Applicative Lifting

Andreas Lochbihler       Joshua Schneider

August 16, 2018

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
   1.1 Equality restricted to a set .................................. 3
   1.2 Proof automation ............................................. 3
   1.3 Overloaded applicative operators ............................ 5

2 Common applicative functors
   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 ⇒ bool
where [simp]: eq-on A = (\∀ x y. x ∈ A ∧ x = y)

lemma rel-fun-eq-onI: (\∀ 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 f x y ≡ 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\}),
]
\[(K, \@\{\text{thm K-def}\}),\]
\[(S, \@\{\text{thm S-def}\}),\]
\[(T, \@\{\text{thm T-def}\}),\]
\[(W, \@\{\text{thm W-def}\})]\]

**private attribute-setup** `combinator-eq` =
\[
\text{Scan.lift (Scan.option (Args.$$\$\$\$ \text{weak} |-- Scan.optional (Args.colon |-- Scan.repeat1 Args.name) []) >>}
\text{Applicative.combinator-rule-attrib)}
\]

**lemma** `[combinator-eq]`:
\[
B \equiv S (K S) K \text{ unfolding combinator-unfold}.
\]

**lemma** `[combinator-eq]`:
\[
C \equiv S (S (K (S (K S) K)) S) (K K) \text{ unfolding combinator-unfold}.
\]

**lemma** `[combinator-eq]`:
\[
I \equiv W K \text{ unfolding combinator-unfold}.
\]

**lemma** `[combinator-eq]`:
\[
I \equiv C K () \text{ unfolding combinator-unfold}.
\]

**lemma** `[combinator-eq]`:
\[
S \equiv B (B W) (B B C) \text{ unfolding combinator-unfold}.
\]

**lemma** `[combinator-eq]`:
\[
T \equiv C I \text{ unfolding combinator-unfold}.
\]

**lemma** `[combinator-eq weak: C]`:
\[
C \equiv C (B B (B W (C (B C (B B C (K I)))) (\text{cuncurry (K I)}){(\text{cuncurry K)})))) \text{ cpair unfolding combinator-unfold uncurry-pair}.
\]

```ml
end
```

**method-setup** `applicative-unfold` =
\[
\text{Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>}
\text{SIMPLE-METHOD' (Applicative.unfold-wrapper-tac ctxt opt-af)});\]

unfold into an applicative expression

**method-setup** `applicative-fold` =
\[
\text{Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>}
\text{SIMPLE-METHOD' (Applicative.fold-wrapper-tac ctxt opt-af)});\]

fold an applicative expression

**method-setup** `applicative-nf` =
\[
\text{Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>}
\text{SIMPLE-METHOD' (Applicative.normalize-wrapper-tac ctxt opt-af)});\]

prove an equation that has been lifted to an applicative functor, using normal forms

**method-setup** `applicative-lifting` =
\[
\text{Applicative.parse-opt-afun >> (fn opt-af => fn ctxt =>}
\text{SIMPLE-METHOD' (Applicative.lifting-wrapper-tac ctxt opt-af)});\]

prove an equation that has been lifted to an applicative functor

```ml
\langle \text{Outer-Syntax.local-theory-to-proof @{command-keyword applicative} \rangle}
\```
register applicative functors

\(\text{Parse.binding } \leftarrow \)
\[
\text{Scan.optional } \leftarrow \{\text{keyword } \}
\text{parse.list Parse.short-ident } \leftarrow \{\text{keyword }\}
\]
\(\}	ext{[] } \leftarrow \)
\[
\leftarrow \{\text{keyword for }\}
\text{parse.reserved pure } \leftarrow \{\text{keyword :}\}
\text{parse.term } \leftarrow \)
\[
\text{parse.reserved ap } \leftarrow \{\text{keyword :}\}
\text{parse.term } \leftarrow \)
\[
\text{Scan.option } \leftarrow \{\text{keyword :}\}
\text{parse.reserved rel } \leftarrow \{\text{keyword :}\}
\text{parse.term } \leftarrow \)
\[
\text{Scan.option } \leftarrow \{\text{keyword :}\}
\text{parse.reserved set } \leftarrow \{\text{keyword :}\}
\text{parse.term } \leftarrow \)
\]
\(\Rightarrow \text{Applicative.applicative-cmd})\)

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

attribute-setup applicative-unfold =
\(\text{(Scan.lift } \leftarrow \{\text{Scan.option Parse.name } \leftarrow \{\text{Applicative.add-unfold-attrib}\})\)
\(\text{register rules for unfolding into applicative expressions}\)

attribute-setup applicative-lifted =
\(\text{(Scan.lift } \leftarrow \{\text{Parse.name } \leftarrow \{\text{Applicative.forward-lift-attrib}\})\)
\(\text{lift an equation to an applicative functor}\)

1.3 Overloaded applicative operators

consts
\[\text{pure } :: \{\text{a} \Rightarrow \{\text{b}\}\}\]
\[\text{ap } :: \{\text{a} \Rightarrow \{\text{b} \Rightarrow \{\text{c}\}\}\}\]

bundle applicative-syntax
begin
notation \(\text{ap } \{\text{infixl} \circ 70\}\)
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 \(\text{const } x = (\lambda \cdot \text{x})\)
definition \(\text{apf } x y = (\lambda \cdot \text{x z } (\text{y z}))\)

5
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.

\textbf{applicative env (K, W)}

for
\begin{itemize}
  \item \texttt{pure: const}
  \item \texttt{ap: apf}
  \item \texttt{rel: rel-fun (=)}
  \item \texttt{set: range}
\end{itemize}

\texttt{by(simp-all add: const-def apf-def rel-fun-def)}

\textbf{lemma}
\texttt{includes applicative-syntax}
\texttt{shows constant (λf x y. f y x) \circ f \circ x \circ y = f \circ y \circ x}
\texttt{by applicative-lifting simp}

\section{2.2 Option}

\textbf{theory Applicative-Option imports}
\texttt{Applicative HOL-\textit{Library}, Adhoc-Overloading}
\texttt{begin}

\texttt{fun ap-option :: ('a ⇒ 'b) option ⇒ 'a option ⇒ 'b option}
\texttt{where}
\begin{itemize}
  \item \texttt{ap-option (Some f) (Some x) = Some (f x)}
  \item \texttt{ap-option - - = None}
\end{itemize}

\texttt{abbreviation (input) pure-option :: 'a ⇒ 'a option}
\texttt{where}
\begin{itemize}
  \item \texttt{pure-option ≡ Some}
\end{itemize}

\texttt{adhoc-overloading Applicative.pure pure-option}
\texttt{adhoc-overloading Applicative.ap ap-option}

\texttt{lemma some-ap-option: ap-option (Some f) x = map-option f x}
\texttt{by (cases x) simp-all}

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

\texttt{lemma ap-option-transfer[transfer-rule]:}
\begin{itemize}
  \item \texttt{rel-fun (rel-option (rel-fun A B)) (rel-fun (rel-option A) (rel-option B)) ap-option}
\end{itemize}
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 (λy 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 y x) ◦ 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 (eq-on (set-option x)) x x by (auto intro: op-
        tion.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) ⇒ ('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 (o)) 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
The document contains a proof of properties for a type constructor `either` with respect to an applicative structure. It includes definitions, proofs, and code snippets for generating and using the applicative instance for `either`. The proof involves showing properties for `pure` and the composition of functions when applying to invariants of `either`. The code excerpts demonstrate how to interpret and generate these properties using Isabelle/HOL.
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 x. 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
  case (Cons f gs)
  have [?B g] ⊗ (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
fix \( R :: \{a ightarrow b \} \rightarrow \text{bool} \)

show \( \text{rel-fun} \ R \ (\text{list-all2} \ R) \ (\lambda x. [x]) \ (\lambda x. [x]) \) by transfer-prover

fix \( R \) and \( f :: \{a \rightarrow b\} \) list and \( g :: \{a \rightarrow c\} \) list and \( x \)

assume \( \text{transfer-rule}: \text{list-all2} \ (\text{rel-fun} \ (\text{eq-on} \ (\text{set} \ x)) \) \( R \) \( f \) \( g \)

have \( \text{transfer-rule}: \text{list-all2} \ (\text{eq-on} \ (\text{set} \ x)) \) \( x \) \( x \) by \( \text{simp add: list-all2-same} \)

show \( \text{list-all2} \ R \ (f \circ x) \ (g \circ x) \) by transfer-prover

qed \( \text{simp add: cons-ap-list} \)

lemma map-ap-conv [applicative-unfold]: \( \text{map} \ f \ x = \{f\} \circ x \)

unfolding \( \text{ap-list-def} \) \( \text{List.bind-def} \)

by \( \text{simp} \)

end

end

3 Distinct, non-empty list

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

begin

lemma bind-eq-Nil-iff [simp]: \( \text{List.bind} \) \( xs \) \( f \) = \( [] \) \( \iff \) \( (\forall x \in \text{set} \ xs. \ f x = []) \)

by \( \text{simp add: List.bind-def} \)

lemma zip-eq-Nil-iff [simp]: \( \text{zip} \) \( xs \) \( ys \) \( = \) \( [] \) \( \iff \) \( xs = [] \) \( \lor \) \( ys = [] \)

by \( \text{(cases \( xs \) \( ys \) \( rule: \) \( \text{list.exhaust} \) \( \text{[case-product \( \text{list.exhaust} \])} \) \( \text{simp-all} \) \)

lemma remdups-append1: \( \text{remdups} \) \( \text{remdups} \) \( xs \) \( @ \) \( ys \) \( = \) \( \text{remdups} \) \( xs \) \( @ \) \( ys \)

by \( \text{induction \( xs \)} \) \( \text{simp-all} \)

lemma remdups-append2: \( \text{remdups} \) \( xs \) \( @ \) \( \text{remdups} \) \( ys \) \( = \) \( \text{remdups} \) \( xs \) \( @ \) \( ys \)

by \( \text{induction \( xs \)} \) \( \text{simp-all} \)

lemma remdups-append1-drop: \( \text{set} \) \( xs \) \( \subseteq \) \( \text{set} \) \( ys \) \( \Longrightarrow \) \( \text{remdups} \) \( xs \) \( @ \) \( ys \) \( = \) \( \text{remdups} \) \( ys \)

by \( \text{induction \( xs \)} \) \( \text{auto} \)

lemma remdups-concat-map: \( \text{remdups} \) \( \text{concat} \) \( \text{map} \) \( \text{remdups} \) \( \text{xss} \) \( = \) \( \text{remdups} \) \( \text{concat} \) \( \text{xss} \)

by \( \text{induction \( xss \)} \) \( \text{(simp-all add: remdups-append1, metis remdups-append2)} \)

lemma remdups-concat-remdups: \( \text{remdups} \) \( \text{concat} \) \( \text{remdups} \) \( \text{xss} \) \( = \) \( \text{remdups} \) \( \text{concat} \) \( \text{xss} \)

apply \( \text{induction \( xss \)} \)

apply \( \text{(auto simp add: remdups-append1-drop)} \)
apply (subst remdups-append1-drop; auto)
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 (=) (rel-fun (pcr-dnelist (=)) (pcr-dnelist (=))) (λf xs. remdups (map f xs)) Applicative-DNELList.map
by (simp add: map-def rel-fun-def dnelist.per-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
end

lift-definition ap-dnelist :: ('a ⇒ 'b) dnelist ⇒ 'a dnelist ⇒ 'b dnelist
is λf x. remdups (ap-list f x)
by(auto simp add: ap-list-def)

adhoc-overloading Applicative.ap ap-dnelist

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

context includes applicative-syntax
begin

lemma ap-pure-dnelist:
  pure-dnelist f ⋄ x = Applicative-DNEList.map f x
by transfer simp

applicative dnelist (K)
for pure: pure-dnelist
  ap: ap-dnelist

proof −
  show pure-dnelist (λx. x) ⋄ x = x for x :: 'a dnelist
    by transfer simp

  have *: remdups (remdups (remdups ([λg f x. g (f x)] ⋄ g) ⋄ f) ⋄ x) = remdups
    (g ⋄ remdups (f ⋄ x))
      (is ?lhs = ?rhs for g :: ('b ⇒ 'c) list and f :: ('a ⇒ 'b) list and x
        proof −
          have ?lhs = remdups (concat (map (λf. map f x) (remdups (concat (map (λx. map (λf y. x (f y)) f)) g))))
            unfolding ap-list-def List.bind-def
            by(subst (2) remdups-concat-remdups[symmetric])(simp add: o-def remdups-map-remdups
              remdups-concat-remdups)
          also have ... = remdups (concat (map (λf. map f x) (concat (map (λx. map (λf y. x (f y)) f)) g))))
            by(subst (1) remdups-concat-remdups[symmetric])(simp add: remdups-map-remdups
              remdups-concat-remdups)
          also have ... = remdups (concat (map remdups (map (λg. map g (concat
              (map (λf. map f x)) f)))) g))
            using list.pure-B-conv[of g f x] unfolding remdups-concat-map
            by(simp add: ap-list-def List.bind-def o-def)
          also have ... = ?rhs unfolding ap-list-def List.bind-def
            by(subst (2) remdups-concat-map[symmetric])(simp add: o-def remdups-map-remdups)
          finally show ?thesis .
        qed
      show pure-dnelist (λg f x. g (f x)) ⋄ g ⋄ f ⋄ x = g ⋄ (f ⋄ x)
        for g :: ('b ⇒ 'c) dnelist and f :: ('a ⇒ 'b) dnelist and x
        by transfer(rule *)

14
show pure-dnelist f ⊙ pure-dnelist x = pure-dnelist (f x) for f :: 'a ⇒ 'b and x
by transfer simp
show f ⊙ pure-dnelist x = pure-dnelist (λf. f) ⊙ f for f :: ('a ⇒ 'b) dnelist
and x
by transfer(simp add: list.interchange)

have *: remdups (remdups ([λ x y. x] ⊙ x) ⊙ 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 ([λ x y. x] ⊙ x) ⊙ 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) ⊙ x ⊙ 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 y. f y x) ⊙ pure-dnelist ((* )) ⊙ x ⊙ y ≠
pure-dnelist ((* )) ⊙ y ⊙ 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
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 x by(cases g 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 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 (+) △ (x :: (nat, int) monoid-ap) ○ y = pure (+) ○ y ○ x  
by(applicative-lifting comm-monoid-add) simp  
end

3.2 Filters

theory Applicative-Filter imports  
Complex-Main  
Applicative  
HOL−Library. Conditional-Parametricity  
begin

definition pure-filter :: ′a ⇒ ′a filter where  
pure-filter x = principal {x}

definition ap-filter :: (′a ⇒ ′b) filter ⇒ ′a filter ⇒ ′b filter where  
ap-filter F X = filtermap (λ(f, x). f x) (prod-filter F X)

lemma eq-on-UNIV: eq-on UNIV = (=)
  by auto

declare filtermap-parametric[transfer-rule]

parametric-constant pure-filter-parametric[transfer-rule]: pure-filter-def
parametric-constant ap-filter-parametric [transfer-rule]: ap-filter-def

applicative filter (C)  
— K is available for not-bot filters and W isholds not available
for
  pure: pure-filter
  ap: ap-filter
  rel: rel-filter

proof —
show ap-filter (ap-filter f) (pure-filter x) = pure-filter (f x) for f :: ′a ⇒ ′b
and x
  by(simp add: ap-filter-def pure-filter-def principal-prod-principal)
show ap-filter (ap-filter (ap-filter (pure-filter (λg f x. g ((f x))))) g) f) x =
ap-filter g (ap-filter f x) for f :: ′a ⇒ ′b filter and g :: ′b ⇒ ′c filter and x
show ap-filter (pure-filter (λx. x)) x = x for x :: ′a filter
  by(simp add: ap-filter-def pure-filter-def prod-filter-principal-singleton filtermap-filtermap)
show ap-filter (ap-filter (ap-filter (pure-filter (λf x y. f y x)) f) x) y =
ap-filter (ap-filter f y) x for f :: ′b ⇒ ′a ⇒ ′c filter and x y

17
apply(subst (2) prod-filter-commute)
apply(simp add: filtermap-filtermap prod-filtermap1 prod-filtermap2)
done
show rel-fun R (rel-filter R) pure-filter pure-filter for R :: 'a ⇒ 'b ⇒ bool
  by(rule pure-filter-parametric)
show rel-filter R (ap-filter f x) (ap-filter g x) if rel-filter (rel-fun (eq-on UNIV) R) f g
  for R and f :: ('a ⇒ 'b) filter and g :: ('a ⇒ 'c) filter and x
  supply that\[unfolded eq-on-UNIV, transfer-rule\] by transfer-prover
qed

end

3.3 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

3.4 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
by (coinduction arbitrary: f) auto

lemma ap-stream-composition: pure (λg f x. g (f x)) ⊙ g ⊙ f ⊙ x = g ⊙ (f ⊙ 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 ⇒ 'a ⇒ 'c) stream and f x
  show pure (λg f x. g x (f x)) ⊙ g ⊙ f ⊙ x = g ⊙ x ⊙ (f ⊙ x)
  by (coinduction arbitrary: g f x) auto
next
  fix x :: 'b stream and y :: 'a stream
  show pure (λx y. x) ⊙ x ⊙ y = x
  by (coinduction arbitrary: x y) auto
next
  fix R :: 'a ⇒ 'b ⇒ bool
  show (R ===> 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 ⇒ 'b) stream and g :: ('a ⇒ 'c) stream and x
  assume [transfer-rule]: stream-all2 (eq-on (sset x)) x x by (simp add: stream.rel-refl-strong)
  have [transfer-rule]: stream-all2 (eq-on (sset x)) x x by (simp add: stream.rel-refl-strong)
  show stream-all2 (f ⊙ x) (g ⊙ x) by transfer-prover
qed (rule ap-stream-homo)

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

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

end
3.5 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

3.6 Probability mass functions

theory Applicative-PMF imports Applicative HOL−Probability.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)

end
lemma ap-pmf-comp: pure-pmf (⊙) ∘ 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-pair-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 ∘ pure-pmf x = pure-pmf (λf. f x) ∘ u
by(simp add: ap-pmf-def pair-return-pmf1 pair-return-pmf2 pmf.map-comp o-def)

lemma ap-pmf-K: return-pmf (λx. x) ∘ x ∘ 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: return-pmf (λf x y. f y x) ∘ f ∘ x ∘ y = f ∘ y ∘ 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-pair-pmf
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(rule ap-pmf-comp[unfolded o-def[abs-def]] ap-pmf-homo ap-pmf-C ap-pmf-K)+

end

end
3.7 Probability mass functions implemented as lists with duplicates

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

lemma sum-list-concat-map: sum-list (concat (map f xs)) = sum-list (map (λx. sum-list (f x)) xs)
by(induction xs) simp-all

context includes applicative-syntax begin

lemma set-ap-list simp:[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, I)] ∈ ?pfp for x by simp
  qed

setup-lifting type-definition-pfp

lift-definition pure-pfp :: 'a ⇒ 'a pfp is λx. [(x, I)] 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)] ⋄ fs ⋄ 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)] ⋄ xs ⋄ 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 (\(\lambda x. x\)) \(\circ\) \(x = x\) for \(x :: 'a pfp\)
by transfer(simp add: ap-list-def List.bind-def)

show pure-pfp \(f \circ\) pure-pfp \(x = pure-pfp (f x)\) for \(f :: 'a \Rightarrow 'b\) and \(x\)
by transfer(applicative-lifting; simp)

show pure-pfp (\(\lambda g f x. g (f x)\)) \(\circ\) \(g \circ f \circ x = g \circ (f \circ x)\)
for \(g :: ('b \Rightarrow 'c) pfp\) and \(f :: ('a \Rightarrow 'b) pfp\) and \(x\)
by transfer(applicative-lifting; clarsimp)

show \(f \circ pure-pfp x = pure-pfp (\lambda f x. f x) \circ f\) for \(f :: ('a \Rightarrow 'b) pfp\) and \(x\)
by transfer(applicative-lifting; clarsimp)

qed

end

end

3.8 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 \(f \star\) 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-Analytic.Finite-Cartesian-Product
  HOL-Library.Adhoc-Overloading
begin

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

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

adhoc-overloading Applicative.ap ap-vec

applicative vec \((K, W)\)

23
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(simp add: ap-vec-def)
end

theory Applicative-Functor imports
  Applicative-Environment
  Applicative-Option
  Applicative-Sum
  Applicative-Set
  Applicative-List
  Applicative-DNEList
  Applicative-Monoid
  Applicative-Filter
  Applicative-State
  Applicative-Stream
  Applicative-Open-State
  Applicative-PMF
  Applicative-Probability-List
  Applicative-Star
  Applicative-Vector
begin
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 point-wise operations for the algebraic type classes
context includes applicative-syntax
begin

24
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]: $f \times g = \text{pure} \circ (\ast) \circ f \circ g$

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

4.2 Pointwise arithmetic on streams

theory Stream-Algebra

imports Applicative-Stream

begin

instantiation stream :: (zero) zero begin

definition [applicative-unfold]: $0 = \text{sc} \ 0$

instance ..

end

instantiation stream :: (one) one begin

definition [applicative-unfold]: $1 = \text{sc} \ 1$

instance ..

end

instantiation stream :: (plus) plus begin

context includes applicative-syntax begin

definition [applicative-unfold]: $x + y = \text{pure} \circ (+) \circ x \circ (y :: 'a \ stream)$

end

instance ..

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

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

instantiation stream :: (times) times begin
category includes applicative-syntax begin
definition [applicative-unfold]: x * y = pure (\( \ast \)) \circ x \circ (y :: 'a \ stream)
end
instance ..
end

instance stream :: (Rings.dvd) Rings.dvd ..

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

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

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

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

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

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

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

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

instance stream :: (comm-monoid-mult) comm-monoid-mult
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

instance stream :: (cancel-ab-semigroup-add) cancel-ab-semigroup-add
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

27
by intro-classes (applicative-lifting, simp add: ring-distrib) +

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
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 ..
end

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 correct-
ness 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] sug-
gest 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.

29
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) (+)

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 (λl. 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 (#) ⋄ fresh ⋄ label-list xs

lemma label-append: label-list (a @ b) = pure (@) ⋄ label-list a ⋄ label-list b
— The proof lifts the defining equations of (@) to the state monad.
proof (induction a)
case Nil
show ?case

30
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) ⇒ ′af1) ⇒ (′af1 ⇒ ′af3 ⇒ ′af13) ⇒ (′af13 ⇒ ′af2 ⇒ ′af) ⇒ ′af2 dual ⇒ ′af3 dual ⇒ ′af dual
  is λpure ap1 ap2 f x. ap2 (ap1 (pure (λx f. 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 \( \text{ap-state-rev} :: ('s, 'a \Rightarrow 'b) \text{state-rev} \Rightarrow ('s, 'a) \text{state-rev} \Rightarrow ('s, 'b) \text{state-rev} \)
where \( \text{ap-state-rev} = \text{ap-dual State-Monad}.\text{return State-Monad}.\text{ap State-Monad}.\text{ap} \)

adhoc-overloading \text{Applicative}.\text{pure pure-state-rev}
adhoc-overloading \text{Applicative}.\text{ap ap-state-rev}

applicative \text{state-rev}
for
\( \text{pure}: \text{pure-state-rev} \)
\( \text{ap}: \text{ap-state-rev} \)
unfolding \text{pure-state-rev-def ap-state-rev-def} by(transfer, applicative-nf, rule refl)+

type-synonym \((s, 'a) \text{state-rev-rev} = (s, 'a) \text{state-rev dual} \)

definition \text{pure-state-rev-rev} :: 'a \Rightarrow (s, 'a) \text{state-rev-rev}
where \text{pure-state-rev-rev} = \text{pure-dual pure-state-rev ap-state-rev ap-state-rev ap-state-rev}

adhoc-overloading \text{Applicative}.\text{pure pure-state-rev-rev}
adhoc-overloading \text{Applicative}.\text{ap ap-state-rev-rev}

applicative \text{state-rev-rev}
for
\( \text{pure}: \text{pure-state-rev-rev} \)
\( \text{ap}: \text{ap-state-rev-rev} \)
unfolding \text{pure-state-rev-rev-def ap-state-rev-rev-def} by(transfer, applicative-nf, rule refl)+

lemma \text{ap-state-rev-B}: \text{B} f \circ B x = B (\text{State-Monad}.\text{return} (\lambda x f f x) \circ x \circ f)
unfolding \text{ap-state-rev-def} by(fact ap-dual.abs-eq)

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

lemma \text{ap-state-rev-rev-B}: \text{B} f \circ B x = B (\text{pure-state-rev} (\lambda x f f x) \circ x \circ f)
unfolding \text{ap-state-rev-rev-def} by(fact ap-dual.abs-eq)

lemma \text{ap-state-rev-rev-pure-B}: \text{pure} f \circ B x = B (\text{pure-state-rev} f \circ x)
unfolding \text{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 ((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
*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 (snd s)) (λs. State-Monad.return (shd s)))

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

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

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

B ◦ B is an idiom morphism

lemma B-pure: pure x = B (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) ◦ B (B x) = B (B (f ◦ x))
proof
  have B (B f) ◦ B (B x) = B (B (pure (λx f. f x) ◦ f ◦ (pure (λx f. f x) ◦ x ◦ pure (λx f. f x)))))
    (is - = B (B ?exp))
    unfolding ap-state-rev-rev-B B-pure ap-state-rev-B ..
  also have ?exp = f ◦ 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 ◦ f x
| traverse-rev-rev f (Node l r) = pure Node ◦ traverse-rev-rev f l ◦ 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 ◦ traverse-rev-rev (B ◦ f)

lemma traverse-B-B: traverse-rev-rev (B ◦ B ◦ f) = B ◦ B ◦ 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 ◦ recurse-rev (B ◦ f) (is ?lhs = ?rhs)
proof
  have ?lhs = un-B ◦ un-B ◦ B ◦ B ◦ traverse f by (simp add: o-def B-inverse)
also have \( \ldots = \text{un-B} \circ \text{un-B} \circ \text{traverse-rev-rev} \ (B \circ B \circ f) \) unfolding \text{traverse-B-B} by(simp add: o-assoc)
also have \( \ldots = ?\text{rhs} \) by(simp add: recurse-rev-def o-assoc)
finally show \( ?\text{thesis} \).
qed

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

Apply traversals to labelling

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

\begin{enumerate}
\item \textbf{definition} adorn :: \( \alpha \Rightarrow (\beta \text{ stream}, \alpha \times \beta) \text{ state} \)
\item \textbf{where} adorn \( a = \text{pure } (\text{Pair } a) \circ \text{fetch} \)
\end{enumerate}

\begin{enumerate}
\item \textbf{abbreviation} label :: \( \alpha \text{ tree } \Rightarrow (\beta \text{ stream}, (\alpha \times \beta) \text{ tree}) \text{ state} \)
\item \textbf{where} label \( \equiv \text{traverse } \text{adorn} \)
\end{enumerate}

\begin{enumerate}
\item \textbf{abbreviation} unlabel :: \( (\alpha \times \beta) \text{ tree } \Rightarrow (\beta \text{ stream}, \alpha \text{ tree}) \text{ state} \)
\item \textbf{where} unlabel \( \equiv \text{recurse } \text{strip} \)
\end{enumerate}

\begin{enumerate}
\item \textbf{lemma} strip-adorn: \( \text{strip} \cdot \text{adorn} = \text{pure} \)
\item by(simp add: strip-def adorn-def fun-eq-iff fetch-def[abs-def] ap-conv-bind-state)
\end{enumerate}

\begin{enumerate}
\item \textbf{lemma} correctness-monadic: \( \text{unlabel} \cdot \text{label} = \text{pure} \)
\end{enumerate}
by (rule recurse-traverse)(rule strip-adorn)

4.3.3 Applicative correctness statement

Repeating an effect

primrec repeatM :: nat ⇒ ('s, 'x) state ⇒ ('s, 'x list) state
where
  repeatM 0 f = State-Monad.return []
| repeatM (Suc n) f = pure (#) ◦ f ◦ 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 ◦ 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, 'x 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

36
shows \( \text{State-Monad.return\ dlabels} \circ \text{label-tree\ t} = \text{symbols\ (leaves\ t)} \)

proof (induction \( t \))

show \( \text{pure\ dlabels} \circ \text{label-tree\ (Leaf\ x)} = \text{symbols\ (leaves\ (Leaf\ x))} \) for \( x :: 'a \)

unfolding \( \text{label-tree-simps\ leaves-simps\ repeatM\ simps\ by\ applicative-nf\ simps} \)

next

fix \( l\ r :: 'a\ tree \)

assume \( \text{IH: pure\ dlabels} \circ \text{label-tree\ l} = \text{symbols\ (leaves\ l)} \) pure \( \text{dlabels} \circ \text{label-tree\ r} = \text{symbols\ (leaves\ r)} \)

let \( ?\text{cat} = \text{case-prod\ append\ and}\ ?\text{disj} = \text{case-prod\ disjoint} \)

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

unfolding \( \text{label-tree-simps\ by\ applicative-nf\ simps} \)

also have \( \ldots = \text{pure\ ?f\ (pure\ dlabels\ \circ\ label-tree\ l)} \circ (\text{pure\ dlabels\ \circ\ label-tree\ r}) \)

unfolding \( \text{IH\ distinct\ ..} \)

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

unfolding \( \text{leaves-simps\ repeatM-plus\ by\ applicative-nf\ simps} \)

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

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

qed

end

4.3.4 Probabilistic tree relabelling

primrec \( \text{mirror} :: 'a\ tree \Rightarrow 'a\ tree \)

where

\( \text{mirror\ (Leaf\ x)} = \text{Leaf\ x} \)

| \( \text{mirror\ (Node\ l\ r)} = \text{Node\ (mirror\ r)\ (mirror\ l)} \)

datatype \( \text{dir} = \text{Left} | \text{Right} \)

hide-const (open) \( \text{path} \)

function (sequential) \( \text{subtree} :: \text{dir\ list} \Rightarrow 'a\ tree \Rightarrow 'a\ tree \)

where

\( \text{subtree\ (Left\ #\ path)}\ \text{Node\ l\ r} = \text{subtree\ path\ l} \)

| \( \text{subtree\ (Right\ #\ path)}\ \text{Node\ l\ r} = \text{subtree\ path\ r} \)

| \( \text{subtree}\ []\ \text{Leaf\ x} = \text{Leaf\ x} \)

| \( \text{subtree}\ []\ t = t \)

by pat-completeness auto

termination by lexicographic-order

adhoc-overloading \( \text{Applicative.pure\ pure-pmf} \)

context fixes \( p :: 'a \Rightarrow 'b\ \text{pmf} \) begin

primrec \( \text{plabel} :: 'a\ tree \Rightarrow 'b\ tree\ \text{pmf} \)

end
where
\[
\begin{align*}
\text{plabel } (\text{Leaf } x) &= \text{pure Leaf } \circ p \ xu \\
| \text{plabel } (\text{Node } t \ r) &= \text{pure Node } \circ \text{plabel } t \circ \text{plabel } r
\end{align*}
\]

lemma plabel-mirror: \( \text{plabel } (\text{mirror } t) = \text{pure mirror } \circ \text{plabel } t \)

proof (induction \( t \))
  case (Leaf \( x \))
  \begin{center}
  show \( ?\text{case unfolding plabel.simps mirror.simps by(applicative-lifting) simp} \)
  \end{center}

next
  case (Node \( t1 \ t2 \))
  \begin{center}
  show \( ?\text{case unfolding plabel.simps mirror.simps Node.IH by(applicative-lifting) simp} \)
  \end{center}

dqed

lemma plabel-subtree: \( \text{plabel } (\text{subtree } \text{path } t) = \text{pure } (\text{subtree } \text{path}) \circ \text{plabel } t \)

proof (induction \( \text{path } t \) rule: subtree.induct)
  case Left: \( (1 \text{ path } l \ r) \)
  \begin{center}
  show \( ?\text{case unfolding plabel.simps subtree.simps Left.IH by(applicative-lifting) simp} \)
  \end{center}

next
  case Right: \( (2 \text{ path } l \ r) \)
  \begin{center}
  show \( ?\text{case unfolding plabel.simps subtree.simps Right.IH by(applicative-lifting) simp} \)
  \end{center}

next
  case (3 \( uu \ xu \))
  \begin{center}
  show \( ?\text{case unfolding plabel.simps subtree.simps by(applicative-lifting) simp} \)
  \end{center}

next
  case (4 \( v \ va \))
  \begin{center}
  show \( ?\text{case unfolding plabel.simps subtree.simps by(applicative-lifting) simp} \)
  \end{center}
dqed

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
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) ∈ R ⇒ (y, z) ∈ R ⇒ (x, y) ∈ joinable R

**Unfolding** joinable-simp by blast

**Lemma** joinableD: (x, y) ∈ joinable R ⇒ ∃ z. (x, z) ∈ R ∧ (y, z) ∈ R

**Unfolding** joinable-simp.

**Lemma** joinableE:
- assumes (x, y) ∈ joinable R
- obtains z where (x, z) ∈ R ∧ (y, z) ∈ R

**Using** assms unfolding joinable-simp by blast

**Lemma** refl-on-joinable: refl-on {x. ∃ y. (x, y) ∈ R} (joinable R)

**By** (auto intro!: refl-onI simp only: joinable-simp)

**Lemma** refl-joinable-iff: (∀ x. ∃ y. (x, y) ∈ R) = refl (joinable R)

**By** (auto intro!: refl-onI dest: refl-onD simp add: joinable-simp)

**Lemma** refl-joinable: refl R ⇒ refl (joinable R)

**Using** refl-joinable-iff by (blast dest: refl-onD)

**Lemma** joinable-refl: refl R ⇒ (x, x) ∈ joinable R

**Using** refl-joinable by (blast dest: refl-onD)

**Lemma** sym-joinable: sym (joinable R)

**By** (auto intro!: symI simp only: joinable-simp)

**Lemma** joinable-sym: (x, y) ∈ joinable R ⇒ (y, x) ∈ joinable R

**Using** sym-joinable by (rule symD)

**Lemma** joinable-mono: R ⊆ S ⇒ joinable R ⊆ joinable S

**By** (rule subrelI) (auto simp only: joinable-simp)

**Lemma** refl-le-joinable:
- assumes refl R
- shows R ⊆ joinable R

**Proof** (rule subrelI)
- fix x y
- assume (x, y) ∈ R
moreover from \(\text{refl } R\): 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: \(\forall x \ y. \ (x, y) \in R \Rightarrow (P \ x, P \ y) \in R\)
assumes joinable: \((x, y) \in \text{joinable } R\)
shows \((P \ x, P \ y) \in \text{joinable } R\)
proof –
from joinable obtain \(z\) where \(xz: (x, z) \in R\) and \(yz: (y, z) \in R\) by (rule joinableE)
from \(R\)-subst \(xz\) have \((P \ x, P \ z) \in R\).
moreover from \(R\)-subst \(yz\) have \((P \ y, P \ z) \in R\).
ultimately show ?thesis by (rule joinableI)
qed

5.1.2 Confluence

definition confluent :: ‘a rel ⇒ bool
where confluent \(R\) ←→ \((\forall x \ y \ y'. \ (x, y) \in R \land (x, y') \in R \Rightarrow (y, y') \in \text{joinable } R)\)

lemma confluentI:
\(\forall x \ y \ y'. \ (x, y) \in R \Rightarrow (x, y') \in R \Rightarrow \exists z. \ (y, z) \in R \land (y', z) \in R \Rightarrow \text{confluent } R\)
unfolding confluent-def by (blast intro: joinableI)

lemma confluentD:
confluent \(R\) ⇒ \((x, y) \in R \Rightarrow (x, y') \in R \Rightarrow (y, y') \in \text{joinable } R\)
unfolding confluent-def by blast

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

lemma trans-joinable:
assumes trans \(R\) and confluent \(R\)
shows trans \((\text{joinable } R)\)
proof (rule transI)
fix \(x \ y \ z\)
assume \((x, y) \in \text{joinable } R\)
then obtain \(u\) where \(xu: (x, u) \in R\) and \(yu: (y, u) \in R\) by (rule joinableE)
assume \((y, z) \in \text{joinable } R\)
then obtain \(v\) where \(yw: (y, v) \in R\) and \(zv: (z, v) \in R\) by (rule joinableE)
from \(yu \ yw\) /confluent \(R\): obtain \(w\) where \(uw: (u, w) \in R\) and \(vw: (v, w) \in R\)
by (blast elim: confluentE)
from \(xu \ uw\) /trans \(R\): have \((x, w) \in R\) by (blast elim: transE)
moreover from \(zw \ vw\) /trans \(R\): have \((z, w) \in R\) by (blast elim: transE)
ultimately show \((x, z) \in \text{joinable } R\) by (rule joinableE)

definition joinablep :: \('a \Rightarrow 'b \Rightarrow \text{bool}') \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}

where joinablep P x y \iff (\exists z. P x z \land P y z)

lemma joinablep-joinable[pred-set-cone]:
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 joinablep-refl: reflp P \implies joinablep P x x
using reflp-joinablep by (rule reflpD)

lemma reflp-le-joinablep: reflp P \implies P \leq joinablep P

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 \leq n + k \land k \leq i \implies (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 \land free t i \lor k + n \leq i \land free t (i - n))
by (induction t arbitrary: k i) (auto simp add: Suc-diff-le)

5.2.2 Reduction

abbreviation beta-eta :: dB \Rightarrow dB \Rightarrow bool (infixl \rightarrow_{\beta\eta} 50)
where beta-eta \equiv sup beta eta

abbreviation beta-eta-reds :: dB \Rightarrow dB \Rightarrow bool (infixl \rightarrow_{\beta\eta}^* 50)
where s \rightarrow_{\beta\eta}^* t \equiv (beta-eta)^* s t

lemma beta-into-beta-eta-reds: s \rightarrow_{\beta} t \implies s \rightarrow_{\beta\eta}^* t
by auto

lemma eta-into-beta-eta-reds: s \rightarrow_{\eta} t \implies s \rightarrow_{\beta\eta}^* t
by auto

lemma beta-reds-into-beta-eta-reds: s \rightarrow_{\beta}^* t \implies s \rightarrow_{\beta\eta}^* t
by (auto intro: rtranclp-mono[THEN predicate2D])
lemma eta-reds-into-beta-eta-reds: \( s \eta \rightarrow t \Rightarrow s \beta \eta \rightarrow t \)
by (auto intro: rtranclp-mono[THEN predicate2D])

lemma beta-eta-appL[intro]: \( s \beta \eta \rightarrow s \rightarrow \beta \eta \ast t \Rightarrow s \rightarrow \beta \eta \ast s' \rightarrow t \)
by (induction set: rtranclp) (auto intro: rtranclp.rtrancl-into-rtrancl)

lemma beta-eta-appR[intro]: \( t \beta \eta \rightarrow t \rightarrow \beta \eta \ast s' \Rightarrow s \rightarrow \beta \eta \ast s' \rightarrow t \)
by (induction set: rtranclp) (auto intro: rtranclp.rtrancl-into-rtrancl)

lemma beta-eta-abs[intro]: \( t \beta \eta \rightarrow Abs t \rightarrow \beta \eta \ast Abs t' \)
by (induction set: rtranclp) (auto intro: rtranclp.rtrancl-into-rtrancl)

lemma beta-eta-lift: \( s \beta \eta \rightarrow lift s k \Rightarrow lift \rightarrow \beta \eta \ast lift t k \)
proof (induction pred: rtranclp)
  case base show ?case ..
next
  case (step y z)
  hence lift y k \beta lift z k using lift-preserves-beta eta-lift by blast
with step.IH show lift s k \beta lift z k by iprover
qed

lemma confluent-beta-eta-reds: Joinable.confluent \{ (s, t). s \beta \eta \rightarrow 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 \Rightarrow dB \Rightarrow bool (infixl \leftrightarrow 50)
where term-equiv = joinablep beta-eta-reds

lemma term-equivI:
  assumes \( s \beta \eta \rightarrow u \) and \( t \beta \eta \rightarrow u \)
  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 \( s \rightarrow \beta \eta \ast u \) and \( t \rightarrow \beta \eta \ast u \)
  using assms unfolding term-equiv-def by (rule joinableE[to-pred])

lemma reds-into-equiv[elim]: \( s \beta \eta \rightarrow t \Rightarrow s \leftrightarrow t \)
by (blast intro: term-equivI)

lemma beta-into-equiv[elim]: \( s \beta \rightarrow t \Rightarrow s \leftrightarrow t \)
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-eq) (rule eta-into-beta-eta-reds)

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

lemma term-refl[iff]: t \leftrightarrow t
unfolding term-equiv-def by (blast intro: joinable-refl reflpI)

lemma term-sym[simp]: (s \leftrightarrow t) \implies (t \leftrightarrow s)
unfolding term-equiv-def by (rule joinable-sym[to-pred])

lemma conversep-term[simp]: conversep (\leftrightarrow) = (\leftrightarrow)
by (auto simp add: fun-eq-iff intro: term-sym)

lemma term-trans[trans]: s \leftrightarrow t \implies t \leftrightarrow u \implies s \leftrightarrow u
unfolding term-equiv-def
by (blast elim: transpE)

lemma term-beta-trans[trans]: s \leftrightarrow t \implies t \rightarrow_\beta u \implies s \leftrightarrow u
by (fast dest!: beta-into-beta-eta-reds intro: term-trans)

lemma term-eta-trans[trans]: s \leftrightarrow t \implies t \rightarrow_\eta u \implies s \leftrightarrow u
by (fast dest!: eta-into-beta-eta-reds intro: term-trans)

lemma equiv-appL[intro]: s \leftrightarrow s' \implies s \cdot t \leftrightarrow s' \cdot t
unfolding term-equiv-def using beta-eta-appL
by (iprover intro: joinable-subst[to-pred])

lemma equiv-appR[intro]: t \leftrightarrow t' \implies s \cdot t \leftrightarrow s' \cdot t'
unfolding term-equiv-def using beta-eta-appR
by (iprover intro: joinable-subst[to-pred])

lemma equiv-app: s \leftrightarrow s' \implies t \leftrightarrow t' \implies s \cdot t \leftrightarrow s' \cdot t'
by (blast intro: term-trans)

lemma equiv-abs[intro]: t \leftrightarrow t' \implies \text{Abs } t \leftrightarrow \text{Abs } t'
unfolding term-equiv-def using beta-eta-abs
by (iprover intro: joinable-subst[to-pred])

lemma equiv-lift: s \leftrightarrow t \implies \text{lift } s k \leftrightarrow \text{lift } t k
by (auto intro: term-equiv1 beta-eta-lift elim: term-equivE)

lemma equiv-liftn: s \leftrightarrow t \implies \text{liftn } n s k \leftrightarrow \text{liftn } n t k
by (induction n) (auto intro: equiv-lift)
Our definition is equivalent to the symmetric and transitive closure of the reduction relation.

\textbf{lemma} equiv-eq-rtsc-l-reds: \(\text{term-equiv} = (\text{sup beta-eta beta-eta}^{1-1})^{**}\)
\textbf{unfolding} term-eq-def
\textbf{using} confluent-beta-eta-reds
\textbf{by} (rule joinable-eq-rtsc[ta-pred])

\textbf{end}

5.3 Combinators defined as closed lambda terms

\textbf{theory} Combinators
\textbf{imports} Beta-Eta
begin

\textbf{definition} I-def: \(I = \text{Abs (Var 0)}\)
\textbf{definition} B-def: \(B = \text{Abs (Abs (Var 2 * (Var 1 * Var 0)))}\)
\textbf{definition} T-def: \(T = \text{Abs (Var 0 * Var 1)}\) — reverse application

\textbf{lemma} I-eval: \(I \cdot x \rightarrow \beta x\)
\textbf{proof} –
\textbf{have} \(I \cdot x \rightarrow \beta \text{Var 0}[x/0]\) \textbf{unfolding} I-def ..
\textbf{then show} \(?\text{thesis by simp}\)
\textbf{qed}

\textbf{lemma} I-equiv[iff]: \(I \cdot x \leftrightarrow x\)
\textbf{using} I-eval ..

\textbf{lemma} I-closed[simp]: \(\text{liftn n I k = I}\)
\textbf{unfolding} I-def \textbf{by simp}

\textbf{lemma} B-eval1: \(B \cdot g \rightarrow \beta \text{Abs (Abs (Var 2 * (Var 1 * Var 0)))}\)
\textbf{proof} –
\textbf{have} \(B \cdot g \rightarrow \beta \text{Abs (Var 2 * (Var 1 * Var 0))}[g/0]\) \textbf{unfolding} B-def ..
\textbf{then show} \(?\text{thesis by (simp add: numerals)}\)
\textbf{qed}

\textbf{lemma} B-eval2: \(B \cdot g \cdot f \rightarrow \beta^* \text{Abs (lift g 0 * (lift f 0 * Var 0))}\)
\textbf{proof} –
\textbf{have} \(B \cdot g \cdot f \rightarrow \beta^* \text{Abs (lift (lift g 0) 0 * (Var 1 * Var 0))}\) \textbf{using} B-eval1 \textbf{by blast}
\textbf{also have} \(\ldots \rightarrow \beta \text{Abs (lift (lift g 0) 0 * (Var 1 * Var 0))}[f/0]\) ..
\textbf{also have} \(\ldots = \text{Abs (lift g 0 * (lift f 0 * Var 0))}\) \textbf{by simp}
\textbf{finally show} \(?\text{thesis }\).
\textbf{qed}

\textbf{lemma} B-eval: \(B \cdot g \cdot f \cdot x \rightarrow \beta^* g \cdot (f \cdot x)\)
\textbf{proof} –
\textbf{have} \(B \cdot g \cdot f \cdot x \rightarrow \beta^* \text{Abs (lift g 0 * (lift f 0 * Var 0))} \cdot x\)
using B-eval2 by blast
also have \( \rightarrow_\beta (\text{lift } g \ 0 \ 0 \ 0 \ (\text{lift } f \ 0 \ 0 \ 0 \ \text{Var } 0)) [x/0] \). ..
also have \( = g \cdot (f \cdot x) \) by simp
finally show \( \text{thesis} \).
qed

lemma B-equiv[iff]: \( B \cdot g \cdot f \cdot x \leftrightarrow g \cdot (f \cdot x) \)
using B-eval ..

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

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

lemma T-eval: \( T \cdot x \cdot f \rightarrow_\beta^* f \cdot x \)
proof –
  have \( T \cdot x \cdot f \rightarrow_\beta^* \text{Abs } (\text{Var } 0 \cdot \text{lift } x \ 0) \cdot f \)
  using T-eval1 by blast
  also have \( \rightarrow_\beta (\text{Var } 0 \cdot \text{lift } x \ 0) [f/0] \) ..
  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]: lift 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 ◦ 150)

primrec opaque :: ′a itrm ⇒ ′a list
where
    opaque (Opaque x) = [x]
| opaque (Pure -) = []
| opaque (f ◦ x) = opaque f @ opaque x

abbreviation iorder x ≡ length (opaque x)

inductive itrm-cong :: (′a itrm ⇒ ′a itrm ⇒ bool) ⇒ ′a itrm ⇒ ′a itrm ⇒ bool for R
where
    into-itrm-cong: R x y ⇒ itrm-cong R x y
| pure-cong[intro]: x ↔ y ⇒ itrm-cong R (Pure x) (Pure y)
| ap-cong: itrm-cong R f f’ ⇒ itrm-cong R x x’ ⇒ itrm-cong R (f ◦ x) (f’ ◦ x’)
| itrm-refl[iff]: itrm-cong R x x
| itrm-sym[sym]: itrm-cong R x y ⇒ itrm-cong R y x
| itrm-trans[trans]: itrm-cong R x y ⇒ itrm-cong R y z ⇒ itrm-cong R x z

lemma ap-congL[intro]: itrm-cong R f f’ ⇒ itrm-cong R (f ◦ x) (f’ ◦ x)
by (blast intro: ap-cong)

lemma ap-congR[intro]: itrm-cong R x x’ ⇒ itrm-cong R (f ◦ x) (f ◦ x’)
by (blast intro: ap-cong)

Idiomatic terms are similar iff they have the same structure, and all contained lambda terms are equivalent.

abbreviation similar :: ′a itrm ⇒ ′a itrm ⇒ bool (infixl ≡ 50)
where x ≡ y ≡ itrm-cong (λ - . False) x y

lemma pure-similarE:
assumes Pure x’ ≡ y
obtains y’ where y = Pure y’ and x’ ↔ y’
proof –
define x :: ′a itrm where x = Pure x’
from assms have x ≡ y unfolding x-def .
then have (∀x’’. x = Pure x’’ → (∃y’. y = Pure y’ ∧ x’’ ↔ y’)) ∧
(∀x’’. y = Pure x’’ → (∃y’. x = Pure y’ ∧ x’’ ↔ y’))
proof (induction)
case pure-cong thus ?case by (auto intro: term-sym)
next
case itrm-trans thus ?case by (fastforce intro: term-trans)
qed simp-all
with that show thesis unfolding x-def by blast
lemma opaque-similarE:
assumes Opaque x' ∼ y
obtains y' where y = Opaque y' and x' = y'
proof -
define x :: 'a itrm where x = Opaque x'
from assms have x ∼ y unfolding x-def .
then have (∀ x''. x = Opaque x'' → (∃ y'. y = Opaque y' ∧ x'' = y')) ∧
(∀ x''. y = Opaque x'' → (∃ y'. x = Opaque y' ∧ x'' = y'))
by induction fast+
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' ∩ x2' ∩ y2)) ∧
(∀ x1' x2'. y = x1' ∩ x2' → (∃ y1 y2. x1 ∩ x2 = y1 ∩ y2 ∩ x1' ∩ 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 ≡ itrm-cong 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 ≃ y → iorder x = iorder y
by (auto dest: opaque-equiv)

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

definition idiom-ext-rule = sup idiom-rule extra-rule
abbreviation itrm-ext-equiv :: 'a itrm ⇒ 'a itrm ⇒ bool (infixl ≃+ 50)
where x ≃+ y ≡ itrm-cong idiom-ext-rule x y

lemma equiv-into-ext-equiv: x ≃ y → x ≃+ 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 ⇒ 'a itrm ⇒ nat ⇒ dB
  where
  unlift' n (Opaque _) i = Var i
  | unlift' n (Pure x) i = liftn n x 0
  | unlift' n (f ∘ x) i = unlift' n f (i + iorder x) * unlift' n x i

abbreviation unlift x ≡ (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 \textit{absn-cong[intro]}: \[s \leftrightarrow t \implies (\text{Abs}^n s) \leftrightarrow (\text{Abs}^n t)\]

by (induction \(n\)) auto

lemma \textit{free-unlift}: \[\text{free } (\text{unlift}' n x i) j \implies j \geq n \vee (j \geq i \land j < i + \text{iorder } x)\]

proof (induction \(x\) arbitrary: \(i\))
  case (\text{Opaque } x)
  thus \(?\text{case by simp}\)
next
  case (\text{Pure } x)
  thus \(?\text{case using free-liftn by simp}\)
next
  case (\text{IAp } x y)
  thus \(?\text{case by fastforce}\)
qed

lemma \textit{unlift-subst}: \[j \leq i \land j \leq n \implies (\text{unlift}' (\text{Suc } n) t (\text{Suc } i))[s/j] = \text{unlift}' n t i\]

proof (induction \(t\) arbitrary: \(i\))
  case (\text{Opaque } x)
  thus \(?\text{case by simp}\)
next
  case (\text{Pure } x)
  thus \(?\text{case using subst-liftn by simp}\)
next
  case (\text{IAp } x y)
  hence \(j \leq i + \text{iorder } y\) by simp
  with \(\text{IAp}\) show \(?\text{case by auto}\)
qed

lemma \textit{unlift'-equiv}: \[x \simeq y \implies \text{unlift}' n x i \leftrightarrow \text{unlift}' n y i\]

proof (induction arbitrary: \(n \ i \ \text{pred: itrm-cong}\))
  case (\text{into-itrm-cong } x y)
  thus \(?\text{case}\)
proof induction
  case (\text{idiom-id } x)
  show \(?\text{case using I-equiv[symmetric] by simp}\)
next
  case (\text{idiom-comp } g f x)
  let \(?G = \text{unlift}' n g (i + \text{iorder } f + \text{iorder } x)\)
  let \(?F = \text{unlift}' n f (i + \text{iorder } x)\)
  let \(?X = \text{unlift}' n x i\)
  have \(\text{unlift}' n (g \circ (f \circ x)) i = ?G \cdot (?F \cdot ?X)\)
    by (simp add: add.assoc)
  moreover have \(\text{unlift}' n (\text{Pure } B \circ g \circ f \circ x) i = B \cdot ?G \cdot ?F \cdot ?X\)
    by (simp add: add.commute add.left-commute)
  moreover have \(?G \cdot (?F \cdot ?X) \leftrightarrow B \cdot ?G \cdot ?F \cdot ?X\) using \(B\text{-equiv[symmetric]}\)
  ultimately show \(?\text{case by simp}\)
next
  case (\text{idiom-hom } f x)
show \( ?\text{case} \) by auto

next

  case (idiom-xchng \( f \) \( x \))  
  let \( ?F = \text{unlift}' n f i \)  
  let \( ?X = \text{liftn} n x 0 \)  
  have \( \text{unlift}' n (f \circ \text{Pure} x) i = ?F \cdot ?X \) by simp
  moreover have \( \text{unlift}' n (\text{Pure} (T' x) \circ f) i = T' \cdot ?X \cdot ?F \) by simp
  moreover have \( ?F' \cdot ?X \leftrightarrow T' \cdot ?X \cdot ?F \) using \text{\textit{T-equiv\[symmetric\].}}
  ultimately show \( ?\text{case} \) by simp

qed

next

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

next

  case (ap-cong \( f \) \( f' \) \( x \) \( x' \))
  from \( (x \simeq x') \) have \( \text{iorder-eq: iorder x = iorder x'} \) by (rule \text{iorder-equiv})
  have \( \text{unlift}' n (f \circ x) i = \text{unlift}' n f (i + \text{iorder x}) \cdot \text{unlift}' n x i \) by simp
  moreover have \( \text{unlift}' n (f' \circ x') i = \text{unlift}' n f' (i + \text{iorder x}) \cdot \text{unlift}' n x' i \) using \text{iorder-eq} by simp
  ultimately show \( ?\text{case} \) using \text{ap-cong.IH} by (auto intro: \text{equiv-app})

next

  case \text{\textit{itrn-refl}}
  thus \( ?\text{case} \) by simp

next

  case \text{\textit{itrn-sym}}
  thus \( ?\text{case} \) using \text{\textit{term-sym}} by simp

next

  case \text{\textit{itrn-trans}}
  thus \( ?\text{case} \) using \text{\textit{term-trans}} by blast

qed

lemma \text{\textit{unlift-equiv: x \simeq y}} \implies \text{unlift x} \leftrightarrow \text{unlift y}

proof

  assume \( x \simeq y \)
  then have \( \text{unlift}' (\text{iorder y}) x 0 \leftrightarrow \text{unlift}' (\text{iorder y}) y 0 \) by (rule \text{\textit{unlift'-equiv}})
  moreover from \( (x \simeq y) \) have \( \text{iorder x = iorder y} \) by (rule \text{\textit{iorder-equiv}})
  ultimately show \( ?\text{thesis} \) by auto

qed

Preserving variables  \textbf{primrec} \text{\textit{unlift-vars: nat \Rightarrow nat itrm \Rightarrow dB}}

where

\[ \text{\textit{unlift-vars n (Opaque i) = Var i}} \]
\[ | \text{\textit{unlift-vars n (Pure x) = liftn n x 0}} \]
\[ | \text{\textit{unlift-vars n (x \circ y) = unlift-vars n x \cdot unlift-vars n y}} \]

lemma \text{\textit{all-pure-unlift-vars: opaque x = [] \implies x \simeq Pure (unlift-vars 0 x)}}

proof (induction \( x \))

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

next

51
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 ∘ y) = unlift-vars 0 x ∘ unlift-vars 0 y 
    by simp 
  from no-opaque IAp.IH have x ∘ y ≃ Pure (unlif vars 0 x) ∘ Pure (unlif vars 0 y) 
    by (blast intro: ap-cong)
  also have ... ≃ Pure (unlif vars 0 x ∘ unlif vars 0 y) by (rule itrm-hom)
  also have ... = Pure (unlif vars 0 (x ∘ 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) = (Abs "?n (unlif' ?n f 1 ∘ Var 0)
    by simp
  also have ... = (Abs "iorder f) (Abs (unlif' ?n f 1 ∘ Var 0))
    using funpow-Suc-inside by simp
also have \( \leftrightarrow \) unlift \( f \) proof 

- have \( \neg \) free \( (\text{unlift'} \ ?n \ f \ 1) \ 0 \) using free-unlift by fastforce
  
  hence \( \text{Abs} \ (\text{unlift'} \ ?n \ f \ 1 \ Var \ 0) \rightarrow_{\eta} (\text{unlift'} \ ?n \ f \ 1)[\text{Var} \ 0/0] \)

  also have \( \exists = \text{unlift'} (\text{iorder} \ f) \ 0 \)

  using unlift-subst by (metis One-nat-def Suc-eq-plus1 le0)

  finally show \( ?\text{thesis} \)

  by (simp add: r-into-rtranclp absn-cong eta-into-equiv)

qed

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

using ap-cf.IH by (auto intro: term-sym term-trans)

qed

lemma cf-similarI:

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

and opaque \( x = \text{opaque} \ y \)

and CF-pure \( x \leftrightarrow \text{CF-pure} \ y \)

shows \( x \cong y \)

using assms proof (induction arbitrary: \( y \))

case (pure-cf \( x \))

hence opaque \( y = [] \) by auto

with \( y \in \text{CF} \) obtain \( y' \) where \( y = \text{Pure} \ y' \) by cases auto

with pure-cf.prems show \( ?\text{case} \) by auto

next

case (ap-cf \( f \ x \))

from \( \text{opaque} \ (f \circ \text{Opaque} \ x) = \text{opaque} \ y \)

obtain \( y1 \ y2 \) where opaque \( y = y1 @ y2 \)

and \( \text{opaque} \ f = y1 \) and \( [x] = y2 \) by fastforce

from \( [x] = y2 \) obtain \( y' \) where \( y2 = [y'] \) and \( x = y' \)

by auto

with \( y \in \text{CF} \) and \( \text{opaque} \ y = y1 @ y2 \) obtain \( g \)

where opaque \( g = y1 \) and \( \text{y-split: } y = g \circ \text{Opaque} \ y' \ g \in \text{CF} \) by cases auto

with ap-cf.prems \( \text{opaque} \ f = y1 \)

have \( \text{opaque} \ f = \text{opaque} \ g \text{-pure} \ f \leftrightarrow \text{CF-pure} \ g \) by auto

with ap-cf.IH \( \text{g} \in \text{CF} \) have \( f \cong g \) by simp

with ap-cf.prems \( \text{y-split} \ x = y' \) show \( ?\text{case} \) by (auto intro: ap-cong)

qed

lemma cf-similarD:

assumes \( \text{in-cf: } x \in \text{CF} \ y \in \text{CF} \)

and similar: \( x \cong y \)

shows \( \text{CF-pure} \ x \leftrightarrow \text{CF-pure} \ y \land \text{opaque} \ x = \text{opaque} \ y \)

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 \( \text{in-cf: } x \in \text{CF} \ y \in \text{CF} \)
and equiv: \( x \simeq y \) shows \( x \equiv y \)

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 \( \text{CF-pure} \ x \leftrightarrow \text{CF-pure} \ y \)
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

primrec norm-pn :: 'a itrm ⇒ 'a itrm
where
  norm-pn \( f \) (Opaque \( x \)) = undefined
| norm-pn \( f \) (Pure \( x \)) = Pure \( (f \cdot x) \)
| norm-pn \( f \) (\( n \odot x \)) = norm-pn \( (B \odot f) \) \( n \odot x \)

primrec norm-nn :: 'a itrm ⇒ 'a itrm
where
  norm-nn \( n \) (Opaque \( x \)) = undefined
| norm-nn \( n \) (Pure \( x \)) = norm-pn \( (T \odot x) \) \( n \)
| norm-nn \( n \) (\( n' \odot x \)) = norm-nn \( (norm-pn \ B \) \( n \)) \( n' \odot x \)

primrec norm :: 'a itrm ⇒ 'a itrm
where
  norm (Opaque \( x \)) = Pure \( I \odot \) Opaque \( x \)
| norm (Pure \( x \)) = Pure \( x \)
| norm \( f \odot x \) = norm-nn \( (\text{norm} \ f) \) (\( \text{norm} \) \( x \))

lemma norm-pn-in-cf:
assumes \( x \in \text{CF} \)
shows \( \text{norm-pn} \ f \ x \in \text{CF} \)
using assms
by (induction \( x \) arbitrary: \( f \)) auto

lemma norm-nn-in-cf:
assumes \( n \in \text{CF} \ n' \in \text{CF} \)
shows \( \text{norm-nn} \ n \ n' \in \text{CF} \)
using assms(2,1)
by (induction \( n' \) arbitrary: \( n \)) (auto intro: norm-pn-in-cf)

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

lemma norm-pn-equiv:
assumes $x \in CF$
  shows $\text{norm-pn} f x \simeq \text{Pure } f \circ x$
using assms proof (induction $x$ arbitrary: $f$)
  case (pure-cf $x$)
  have $\text{Pure } (f \cdot x) \simeq \text{Pure } f \circ \text{Pure } x$ using itrm-hom[symmetric].
  then show case by simp
next
  case (ap-cf $n$ $x$)
  from ap-cf.IH have $\text{norm-pn} (B \cdot f) n \simeq \text{Pure } (B \cdot f) \circ n$.
  then have $\text{norm-pn} (B \cdot f) n \circ \text{Opaque } x \simeq \text{Pure } (B \cdot f) \circ n \circ \text{Opaque } x$.
  also have ... $\simeq \text{Pure } B \circ \text{Pure } f \circ n \circ \text{Opaque } x$
  using itrm-hom[symmetric] by blast
  also have ... $\simeq \text{Pure } f \circ (n \circ \text{Opaque } x)$ using itrm-comp.
  finally show case by simp
qed

lemma norm-nn-equiv:
  assumes $n \in CF$ $n' \in CF$
  shows $\text{norm-nn } n n' \simeq n \circ n'$
using assms(2,1) proof (induction $n'$ arbitrary: $n$)
  case (pure-cf $x$)
  then have $\text{norm-pn} (T \cdot x) n \simeq \text{Pure } (T \cdot x) \circ n$ by (rule norm-pn-equiv)
  also have ... $\simeq n \circ \text{Pure } x$ using itrm-xchng[symmetric].
  finally show case by simp
next
  case (ap-cf $n'$ $x$)
  have $\text{norm-nn} (\text{norm-pn } B n) n' \circ \text{Opaque } x \simeq \text{Pure } B \circ n \circ n' \circ \text{Opaque } x$
proof
  from $(n \in CF)$ have $\text{norm-pn } B n \in CF$ by (rule norm-pn-in-cf)
  with ap-cf.IH have $\text{norm-nn} (\text{norm-pn } B n) n' \simeq \text{norm-pn } B n \circ n'$.
  also have ... $\simeq \text{Pure } B \circ n \circ n'$ using norm-pn-equiv $(n \in CF)$ by blast
  finally show norm-nn $(\text{norm-pn } B n) n' \simeq \text{Pure } B \circ n \circ n'$.
qed
  also have ... $\simeq n \circ (n' \circ \text{Opaque } x)$ using itrm-comp.
  finally show case by simp
qed

lemma norm-equiv: $\text{norm } x \simeq x$
proof (induction)
  case (Opaque $x$)
  have $\text{Pure } I \circ \text{Opaque } x \simeq \text{Opaque } x$ using itrm-id.
  then show case by simp
next
  case (Pure $x$)
  show case by simp
next
  case (IAp $f$ $x$)
  have $\text{norm } f \in CF$ and $\text{norm } x \in CF$ by (rule norm-in-cf)+
  then have $\text{norm-nn } (\text{norm } f) (\text{norm } x) \simeq \text{norm } f \circ \text{norm } x$
lemma normal-form: obtains \( n \) where \( n \simeq x \) and \( n \in CF \)
using norm-equiv norm-in-cf ..

5.4.5 Lifting with normal forms

lemma nf-unlift:
assumes equiv: \( n \simeq x \) and cf: \( n \in CF \)
shows \( CF\text{-pure } n \leftrightarrow \text{unlift } x \)
proof –
from cf have \( CF\text{-pure } n \leftrightarrow \text{unlift } n \) by (rule cf-unlift)
also from equiv have \( \text{unlift } n \leftrightarrow \text{unlift } x \) by (rule unlift-equiv)
finally show \( \text{thesis} . \)
qed

theorem 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 nf-x: \( n \simeq x \) \( n \in CF \) by (rule normal-form)
also from nf-x have \( \text{unlift } x \leftrightarrow \text{unlift } \text{opaque } x \) by (rule opaque-equiv)
also from nf-y have \( \text{unlift } y \leftrightarrow \text{unlift } \text{opaque } y \) by (rule opaque-equiv)
finally have \( \text{opaque-eq}: \text{opaque } n = \text{opaque } n' \).

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

from nf-x(1) have \( x \simeq n \) ..
also have \( n \simeq n' \)
using nf-x nf-y pure-eq opaque-eq
by (blast intro: similar-into-eqv cf-similarI)
also from 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 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 \(\text{var-dist } = \text{ fold } (\lambda i t . t \cdot \text{Var } i)\)

**Lemma**: \(\text{var-dist-Nil}[\text{simp}]: \text{var-dist } [] \cdot t = t\)
unfolding \(\text{var-dist-def} \text{ by simp}\)

**Lemma**: \(\text{var-dist-Cons}[\text{simp}]: \text{var-dist } (v \# vs) t = \text{var-dist } (t \cdot \text{Var } v)\)
unfolding \(\text{var-dist-def} \text{ by simp}\)

**Lemma**: \(\text{var-dist-append1} [\text{simp}]: \text{var-dist } (vs \@ [v]) t = \text{var-dist } vs (t \cdot \text{Var } v)\)
unfolding \(\text{var-dist-def} \text{ by simp}\)

**Lemma**: \(\text{var-dist-frees} [\text{simp}]: \text{frees } (\text{var-dist } vs t) = \text{frees } t \cup \text{set vs}\)
by (induction vs arbitrary: \(t\)) \(\text{auto}\)

**Lemma**: \(\text{var-dist-subst-lt} [\text{simp}]: \forall v \in \text{set vs}, i < v \Rightarrow (\text{var-dist vs } s)[t/i] = \text{var-dist } (\text{map } (\lambda v. v - 1) vs) (s[t/i])\)
by (induction vs arbitrary: \(s\)) \(\text{simp-all}\)

**Definition**: \(\text{vsubst} :: \text{nat } \Rightarrow \text{nat } \Rightarrow \text{nat } \Rightarrow \text{nat}\)
where \(\text{vsubst } u v w = (\text{if } u < w \text{ then } u \text{ else if } u = w \text{ then } v \text{ else } u - 1)\)

**Lemma**: \(\text{vsubst-subst-lt} [\text{simp}]: u < w \Rightarrow \text{vsubst } u v w = u\)
unfolding \(\text{vsubst-def} \text{ by simp}\)

**Lemma**: \(\text{var-dist-subst-Var} [\text{simp}]: (\text{var-dist vs s})[t/i] = \text{var-dist } (\text{map } (\lambda v. \text{vsubst } v i j) vs) (s[t/i])\)
by (induction vs arbitrary: \(s\)) \(\text{simp-all}\)

**Lemma**: \(\text{var-dist-cong} [\text{simp}]: s \leftrightarrow t \Rightarrow \text{var-dist vs s } \leftrightarrow \text{var-dist vs t}\)
by (induction vs arbitrary: \(s, t\)) \(\text{auto}\)

**Preliminaries**: Eta reductions with permuted variables

**Lemma**: \(\text{absn-subst} [\text{simp}]: ((\text{Abs}^{\sim n}) s)[t/k] = (\text{Abs}^{\sim n}) (s[\text{liftn } n \cdot t 0/k+n])\)
by (induction \(n\) arbitrary: \(t, k\)) \(\text{simp-all add: liftn-lift-swap}\)

**Lemma**: \(\text{absn-beta-equiv} [\text{simp}]: (\text{Abs}^{\sim \text{Suc } n}) s \cdot t \leftrightarrow (\text{Abs}^{\sim n}) (s[\text{liftn } n \cdot t 0/n])\)
proof –
have \((\text{Abs}^{\sim \text{Suc } n}) s \cdot t = \text{Abs } ((\text{Abs}^{\sim n}) s) \cdot t \text{ by simp}\)
also have ... \(\leftrightarrow ((\text{Abs}^{\sim n}) s)[t/0]\) by (rule beta-into-equiv) (rule beta.beta)
also have \( = (\operatorname{Abs}^n n) (s[liftn n t 0/n]) \) by (simp add: absn-subst)
finally show \(?thesis\).
qed

lemma absn-dist-eta: \( (\operatorname{Abs}^n n) (\operatorname{var-dist} (\operatorname{rev} [0..<n]) (liftn n t 0)) \leftrightarrow t \)
proof (induction n)
  case 0 show \(?case\) by simp
next
case (Suc n)
  let \(?dist-range = \lambda k. \operatorname{var-dist} (\operatorname{rev} [0..<Suc n]) (liftn k t 0)\)
  have dist-last: \(?dist-range 0 (Suc n) = ?dist-range 1 (Suc n) \cdot \operatorname{Var} 0\)
    unfolding append var-dist-append1
    by (rule eta)
  also have \( \neg \operatorname{free} (?dist-range 1 (Suc n)) 0 \) proof –
    have frees (?dist-range 1 (Suc n)) = frees (liftn (Suc n) t 0) \cup \{1..n\}
      unfolding var-dist-frees by fastforce
    then have 0 \notin \operatorname{frees} (?dist-range 1 (Suc n)) by simp
    then show \(?thesis\) by simp
  qed
  then have \( \operatorname{Abs} (?dist-range 0 (Suc n)) \rightarrow_\eta (?dist-range 1 (Suc n)) [\operatorname{Var} 0/0]\)
    unfolding dist-last by (rule eta)
  also have \( \ldots = \operatorname{var-dist} (\operatorname{rev} [0..<n]) ((\operatorname{liftn} (Suc n) t 0)[\operatorname{Var} 0/0]) \) proof –
    have \( \forall v \in \operatorname{set} (\operatorname{rev} [1..<Suc n]). 0 < v \) by auto
    moreover have \( \operatorname{rev} [0..<n] = \operatorname{map} (\lambda v. v - 1) (\operatorname{rev} [1..<Suc n]) \) by (induction n)
      simp-all
    ultimately show \(?thesis\) by (simp only: var-dist-subst-lt)
  qed
  also have \( \ldots = ?dist-range 0 n \) using subst-liftn[of 0 n t \Var 0]\ by simp
  finally have \( \operatorname{Abs} (?dist-range 0 (Suc n)) \leftrightarrow ?dist-range 0 n \ldots \)
    unfolding funpow-Suc-inside by (rule absn-cong)
  also from Suc.IH have \( \ldots \leftrightarrow t \).
  finally show \(?case\).
qed

primrec strip-context :: nat \Rightarrow dB \Rightarrow nat \Rightarrow dB
where
  | strip-context n (\Var i) k = (if i < k then \Var i else \Var (i - n))
  | strip-context n (\operatorname{Abs} t) k = \operatorname{Abs} (strip-context n t (Suc k))
  | strip-context n (s * t) k = strip-context n s k * strip-context n t k

lemma strip-context-liftn: \( (\operatorname{liftn} (m + n) t k) k = \operatorname{liftn} m t k\)
by (induction t arbitrary: k) simp-all

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

definition \( \text{perm-vars} :: \mathbb{N} \Rightarrow \mathbb{N} \text{-list} \Rightarrow \mathbb{B} \)
where \( \text{perm-vars } n \ vs \leftrightarrow \text{distinct } vs \land \text{set } vs = \{0..<n\} \)

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

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

lemma \( \text{perm-vars-lt} : \text{perm-vars } n \ vs \Rightarrow \forall i \in \text{set } vs. \ i < n \)
unfolding \( \text{perm-vars-def} \) by simp

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

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

abbreviation perm-vars-inv :: nat ⇒ nat list ⇒ nat ⇒ nat
where perm-vars-inv n vs i ≡ the-inv-into {0..<n} ((!) vs) i

lemma perm-vars-inv-nth:
  assumes perm-vars n vs
  and i < n
  shows perm-vars-inv n vs (vs ! i) = i
using assms by (auto intro: the-inv-into-f-f perm-vars-inj-on-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 -
define vsubsts where vsubsts n vs' vs =
  map (λ v.
    if v < n − length vs' then v
    else if v < n then vs' ! (n − v − 1) + (n − length vs')
    else v − length vs') vs
for n vs' vs
let ?app-vars = λ t 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``?m) (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')
  define n' where 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))
  * Var v
    unfolding var-dist-Cons ..
  also have ... ↔ ?app-vars t n' vs' (vsubsts n [v] vs) proof (rule var-dist-cong)
have map (\lambda v. vsubst vv (v + n') n) vs = vsubsts n [v] vs
  unfolding Suc-n'[symmetric] vsubsts-def vsubst-def
  by (auto cong: if-cong)
then have (var-dist vs (liftn (Suc n') t 0)) [liftn n' (Var v) 0/n']
  = var-dist (vsbstds 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)) \cdot Var v
  \leftrightarrow (Abs'"n') (var-dist (vsbstds n [v] vs) (liftn n' t 0))
  by (fastforce intro: absn-beta-equiv[THEN term-trans])
qed
also have ... \leftrightarrow (Abs'"?m') (var-dist (vsbstds n' vs' (vsbstds n [v] vs)) (liftn
  ?m' t 0))
  using vs'-length Cons.IH by blast
also have ... = (Abs'"?m') (var-dist (vsbstds n (v \# vs') vs) (liftn ?m' t 0))
proof
  have vsbstds n' vs' (vsbstds (Suc n') [v] vs) = vsbstds (Suc n') (v \# vs') vs
  unfolding vsbstds-def
  using vs'-length [[linarith-split-limit=10]]
  by auto
then show ?thesis unfolding Suc-n' by simp
qed
finally show ?case unfolding m'-conv.
qed
}

note partial-appd = this

define vs' where vs' = map (\lambda i. n - perm-vars-inv n vs (n - i - 1) - 1)
  [0..<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 (\lambda v. 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 (\lambda v 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 ... = n - i - 1 using perm-vars perm-vars-inv-nth
    by simp
    also from (i < n) have ... = rev [0..<n] ! i by (simp add: rev-nth)
    finally show vs' ! (n - vs ! i - 1) = rev [0..<n] ! i .
    qed
then show ?thesis
  unfolding list.rel-eq[symmetric]
using list.rel-map
by auto
qed
then have vs'-vs: vs\textsubscript{substs} n vs' vs = rev [0..<n]
  unfolding vs\textsubscript{substs-def} vs'-length
  using perm-vars perm-vars-lt
  by (auto intro: map-ext[THEN trans])

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

lemma liftn-absn: liftn n ((Abs\textsuperscript{"{m}}) t) k = (Abs\textsuperscript{"{m}}) (liftn n t (k + m)) 
by (induction m arbitrary: k) auto

lemma liftn-var-dist-lt:
\forall i \in \text{set vs}. i < k \implies liftn n (\text{var-dist vs t}) k = \text{var-dist vs (liftn n t k)} 
by (induction vs arbitrary: t) auto

lemma liftn-context-conv: k \leq k' \implies \forall i \in \text{frees t}. i < k \lor k' \leq i \implies liftn n t k = liftn n t k' 
proof (induction t arbitrary: k k')
  case (Abs t)
  have \forall i \in \text{frees t}. i < \text{Suc} k \lor \text{Suc} k' \leq i proof 
    fix i assume i \in \text{frees t} 
    show i < \text{Suc} k \lor \text{Suc} k' \leq i proof (cases i = 0)
      assume i = 0 then show \(?thesis by simp 
    next
      assume i \neq 0 
      from Abs.prems(2) have \forall i. free t (\text{Suc} i) \longrightarrow i < k \lor k' \leq i by auto 
      then have \forall i. 0 < i \land free t i \longrightarrow i - 1 < k \lor k' \leq i - 1 by simp 
      then have \forall i. 0 < i \land free t i \longrightarrow i < \text{Suc} k \lor \text{Suc} k' \leq i by auto 
      with \langle i \neq 0 : i \in \text{frees t} \rangle show \(?thesis by simp 
    qed
    qed
  with Abs.IH Abs.prems(1) show \(?case by auto 
  qed auto
lemma liftn-liftn0: \( \forall i \in \text{frees} \, t \,. \, k \leq i \implies \text{liftn} n \, t \, k = \text{liftn} n \, t \, 0 \)
using liftn-context-conv by auto

lemma dist-perm-eta-equiv:
assumes perm-vars: perm-vars n vs
and not-free: \( \forall i \in \text{frees} \, s \,. \, n \leq i \land \forall i \in \text{frees} \, t \,. \, i \leq n \)
shows strip-context n s 0 \( \iff \) strip-context n t 0
proof
from perm-vars have vs-lt-n: \( \forall i \in \text{set} \, vs \,. \, i < n \)
using perm-vars-lt by simp
obtain vs′ where etas: \( \forall i \in \text{frees} \, t \,. \, n \leq i \implies (\text{Abs}^\sim n \, (\text{var-dist} \, vs \, t)) \iff \text{strip-context} \, n \, t \, 0 \)
using perm-vars dist-perm-eta by blast
have strip-context n s 0 \( \iff \) (Abs\(^\sim n\) (var-dist vs′ ((Abs\(^\sim n\) (var-dist vs (liftn n t 0))))) \iff strip-context n t 0
using perm-vars dist-perm-eta by blast
have strip-context n s 0 \( \iff \) (Abs\(^\sim n\) (var-dist vs′ ((Abs\(^\sim n\) (var-dist vs (liftn n t 0)))))
proof
have eta\(\sim\)s[TEN then term-sym] not-free(1) .
also have \( \ldots \iff (Abs\(^\sim\) n \, (\text{var-dist} \, vs′ \, ((Abs\(^\sim\) n \, (\text{var-dist} \, vs (\text{liftn} n \, t \, 0))))) \)
proof
have (Abs\(^\sim n\) (var-dist vs (liftn n t 0))) = (Abs\(^\sim n\) (var-dist vs (liftn n s n)))
using not-free(1) liftn-liftn0[of s n] by simp
also have \( \ldots = (Abs\(^\sim n\) \, (\text{var-dist} \, vs \, s) \, n) \)
using liftn-var-dist-lt by simp
also have \( \ldots = \text{liftn} \, n \, ((Abs\(^\sim n\) \, (\text{var-dist} \, vs \, s)) \, 0) \)
using liftn-absn by simp
also have \( \ldots \iff \text{liftn} \, n \, ((Abs\(^\sim n\) \, (\text{var-dist} \, vs \, t)) \, 0) \)
using perm-equiv by (rule equiv-liftn)
also have \( \ldots = (Abs\(^\sim n\) \, (\text{var-dist} \, vs \, t) \, n) \)
using liftn-absn by simp
also have \( \ldots = (Abs\(^\sim n\) \, (\text{var-dist} \, vs (\text{liftn} \, n \, t \, n)) \)
using vs-lt-n liftn-var-dist-lt by simp
also have \( \ldots = (Abs\(^\sim n\) \, (\text{var-dist} \, vs (\text{liftn} \, n \, t \, 0)) \)
using not-free(2) liftn-liftn0[of t n] by simp
finally show (Abs\(^\sim n\) (var-dist vs (liftn n s 0))) \( \iff \) \ldots .
qed
also have \( \ldots \iff \text{strip-context} \, n \, t \, 0 \)
using etas not-free(2) .
finally show \( \text{thesis} \).
qed

General notion of bracket abstraction for lambda terms  
definition foldr-option :: \( \prime a \Rightarrow \prime b \Rightarrow \prime b \text{ option} \) 
where foldr-option f xs e = foldr \( (\lambda a. \text{Option.bind} \, b \, (f \, a)) \) \, xs \, (\text{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 \( \Rightarrow \) dB \( \Rightarrow \) dB option
  assumes bracket-app: term-bracket i s = Some s' \( \Rightarrow \) \( s' \circ \text{Var } i \leftrightarrow s \)
  assumes bracket-frees: term-bracket i s = 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 = Some t
unfolding term-brackets-def by simp

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

lemma term-brackets-ConsI:
  assumes term-brackets vs t = Some t'
    and term-bracket v t' = Some t''
  shows term-brackets (v#vs) t = Some t''
using assms unfolding term-brackets-def foldr-option-def by simp

lemma term-brackets-dist:
  assumes term-brackets vs t = Some t'
    shows \( \text{var-dist } vs t' \leftrightarrow t \)
proof -
  from assms have \( \forall t'', \text{t'} \leftrightarrow t'' \rightarrow \text{var-dist } vs t'' \leftrightarrow t \)
proof (induction vs arbitrary: t')
  case Nil then show ?case by (simp add: term-sym)
next
  case (Cons v vs)
  from Cons.prems obtain u where
    inner: term-brackets vs t = Some u
    and step: term-bracket v u = Some t'
    by (auto elim: term-brackets-Cons-SomeE)
  from step have red1: t'' \( \text{Var } v \leftrightarrow u \) by (rule bracket-app)
  show ?case 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 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 var-dist \( (v \# vs) t'' \leftrightarrow t \).
qed
qed
then show ?thesis by blast
qed

end

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

lemma strip-unlift-vars:
assumes opaque \( x = [] \)
sows strip-context \( n (\text{unlift-vars } n \ x) 0 = \text{unlift-vars } 0 \ x \)
using assms by (induction \( x \)) (simp-all add: strip-context-liftn[where \( m=0 \), simplified])

lemma unlift-vars-frees: \( \forall i \in \text{frees } (\text{unlift-vars } n \ x), \ i \in \text{set } (\text{opaque } x) \lor n \leq i \)
by (induction \( x \)) (auto simp add: free-liftn)

locale itrm-abstraction = special-idiom extra-rule for extra-rule :: nat itrm \( \Rightarrow \cdot + \)
fixes itrm-bracket :: nat \( \Rightarrow \cdot \text{nat } \Rightarrow \text{nat itrm option} \)
assumes itrm-bracket-ap: itrm-bracket \( i \ x = \text{Some } x' \) \( \Rightarrow x' \circ \text{Opaque } i \approx x \)
assumes itrm-bracket-opaque:
\( \text{itrm-bracket } i \ x = \text{Some } x' \Rightarrow \text{set } (\text{opaque } x') = \text{set } (\text{opaque } x) - \{ i \} \)
begin

definition itrm-brackets = foldr-option itrm-bracket

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

lemma itrm-brackets-Cons-SomeE:
assumes itrm-brackets \( (v\#vs) \ x = \text{Some } x' \)
obtains \( y' \text{ where itrm-brackets } vs x = \text{Some } y' \) and itrm-bracket \( v y' = \text{Some } x' \)
using assms unfolding itrm-brackets-def by (elim foldr-option-Cons-SomeE)

definition opaque-dist = fold (\( \lambda i \ y. \ y \circ \text{Opaque } i \))

lemma opaque-dist-cong: \( x \approx y \Rightarrow \text{opaque-dist } vs x \approx \text{opaque-dist } vs y \)
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 -
  define x'' where x'' = 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' \circ Opaque v \simeq^+ y' by (elim itrm-bracket-ap)
  then have x'' \circ Opaque v \simeq^+ y'
  using Cons.prems(2) by (blast intro: itrm-sym itrm-trans)
  note this[symmetric]
  with defined' have fold (\lambda i y. y \circ Opaque i) vs (x'' \circ Opaque v) \simeq^+ 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) \subseteq 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 \textit{itrms-brackets-all-unlif vars}:
\begin{itemize}
\item \textbf{assumes} all-opaque: set \((\text{opaque } x) \subseteq \text{set } \text{vs}\)
\item and defined: \text{itrms-brackets } \text{vs } x = \text{Some } x' \hfill \\
\item shows \(x' \sim+ \text{Pure (unlif vars } 0 x')\)
\end{itemize}
\textbf{proof} (rule equiv-into-ext-eqv)
\begin{itemize}
\item from assms have \text{opaque } x' = [] by (rule itrm-brackets-all)
\item then show \(x' \sim+ \text{Pure (unlif vars } 0 x')\) by (rule all-pure-unlif vars)
\end{itemize}
qed

5.4.7 Lifting with bracket abstraction

locale \textit{lifted-bracket} = \textit{bracket-abstraction} + \textit{itrms-abstraction} +
\begin{itemize}
\item \textbf{assumes} bracket-compat:
\item set \((\text{opaque } x) \subseteq \{0..<n\} \implies i < n \implies \text{term-bracket } i \text{ (unlif vars } n x) = \text{map-option (unlif vars } n \text{ (itrms-bracket } i x)\)
\end{itemize}
\begin{itemize}
\item \textbf{begin}
\item \textbf{lemmas} brackets-unlif vars-swap:
\item \textbf{assumes} all-opaque: set \((\text{opaque } x) \subseteq \{0..<n\}\)
\item and vs-bound: set vs \(\subseteq \{0..<n\}\)
\item and defined: \text{itrms-brackets } \text{vs } x = \text{Some } x' \hfill \\
\item shows \text{term-brackets vs (unlif vars } n x) = \text{Some (unlif vars } n x')\)
\item \textbf{using} vs-bound defined \textbf{proof} (induction vs arbitrary: \(x')\)
\item case Nil
\item then show \(\text{case by simp}\)
\item \textbf{next}
\item case \((\text{Cons v vs})\)
\item then obtain \(y'\)
\item where ivs: \text{itrms-brackets vs } x = \text{Some } y'
\item and iv: \text{itrms-bracket } v y' = \text{Some } x'
\item by (elim itrm-brackets-Cons-SomeE)
\item \textbf{with} Cons \textbf{have} \text{term-brackets vs (unlif vars } n x) = \text{Some (unlif vars } n y')\)
\item by auto
\item \textbf{moreover}\{ 
\item have \(\text{Some (unlif vars } n x') = \text{map-option (unlif vars } n \text{ (itrms-bracket } v y')\)
\item unfolding iv by simp
\item \textbf{moreover have} set \((\text{opaque } y') \subseteq \{0..<n\}\)
\item \textbf{using} all-opaque ivs by (auto dest: itrm-brackets-opaque)
\item \textbf{moreover have} \(v < n\) using Cons.prems by simp
\item \textbf{ultimately have} \text{term-bracket } v \text{ (unlif vars } n y') = \text{Some (unlif vars } n x')
\item \textbf{using} bracket-compat by auto
\item \}
\item \textbf{ultimately show} \(\text{case by (rule term-brackets-ConsI)}\)
\item \textbf{qed}
theorem bracket-lifting:

assumes all-vars: set (opaque x) \( \cup \) set (opaque y) \( \subseteq \{0..<n\}

and perm-vars: perm-vars n vs

and defined: \( \text{itrm-brackets} \quad vs 
\quad x = \text{Some} 
\quad \text{itrm-brackets} \quad vs 
\quad y = \text{Some} 
\)

and base-eq: \( (\text{Abs}^\sim n) 
\quad (\text{unlift-vars} 
\quad n 
\quad x) 
\leftrightarrow (\text{Abs}^\sim n) 
\quad (\text{unlift-vars} 
\quad n 
\quad y) 
\)

shows \( x \simeq^+ y \)

proof –

\[ \text{from perm-vars have set-vs: set vs = } \{0..<n\} \]

\[ \text{unfolding perm-vars-def by simp} \]

\[ \text{have x-swap: term-brackets vs (unlift-vars n x) = Some (unlift-vars n x')} \]

\[ \text{using all-vars set-vs defined(1) by (auto intro: brackets-unlift-vars-swap)} \]

\[ \text{have y-swap: term-brackets vs (unlift-vars n y) = Some (unlift-vars n y')} \]

\[ \text{using all-vars set-vs defined(2) by (auto intro: brackets-unlift-vars-swap)} \]

\[ \text{from all-vars have set (opaque x) \( \subseteq \) set vs unfolding set-vs by simp} \]

\[ \text{then have complete-x: opaque x' = []} \]

\[ \text{using defined(1) itrm-brackets-all by blast} \]

\[ \text{then have ux-frees: } \forall i \in \text{frees (unlift-vars n x')}. \quad n \leq i \]

\[ \text{using unf} \text{lift-vars-frees by fastforce} \]

\[ \text{from all-vars have set (opaque y) \( \subseteq \) set vs unfolding set-vs by simp} \]

\[ \text{then have complete-y: opaque y' = []} \]

\[ \text{using defined(2) itrm-brackets-all by blast} \]

\[ \text{then have uy-frees: } \forall i \in \text{frees (unlift-vars n y')}. \quad n \leq i \]

\[ \text{using unf} \text{lift-vars-frees by fastforce} \]

\[ \text{have x \simeq^+ opaque-dist vs x'} \]

\[ \text{using defined(1) by (rule itrm-brackets-dist[symmetric])} \]

\[ \text{also have ... \simeq^+ opaque-dist vs (Pure (unlift-vars 0 x'))} \]

\[ \text{using all-vars set-vs defined(1)} \]

\[ \text{by (auto intro: opaque-dist-cong \text{itrm-brackets-all-unlift-vars})} \]

\[ \text{also have ... \simeq^+ opaque-dist vs (Pure (unlift-vars 0 y'))} \]

\[ \text{proof (rule opaque-dist-cong, rule pure-cong)} \]

\[ \text{have (Abs}^\sim n) \quad (\text{var-dist vs (unlift-vars n x'))} \leftrightarrow (Abs}^\sim n) \quad (\text{unlift-vars n x)} \]

\[ \text{using x-swap term-brackets-dist by auto} \]

\[ \text{also have ... } \leftrightarrow (Abs}^\sim n) \quad (\text{unlift-vars n y)} \text{ using base-eq} , \]

\[ \text{also have ... } \leftrightarrow (Abs}^\sim n) \quad (\text{var-dist vs (unlift-vars n y'))} \]

\[ \text{using y-swap term-brackets-dist[THEN term-sym] by auto} \]

\[ \text{finally have strip-context n (unlift-vars n x') 0 } \leftrightarrow \text{strip-context n (unlift-vars n y')} \quad 0 \]

\[ \text{by (intro dist-perm-eta-equiv)} \]

\[ \text{then show unlift-vars 0 x' } \leftrightarrow \text{unlift-vars 0 y'} \]

\[ \text{using strip-unlift-vars complete-x complete-y by simp} \]

\[ \text{finally show x \simeq^+ y'} \]

\[ \text{proof (rule opaque-dist-cong)} \]

\[ \text{show Pure (unlift-vars 0 y') \simeq^+ y'} \]

\[ \text{qed} \]

also have ... \simeq^+ opaque-dist vs y' proof (rule opaque-dist-cong)

show Pure (unlift-vars 0 y') \simeq^+ 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


