (\text{Suc } n) \ 0)[\text{Var } 0/0]\) proof

have \( \forall v \in \text{set (rev } 1..<\text{Suc } n) \). \( 0 < v \) by auto

moreover have \( \text{rev } 0..<\text{Suc } n = \text{map } (\lambda v. v - 1) \ (\text{rev } 1..<\text{Suc } n) \) by (induction \( n \)) simp-all

ultimately show \( \text{thesis} \) by (simp only: var-dist-subst-lt)

qed

also have \( \ldots = \text{dist-range } 0 \ n \) using subst-liftn \( [\text{of } 0 \ n \ 0 \ t \ \text{Var } 0] \) by simp

finally have \( \text{Abs (dist-range } 0 \ (\text{Suc } n)) \leftrightarrow \text{dist-range } 0 \ n \) ..

then have \( (\text{Abs } ^\_ \text{Suc } n) \ (\text{dist-range } 0 \ (\text{Suc } n)) \leftrightarrow (\text{Abs } ^\_ \text{n}) \ (\text{dist-range } 0 \ n) \)

unfolding funpow-Suc-inside by (rule absn-cong)

also from Suc.IH have \( \ldots \leftrightarrow t \).

finally show \( \text{?case} \).

qed

primrec \( \text{strip-context} :: \text{nat} \Rightarrow \text{dB} \Rightarrow \text{nat} \Rightarrow \text{dB} \)

where

\[
\begin{align*}
\text{strip-context } n \ (\text{Var } i) \ k &= (\text{if } i < k \text{ then } \text{Var } i \text{ else } \text{Var } (i - n)) \\
\text{strip-context } n \ (\text{Abs } t) \ k &= \text{Abs (strip-context } n \ t \ (\text{Suc } k)) \\
\text{strip-context } n \ (s \ ° \ t) \ k &= \text{strip-context } n \ s \ k \ ° \ \text{strip-context } n \ t \ k
\end{align*}
\]

lemma \( \text{strip-context-liftn}: \text{strip-context } n \ (\text{liftn } (m + n) \ t \ k) \ k = \text{liftn } m \ t \ k \)

by (induction \( t \) arbitrary: \( k \)) simp-all

lemma \( \text{liftn-strip-context}: \)

assumes \( \forall i \in \text{frees } t. \ i < k \lor k + n \leq i \)

shows \( \text{liftn } n \ (\text{strip-context } n \ t \ k) \ k = t \)

using \( \text{assms} \) proof (induction \( t \) arbitrary: \( k \))

case \( \text{Abs } t \)

have \( \forall i \in \text{frees } t. \ i < \text{Suc } k \lor \text{Suc } k + n \leq i \) proof

fix \( i \) assume \( \text{free: } i \in \text{frees } t \)

show \( i < \text{Suc } k \lor \text{Suc } k + n \leq i \) proof (cases \( i > 0 \))

assume \( i > 0 \)

with \( \text{free Abs.prems} \) have \( i - 1 < k \lor k + n \leq i - 1 \) by simp

then show \( \text{thesis} \) by arith

qed simp

qed

with \( \text{Abs.IH} \) show \( \text{?case} \) by simp

qed auto
lemma absn-dist-eta-free:
  assumes $$\forall i \in \text{frees } t. \, n \leq i$$
  shows $$(\text{Abs} \ ^n) \ (\text{var-dist} \ (\text{rev}[0..<n]) \ t) \leftrightarrow \text{strip-context} \ n \ t \ 0 \ (\text{is } ?\text{lhs} \ t \leftrightarrow ?\text{rhs})$$
proof
  have ?lhs \ (\text{liftn} \ n \ ?\text{rhs} \ 0) \leftrightarrow ?\text{rhs} \text{ by (rule absn-dist-eta)}
  moreover have \ (\text{liftn} \ n \ ?\text{rhs} \ 0 = \ t)
  using assms \text{ by (auto intro: liftn-strip-context)}
  ultimately show \ ?\text{thesis} \text{ by simp}
qed

definition perm-vars :: \text{nat } \Rightarrow \text{nat list } \Rightarrow \text{bool}
where perm-vars \ n \ vs \leftarrow \text{distinct } vs \land \text{set } vs = \{0..<n\}

lemma perm-vars-distinct: \text{perm-vars } n \ vs \rightleftharpoons \text{distinct } vs
unfolding perm-vars-def by simp

lemma perm-vars-length: \text{perm-vars } n \ vs \rightleftharpoons \text{length } vs = n
unfolding perm-vars-def using distinct-card by force

lemma perm-vars-lt: \text{perm-vars } n \ vs \rightleftharpoons \forall i \in \text{set } vs. \, i < n
using \text{perm-vars-length} \text{ by simp}

lemma perm-vars-nth-lt: \text{perm-vars } n \ vs \rightleftharpoons i < n \rightleftharpoons vs ! i < n
using \text{perm-vars-length} \text{ by simp}

lemma perm-vars-inj-on-nth:
  assumes \text{perm-vars } n \ vs
  shows \text{inj-on} \ (\text{nth } vs) \{0..<n\}
proof (rule inj-onI)
  fix \ i \ j
  assume \ i \in \{0..<n\} \text{ and } j \in \{0..<n\}
  with assms have \ i < \text{length } vs \text{ and } j < \text{length } vs
  using \text{perm-vars-length} \text{ by simp+}
  moreover from assms have \text{distinct } vs \text{ by (rule perm-vars-distinct)}
  moreover assume \ vs ! i = \ vs ! j
  ultimately show \ i = j \text{ using nth-eq-iff-index-eq by blast}
qed

abbreviation perm-vars-inv :: \text{nat } \Rightarrow \text{nat list } \Rightarrow \text{nat}
where \text{perm-vars-inv } n \ vs \ i \equiv \text{the-inv-into } \{0..<n\} \ (\text{op } ! \ vs) \ i

lemma perm-vars-inv-nth:
  assumes \text{perm-vars-inv } n \ vs \ i
and \ i < n
  shows \text{perm-vars-inv } n \ vs \ (vs ! i) = i
using assms \text{ by (auto intro: the-inv-into-f-f perm-vars-inv-nth)}
lemma dist-perm-eta:
assumes perm-vars: perm-vars n vs
obtains vs' where ∃ t. ∀ i∈frees t. n ≤ i ⇒ (Abs°n) (var-dist vs' ((Abs°n) (var-dist vs (liftn n t 0)))) ↔ strip-context n t 0
proof –
def vsubsts ≡ λ n vs' vs.
  map (λv.
    if v < n − length vs' then v
    else if v < n then vs'! (n − v − 1) + (n − length vs')
    else v − length vs') vs

let ?app-vars = λt n vs' vs. var-dist vs' ((Abs°n) (var-dist vs (liftn n t 0)))
{  
  fix t :: dB and vs' :: nat list
  assume partial: length vs' ≤ n

  let ?m = n − length vs'
  have ?app-vars t n vs' vs ↔ (Abs°n t) (var-dist (vsubsts n vs' vs) (liftn ?m t 0))
    using partial proof (induction vs' arbitrary: vs n)
  case Nil
    then have vsubsts n [] vs = vs unfolding vsubsts-def by (auto intro: map-idI)
    then show ?case by simp
next
  case (Cons v vs')
  def n' ≡ n − 1
  have Suc-n': Suc n' = n unfolding n'-def using Cons.prems by simp
  have vs'-length: length vs' ≤ n' unfolding n'-def using Cons.prems by simp
  let ?m' = n' − length vs'
  have ?m'-conv: ?m' = n − length (v # vs') unfolding n'-def by simp

  have ?app-vars t n (v # vs') vs = ?app-vars t (Suc n') (v # vs') vs unfolding Suc-n'..
    also have ... ↔ var-dist vs' ((Abs°Suc n') (var-dist vs (liftn (Suc n') t 0))
    using var-dist-Cons ..
  unfolding Suc-n'[symmetric] vsubsts-def vsubsts-def
  by (auto cong: if-cong)
  then have (var-dist vs (liftn (Suc n') t 0)) [liftn (Suc n') (Var v) 0 / n']
    = var-dist (vsubsts n [v] vs) (liftn n' t 0)
    using var-dist-subst-Var subst-liftn by simp
  then show (Abs°Suc n') (var-dist vs (liftn (Suc n') t 0)) • Var v
    ↔ (Abs°n') (var-dist (vsubsts n [v] vs) (liftn n' t 0))
    by (fastforce intro: absn-beta-equiv THEN term-trans)
  qed
also have ... ↔ (Abs°?m') (var-dist (vsubsts ?m' vs' (vsubsts n [v] vs')) (liftn
\[ m' t 0 \]

using vs'-length Cons.IH by blast
also have \( \cdots = (\text{Abs}''?m') (\text{var-dist} \ (\text{vsubsts} \ n \ (v \# \ vs')) \ (\text{liftn} \ ?m' \ t \ 0)) \)
proof
  have vs\text{substs} \ n' \ vs' (vsubsts (Suc \ n') [v] \ vs) = vsubsts (Suc \ n') (v \# \ vs') \ vs
  unfolding vs\text{substs-def}
  using vs'-length \[\text{[linarith-split-limit=10]}\]
  by auto
  then show \( ?\text{thesis} \) unfolding Suc-n' by simp
qed
finally show \( ?\text{case} \) unfolding m'-conv.
qed

\text{note} partial-appd = this
\text{def} vs' ≡ map (λi. n − perm-vars-inv n vs (n − i − 1) − 1) \[0..<n]\n
from perm-vars have vs-length: length vs = n by (rule perm-vars-length)
have vs'-length: length vs' = n unfolding vs'-def by simp

have map (λv. vs'! (n − v − 1)) vs = rev \[0..<n]\ proof
  have length vs = length (rev \[0..<n]\)
  unfolding vs-length by simp
  then have list-all2 (λv v'. vs'! (n − v − 1) = v') vs (rev \[0..<n]\) proof
    fix i assume i < length vs
    then have i < n unfolding vs-length.
    then have vs!i < n using perm-vars perm-vars-nth-lt by simp
    with \( i < n \) have vs'! (n − vs!i − 1) = n − perm-vars-inv n vs (vs!i)
    − 1
    unfolding vs'-def by simp
  also from \( i < n \) have \( \cdots = n − i − 1 \) using perm-vars perm-vars-inv-nth
  by simp
  also from \( i < n \) have \( \cdots = \text{rev} \[0..<n]\!i \) by (simp add: rev-nth)
  finally show vs'! (n − vs!i − 1) = \text{rev} \[0..<n]\!i .
  qed
then show \( ?\text{thesis} \)
  unfolding list.rel-eq[symmetric]
  using list.rel-map
  by auto
  qed
then have vs'vs: vsubsts n \ vs' vs = \text{rev} \[0..<n]\]
  unfolding vsubsts-def vs'length
  using perm-vars perm-vars-lt
  by (auto intro: map-ext[THEN trans])

let \( ?\text{appd-vars} = \lambda t. \text{var-dist} \ (\text{rev} \[0..<n]\) \ t \)
{\text{fix} \ t}
assume not-free: \( \forall i \in \text{frees} \ t. \ n \leq i \)
have ‹?app-vars t n vs' vs ‹?appd-vars t n for t
  using partial-appd[of ‹vs'›] vs'-length vs'-vs by simp
then have (Absˆˆn) (‹?app-vars t n vs' vs›) ‹(Absˆˆn) (‹?appd-vars t n)
  by (rule absn-cong)
also have ‹... ↔ strip-context n t 0
  using not-free by (rule absn-dist-eta-free)
finally have (Absˆˆn) (‹?app-vars t n vs' vs›) ‹strip-context n t 0 .
} with that show ‹thesis›.
qed

lemma liftn-absn: liftn n ((Absˆˆm) t) k = (Absˆˆm) (liftn n t (k + m))
by (induction m arbitrary: k) auto

lemma liftn-var-dist-lt:
  ∀i∈set vs. i < k ⇒ liftn n (var-dist vs t) k = var-dist vs (liftn n t k)
by (induction vs arbitrary: t) auto

lemma liftn-context-conv: k ≤ k' ⇒ ∀i∈frees t. i < k ∨ k' ≤ i ⇒ liftn n t k
  = liftn n t k'
proof (induction t arbitrary: k k')
case (Abs t)
  have ∀i∈frees t. i < Suc k ∨ Suc k' ≤ i proof
    fix i assume i ∈ frees t
    show i < Suc k ∨ Suc k' ≤ i proof (cases i = 0)
      assume i = 0 then show ‹thesis› by simp
    next
      assume i ≠ 0
      from Abs.prems(2) have ∀i. free t (Suc i) → i < k ∨ k' ≤ i by auto
      then have ∀i. 0 < i ∧ free t i → i - 1 < k ∨ k' ≤ i - 1 by simp
      then have ∀i. 0 < i ∧ free t i → i < Suc k ∨ Suc k' ≤ i by auto
      with ⟨i ≠ 0; i ∈ frees t⟩ show ‹thesis› by simp
    qed
  qed
with Abs.IH Abs.prems(1) show ‹case› by auto
qed auto

lemma liftn-liftn0: ∀i∈frees t. k ≤ i ⇒ liftn n t k = liftn n t 0
using liftn-context-conv by auto

lemma dist-perm-eta-equiv:
  assumes perm-vars: perm-vars n vs
  and not-free: ∀i∈frees s. n ≤ i ∀i∈frees t. n ≤ i
  and perm-equiv: (Absˆˆn) (var-dist vs s) ‹(Absˆˆn) (var-dist vs t)
  shows strip-context n s 0 ‹ strip-context n t 0
proof −
  from perm-vars have vs-lt-n: ∀i∈set vs. i < n using perm-vars-lt by simp
  obtain vs'where
    etas: ‹∀t. ∀i∈frees t. n ≤ i ‹
(Abs^n) (var-dist vs' ((Abs^n) (var-dist vs (liftn n t 0)))) ↔ strip-context n t 0
  using perm-vars dist-perm-eta by blast

have strip-context n s 0 ↔ (Abs^n) (var-dist vs' ((Abs^n) (var-dist vs (liftn n s 0))))
  using etas [THEN term-sym] not-free(1).
also have ... ↔ (Abs^n) (var-dist vs' ((Abs^n) (var-dist vs (liftn n t 0))))
proof (rule absn-cong, rule var-dist-cong)
  have (Abs^n) (var-dist vs (liftn n s 0)) = (Abs^n) (var-dist vs (liftn n s n))
    using not-free(1) liftn-liftn0[of s n] by simp
  also have ... = (Abs^n) (liftn n (var-dist vs s) n)
    using vs-lt-n liftn-var-dist-lt by simp
  also have ... = liftn n ((Abs^n) (var-dist vs s) 0)
    using liftn-absn by simp
  also have ... ↔ liftn n ((Abs^n) (var-dist vs t) 0)
    using perm-equiv by (rule equiv-liftn)
  also have ... = (Abs^n) (liftn n (var-dist vs t) n)
    using liftn-absn by simp
  also have ... = (Abs^n) (var-dist vs (liftn n t n))
    using vs-lt-n liftn-var-dist-lt by simp
  also have ... = (Abs^n) (var-dist vs (liftn n t 0))
    using not-free(2) liftn-liftn0[of t n] by simp
finally show (Abs^n) (var-dist vs (liftn n s 0)) ↔ ...
  qed
also have ... ↔ strip-context n t 0
  using etas not-free(2).
finally show ?thesis.
  qed

General notion of bracket abstraction for lambda terms
definition foldr-option :: ('a ⇒ 'b ⇒ 'b option) ⇒ 'a list ⇒ 'b ⇒ 'b option
where foldr-option f xs e = foldr (λa b. Option.bind b (f a)) xs (Some e)

lemma bind-eq-SomeE:
  assumes Option.bind x f = Some y
  obtains x' where x = Some x' and f x' = Some y
using assms by (auto iff: bind-eq-Some-conv)

lemma foldr-option-Nil[simp]: foldr-option f [] e = Some e
unfolding foldr-option-def by simp

lemma foldr-option-Cons-SomeE:
  assumes foldr-option f (x#xs) e = Some y
  obtains y' where foldr-option f xs e = Some y' and f x y' = Some y
using assms unfolding foldr-option-def by (auto elim: bind-eq-SomeE)

locale bracket-abstraction =
  fixes term-bracket :: nat ⇒ dB ⇒ dB option

62
assumes bracket-app: \( \text{term-bracket } i \ s = \text{Some } s' \Rightarrow s' \cdot \text{Var } i \leftrightarrow s \)
assumes bracket-frees: \( \text{term-bracket } i \ s = \text{Some } s' \Rightarrow \text{frees } s' = \text{frees } s - \{i\} \)

begin

definition term-brackets :: nat list \( \Rightarrow \) dB \( \Rightarrow \) dB option
where term-brackets = foldr-option term-bracket

lemma term-brackets-Nil[simp]: term-brackets \([]\) \( t = \text{Some } t \)
unfolding term-brackets-def by simp

lemma term-brackets-Cons-SomeE:
  assumes term-brackets \((v\#vs)\) \( t = \text{Some } t' \)
  obtains \( s' \) where term-brackets \( vs \ t = \text{Some } s' \) and term-bracket \( v \ s' = \text{Some } t' \)
  using assms unfolding term-brackets-def by (elim foldr-option-Cons-SomeE)

lemma term-brackets-Cons-SomeI:
  assumes term-brackets \( vs \ t = \text{Some } t' \) and term-bracket \( v \ t' = \text{Some } t'' \)
  shows term-brackets \( (v\#vs) \ t = \text{Some } t'' \)
  using assms unfolding term-brackets-def foldr-option-def by simp

lemma term-brackets-dist:
  assumes term-brackets \( vs \ t = \text{Some } t' \)
  shows \( \text{var-dist } vs \ t' \leftrightarrow t \)
proof -
  from assms have \( \forall t'', \ t' \leftrightarrow t'' \rightarrow \text{var-dist } vs \ t'' \leftrightarrow t \)
  proof (induction vs arbitrary: \( t' \))
    case Nil then show \( \forall t'' \leftrightarrow t \) by (simp add: term-sym)
  next
    case (Cons \( v \#vs \) \( u \) where
      inner: term-brackets \( vs \ t = \text{Some } u \) and
      step: term-bracket \( v \ u = \text{Some } t' \)
    by (auto elim: term-brackets-Cons-SomeE)
    from step have red1: \( t' \cdot \text{Var } v \leftrightarrow u \) by (rule bracket-app)
    show \( \forall t'' \leftrightarrow t \) proof rule+
      fix \( t'' \) assume \( t' \leftrightarrow t'' \)
      with red1 have red: \( t'' \cdot \text{Var } v \leftrightarrow u \)
      using term-sym term-trans by blast
      have \( \text{var-dist } (v \# vs) \ t'' = \text{var-dist } vs \ (t'' \cdot \text{Var } v) \) by simp
      also have ... \( \leftrightarrow t \) using Cons.IH[OF inner] red[symmetric] by blast
      finally show \( \text{var-dist } (v \# vs) \ t'' \leftrightarrow t \).
    qed
  qed
then show \( \forall t'' \leftrightarrow t \) by blast
qed

end
Bracket abstraction for idiomatic terms. We consider idiomatic terms with explicitly assigned variables.

**Lemma** `strip-unlift-vars`:
- **Assumes** opaque `x` = []
- **Shows** `strip-context n (unlift-vars n x) 0 = unlift-vars 0 x`

**Using** `assms by (induction x) (simp-all add: strip-context-liftn[where m=0, simplified])`

**Lemma** `unlift-vars-frees`: ∀ i ∈ frees (unlift-vars n x). i ∈ set (opaque x) ∨ n ≤ i

**By** `(induction x) (auto simp add: free-liftn)`

**Locale** `itrm-abstraction = special-idiom extra-rule for extra-rule :: nat itrm ⇒ - +`

**Fixes** `itrm-bracket :: nat ⇒ nat itrm ⇒ nat itrm option`

**Assumes** `itrm-bracket-ap`:
\[ \text{itrm-bracket } i \ x = \text{Some } x' \Rightarrow x' \odot \text{Opaque } i \simeq^+ x \]

**Assumes** `itrm-bracket-opaque`:
\[ \text{itrm-bracket } i \ x = \text{Some } x' \Rightarrow \text{set } \text{(opaque } x') = \text{set } \text{(opaque } x) - \{i\} \]

**Definition** `itrm-brackets = foldr-option itrm-bracket`

**Lemma** `itrm-brackets-Nil[simp]`: `itrm-brackets [] x = Some x`

**Unfolding** `itrm-brackets-def by simp`

**Lemma** `itrm-brackets-Cons-SomeE`:
- **Assumes** `itrm-brackets (v#vs) x = Some x'`
- **Obtains** `y' where itrm-brackets vs x = Some y' and itrm-bracket v y' = Some x'`

**Using** `assms unfolding itrm-brackets-def by (elim foldr-option-Cons-SomeE)`

**Definition** `opaque-dist = fold (\lambda i y \odot \text{Opaque } i)`

**Lemma** `opaque-dist-cong`: `x \simeq^+ y \Rightarrow \text{opaque-dist } vs x \simeq^+ \text{opaque-dist } vs y`

**Unfolding** `opaque-dist-def by (induction vs arbitrary: x y) (simp-all add: ap-congL)`

**Lemma** `itrm-brackets-dist`:
- **Assumes** `defined`: `itrm-brackets vs x = Some x'`
- **Shows** `opaque-dist vs x' \simeq^+ x`

**Proof**
- **Def** `x'' \equiv x'`
- **Have** `x' \simeq^+ x''` **Unfolding** `x''-def ..`

**With** `defined show opaque-dist vs x'' \simeq^+ x`

**Unfolding** `opaque-dist-def`

**Proof** `(induction vs arbitrary: x' x'')`

- **Case** Nil then show ?case **Unfolding** `itrm-brackets-def by (simp add: itrm-sym)`

- **Next**
  - **Case** `(Cons v vs)`
from Cons.prems(1) obtain y'
where defined': itrm-brackets vs x = Some y'
and itrm-bracket v y' = Some x'
by (rule itrm-brackets-Cons-SomeE)
then have x' ≃ Opaque v ≻ y' by (elim itrm-bracket-ap)
then have x'' ≃ Opaque v ≻ y'
using Cons.prems(2) by (blast intro: itrm-sym itrm-trans)
note this[symmetric]
with defined' have fold (λi y. y ⋄ Opaque i) vs (x'' ⋄ Opaque v) ≻ x
using Cons.IH by blast
then show ?case by simp
qed

lemma itrm-brackets-opaque:
assumes itrm-brackets vs x = Some x'
shows set (opaque x') = set (opaque x) − set vs
using assms proof (induction vs arbitrary: x')
case Nil
then show ?case unfolding itrm-brackets-def by simp
next
case (Cons v vs)
then show ?case
by (auto elim: itrm-brackets-Cons-SomeE dest!: itrm-bracket-opaque)
qed

lemma itrm-brackets-all:
assumes all-opaque: set (opaque x) ⊆ set vs
and defined: itrm-brackets vs x = Some x'
shows opaque x' = []
proof –
from defined have set (opaque x') = set (opaque x) − set vs
by (rule itrm-brackets-opaque)
with all-opaque have set (opaque x') = {} by simp
then show ?thesis by simp
qed

lemma itrm-brackets-all-unlift-vars:
assumes all-opaque: set (opaque x) ⊆ set vs
and defined: itrm-brackets vs x = Some x'
shows x' ≻ Pure (unlift-vars 0 x')
proof (rule equiv-into-ext-equiv)
from assms have opaque x' = [] by (rule itrm-brackets-all)
then show x' ≻ Pure (unlift-vars 0 x') by (rule all-pure-unlift-vars)
qed

end
5.4.7 Lifting with bracket abstraction

locale lifted-bracket = bracket-abstraction + itrm-abstraction +
assumes bracket-compat:
  set (opaque x) ⊆ {0..<n} ⇒ i < n ⇒
  term-bracket i (unlift-vars n x) = map-option (unlift-vars n) (itrm-bracket i x)
begin

lemma brackets-unlift-vars-swap:
assumes all-opaque: set (opaque x) ⊆ {0..<n}
  and vs-bound: set vs ⊆ {0..<n}
  and defined: itrm-brackets vs x = Some x'
shows term-brackets vs (unlift-vars n x) = Some (unlift-vars n x')
using vs-bound defined proof (induction vs arbitrary: x')
case Nil
then show ?case by simp
next
  case (Cons v vs)
  then obtain y' where ivs: itrm-brackets vs x = Some y'
    and iv: itrm-bracket v y' = Some x'
    by (elim itrm-brackets-Cons-SomeE)
  with Cons have term-brackets vs (unlift-vars n x) = Some (unlift-vars n y')
  by auto
moxorever {
    have Some (unlift-vars n x') = map-option (unlift-vars n) (itrm-bracket v y')
      unfolding iv by simp
    moreover have set (opaque y') ⊆ {0..<n}
      using all-opaque ivs by (auto dest: itrm-brackets-opaque)
    moreover have v < n using Cons.prems by simp
    ultimately have term-bracket v (unlift-vars n y') = Some (unlift-vars n x')
      using bracket-compat by auto
  }
ultimately show ?case by (rule term-brackets-ConsI)
qed

theorem bracket-lifting:
assumes all-vars: set (opaque x) ∪ set (opaque y) ⊆ {0..<n}
  and perm-vars: perm-vars n vs
  and defined: itrm-brackets vs x = Some x' itrm-brackets vs y = Some y'
  and base-eq: (Abs `n) (unlift-vars n x) ↔ (Abs `n) (unlift-vars n y)
shows x ≃ y
proof –
from perm-vars have set-vs: set vs = {0..<n}
  unfolding perm-vars-def by simp
have x-swap: term-brackets vs (unlift-vars n x) = Some (unlift-vars n x')
  using all-vars set-vs defined(1) by (auto intro: brackets-unlift-vars-swap)
have y-swap: term-brackets vs (unlift-vars n y) = Some (unlift-vars n y')
using all-vars set-vs defined(2) by (auto intro: brackets-unlift-vars-swap)

from all-vars have set (opaque x) ⊆ set vs unfolding set-vs by simp
then have complete-x: opaque x' = []
  using defined(1) itrm-brackets-all by blast
then have ux-frees: ∀ i ∈ frees (unlift-vars n x'). n ≤ i
  using unlift-vars-frees by fastforce

from all-vars have set (opaque y) ⊆ set vs unfolding set-vs by simp
then have complete-y: opaque y' = []
  using defined(2) itrm-brackets-all by blast
then have uy-frees: ∀ i ∈ frees (unlift-vars n y'). n ≤ i
  using unlift-vars-frees by fastforce

have x ≃⁺ opaque-dist vs x'
  using defined(1) by (rule itrm-brackets-dist[symmetric])
also have ... ≃⁺ opaque-dist vs (Pure (unlift-vars 0 x'))
  using all-vars set-vs defined(1)
by (auto intro: opaque-dist-cong itrm-brackets-all-unlift-vars)
also have ... ≃⁺ opaque-dist vs (Pure (unlift-vars 0 y'))

proof (rule opaque-dist-cong, rule pure-cong)
  have (Abs`n) (var-dist vs (unlift-vars n x')) ↔ (Abs`n) (unlift-vars n x)
    using x-swap term-brackets-dist by auto
  also have ... ↔ (Abs`n) (unlift-vars n y) using base-eq.
  also have ... ↔ (Abs`n) (var-dist vs (unlift-vars n y'))
    using y-swap term-brackets-dist[THEN term-sym] by auto
finally have strip-context n (unlift-vars n x') 0 ↔ strip-context n (unlift-vars n y') 0
  using perm-vars ux-frees uy-frees
by (intro dist-perm-eta-equiv)
then show unlift-vars 0 x' ↔ unlift-vars 0 y'
  using strip-unlift-vars complete-x complete-y by simp
qed
also have ... ≃⁺ opaque-dist vs y' proof (rule opaque-dist-cong)
  show Pure (unlift-vars 0 y') ≃⁺ y'
    using all-vars set-vs defined(2) itrm-brackets-all-unlift-vars[THEN itrm-sym] by blast
  qed
also have ... ≃⁺ y using defined(2) by (rule itrm-brackets-dist)
finally show ?thesis .
qed

end
References


