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 ........................................ 6
   2.1 Environment functor .......................................... 6
   2.2 Option .......................................................... 6
2.3 Sum types ................................................. 8
2.4 Set with Cartesian product ......................... 10
2.5 Lists ...................................................... 10

3 Distinct, non-empty list .......................... 12
  3.1 Monoid .................................................. 16
  3.2 Filters .................................................. 17
  3.3 State monad ............................................ 18
  3.4 Streams as an applicative functor ............... 18
  3.5 Open state monad .................................. 20
  3.6 Probability mass functions ....................... 20
  3.7 Probability mass functions implemented as lists with duplicates 22
  3.8 Ultrafilter ............................................. 23

4 Examples of applicative lifting ................... 24
  4.1 Algebraic operations for the environment functor .... 24
  4.2 Pointwise arithmetic on streams ................... 25
  4.3 Tree relabelling ...................................... 29
    4.3.1 Pure correctness statement .................. 30
    4.3.2 Correctness via monadic traversals ........... 31
    4.3.3 Applicative correctness statement .......... 36
    4.3.4 Probabilistic tree relabelling ................ 37

5 Formalisation of idiomatic terms and lifting ...... 39
  5.1 Immediate joinability under a relation .......... 39
    5.1.1 Definition and basic properties ............ 39
    5.1.2 Confluence ..................................... 40
    5.1.3 Relation to reflexive transitive symmetric closure 41
    5.1.4 Predicate version .............................. 41
  5.2 Combined beta and eta reduction of lambda terms 42
    5.2.1 Auxiliary lemmas ................................ 42
    5.2.2 Reduction ...................................... 42
    5.2.3 Equivalence ................................... 43
  5.3 Combinators defined as closed lambda terms .... 45
  5.4 Idiomatic terms – Properties and operations .... 46
    5.4.1 Basic definitions .............................. 47
    5.4.2 Syntactic unlifting ........................... 49
    5.4.3 Canonical forms ................................ 52
    5.4.4 Normalisation of idiomatic terms ............ 54
    5.4.5 Lifting with normal forms ................... 56
    5.4.6 Bracket abstraction, twice .................... 57
    5.4.7 Lifting with bracket abstraction ............ 67
1 Lifting with applicative functors

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

begin

1.1 Equality restricted to a set

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

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

lemma rel-fun-map-fun2: rel-fun (eq-on (range h)) A f g ⇒ rel-fun (BNF-Def.Grp UNIV h)^−1−1 A f (map-fun h id g)
  by(auto simp add: rel-fun-def Grp-def eq-onp-def)

lemma rel-fun-refl-eq-onp
  by(auto simp add: rel-fun-def eq-onp-def)

lemma eq-onE: [ eq-on X a b; [ b ∈ X; a = b ] ] ⇒ thesis ] ⇒ thesis by auto

lemma Domainp-eq-on [simp]: Domainp (eq-on X) = (λx. x ∈ X)
by auto

1.2 Proof automation

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

lemma UNIV-E: x ∈ UNIV ⇒ P ⇒ P .

context begin

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

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

lemmas [combinator-repr] = combinator-unfold
private definition cpair ≡ Pair
private definition cuncurry ≡ case-prod

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

ML-file applicative.ML

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

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

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

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

end

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

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

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

ML (Outer-Syntax.local-theory-to-proof {
  command-keyword applicative
})
register applicative functors
  (Parse.binding --
    Scan.optional (@{keyword} () |-- Parse.list Parse.short-ident --| @{keyword}}) []|--
    (@{keyword for} |-- Parse.reserved pure |-- @{keyword :} |-- Parse.term)|--
    (Parse.reserved ap |-- @{keyword} |-- Parse.term) --
    Scan.option (Parse.reserved rel |-- @{keyword :} |-- Parse.term) --
    Scan.option (Parse.reserved set |-- @{keyword :} |-- Parse.term) >>
    Applicative.applicative-cmd)}

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

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

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

1.3 Overloaded applicative operators

consts
  pure :: 'a => 'b
  ap :: 'a => 'b => 'c

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

hide-const (open) ap
Common applicative functors

2.1 Environment functor

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

definition const x = (λ.. x)
definition apf x y = (λz. x z (y z))

adhoc-overloading Applicative.pure const
adhoc-overloading Applicative.ap apf

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

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

lemma
    includes applicative-syntax
    shows const (λ\ m. \n f y x) \ f \ x \ y = \ f \ y \ x
by applicative-lifting simp

end

2.2 Option

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

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

abbreviation (input) pure-option :: 'a ⇒ 'a option
where
    pure-option ≡ Some
adhoc-overloading Applicative.pure pure-option

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

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

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

applicative option (C, W)
for
pure: Some
ap: ap-option
rel: rel-option
set: set-option

proof –
include applicative-syntax
{ fix x :: 'a option
  show pure (λx. x) ⊙ x = x by (cases x) simp-all
next
  fix g :: ('b ⇒ 'c) option and f :: ('a ⇒ 'b) option and x
  show pure (λg f x. g (f x)) ⊙ g ⊙ f ⊙ x = g ⊙ (f ⊙ x)
    by (cases g f x rule: option.exhaust[case-product option.exhaust, case-product option.exhaust]) simp-all
next
  fix f :: ('b ⇒ 'a ⇒ 'c) option and x y
  show pure (λf x y. f 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 f :: ('a ⇒ 'b) option and x
  show pure (λf x. f x x) ⊙ f ⊙ x = f ⊙ x ⊙ x
    by (cases f x rule: 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: option.rel-refl-strong)
    show rel-option R (f ⊙ x) (g ⊙ x) by transfer-prover
}
\textbf{QED} \((\textsf{simp add: some-ap-option ap-some-option})\)

\textbf{Lemma} \textit{map-option-ap-conv[applicative-unfold]}: \textit{map-option} \(f\ x = \textsf{ap-option} (\textsf{pure} f) x\)
\textbf{by} (cases \(x\) rule: \textsf{option.exhaust}) simp-all

\textbf{No-adhoc-overloading} \textit{Applicative.pure pure-option} — We do not want to print all occurrences of \textit{Some} as \textit{pure}

\textbf{end}

\subsection{2.3 Sum types}

\textbf{Theory} \textit{Applicative-Sum} \textbf{imports} \textit{Applicative} \textit{HOL-Library.Adhoc-Overloading}
\textbf{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 \textit{Either} datatype, which prefers the value of the left operand, and a generic one using the \textit{semigroup-add} class. Only the former lifts the \textit{W} combinator, though.

\textbf{Fun} \textit{ap-sum} :: \((\lambda e \Rightarrow \lambda e \Rightarrow \lambda e) \Rightarrow (\lambda a \Rightarrow \lambda b \Rightarrow \lambda e \Rightarrow (\lambda a + \lambda e \Rightarrow \lambda a + \lambda e \Rightarrow \lambda b + \lambda e)\)
\textbf{where}
\begin{align*}
\textit{ap-sum} - (\text{Inl} f) (\text{Inl} x) &= \text{Inl} (f x) \\
\textit{ap-sum} - (\text{Inl} -) (\text{Inr} e) &= \text{Inr} e \\
\textit{ap-sum} - (\text{Inr} e) (\text{Inl} -) &= \text{Inr} e \\
\textit{ap-sum} c (\text{Inr} e1) (\text{Inr} e2) &= \text{Inr} (c e1 e2)
\end{align*}

\textbf{Abbreviation} \textit{ap-either} \equiv \textit{ap-sum} (\lambda x -. x)
\textbf{Abbreviation} \textit{ap-plus} \equiv \textit{ap-sum} (\lambda a -. \textit{semigroup-add} \Rightarrow -)

\textbf{Abbreviation} \textit{(input) pure-sum where} \textit{pure-sum} \equiv \textit{Inl}
\textbf{Adhoc-overloading} \textit{Applicative.pure pure-sum}
\textbf{Adhoc-overloading} \textit{Applicative.ap ap-either}

\textbf{Lemma} \textit{ap-sum-id}: \textit{ap-sum} \(c\) (\text{Inl} \text{\textit{id}}) \(x = x\)
\textbf{by} (cases \(x\)) simp-all

\textbf{Lemma} \textit{ap-sum-ichng}: \textit{ap-sum} \(c\) \(f\) (\text{Inl} \(x\)) = \textit{ap-sum} \(c\) (\text{Inl} (\lambda f \cdot f x)) \(f\)
\textbf{by} (cases \(f\)) simp-all

\textbf{Lemma} \textit{(in semigroup} \textit{) 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 \((\lambda x \ y \ . \ x)\)
by unfold-locales simp

locale either-af =
  fixes B :: \(\prime b \Rightarrow \prime b \Rightarrow \text{bool}\)
  assumes B-refl: reflp B
begin

applicative either \((W)\)
for
  pure: Inl
  ap: ap-either
  rel: \(\lambda A \ R B\)

proof –
include applicative-syntax
{ fix f :: \(\prime c \Rightarrow \prime c \Rightarrow \prime a \ \text{and} \ x\)
  show pure \((\lambda f x \ . \ f x x) \circ f \circ x = f \circ x \circ x\)
  by (cases f x rule: sum.exhaust[case-product sum.exhaust]) simp-all
next
  interpret semigroup \(\lambda x \ y \ . \ x\) \(\text{by}\) (rule semigroup-const)
  fix g :: \(\prime d \Rightarrow \prime e \ \text{and} \ f :: \(\prime c \Rightarrow \prime d \ \text{and} \ x\)
  show pure \((\lambda g f x \ . \ g (f x)) \circ g \circ f \circ x = g \circ (f \circ x)\)
  by (rule ap-sum-comp[abs-def])
next
  fix R and f :: \(\prime c \Rightarrow \prime d \ \text{and} \ g :: \(\prime c \Rightarrow \prime e \ \text{and} \ x\)
  assume rel-sum \((\text{rel-fun \ (eq-on \ UNIV) \ R}) B f g\)
  then show rel-sum R B \((f \circ x) \ (g \circ x)\)
  by (cases f g x rule: sum.exhaust[case-product sum.exhaust])
  (auto intro: B-refl THEN reflpD elim: rel-funE)
} qed (auto intro: ap-sum-id[abs-def] ap-sum-ichng)

end

interpretation either-af \(=\) \(\text{by}\) unfold-locales simp

applicative semigroup-sum
for
  pure: Inl
  ap: ap-plus

using
  ap-sum-id[abs-def]
  ap-sum-ichng
  add.ap-sum-comp[abs-def]
by auto

no-adhoc-overloading Applicative.pure pure-sum

end

2.4 Set with Cartesian product

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

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

adhoc-overloading Applicative.ap ap-set

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

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

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

end

2.5 Lists

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

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

**lemma**  
*Nil-ap[simp]*:  
ap-list [] xs = []

**unfolding**  
ap-list-def by simp

**lemma**  
ap-Nil[simp]:  
ap-list fs [] = []

**unfolding**  
ap-list-def by (induction fs) simp-all

**lemma**  
ap-list-transfer[transfer-rule]:

rel-fun (list-all2 (rel-fun A B)) (rel-fun (list-all2 A) (list-all2 B)) ap-list ap-list

**unfolding**  
ap-list-def[abs-def] List.bind-def

by transfer-prover

**context includes**  
aplicative-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 = ∀g 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
have \([?B] \circ (g \# gs) \circ f \circ x = [?B] g \circ f \circ x \circ [?B] gs \circ f \circ x\)
  by (simp add: cons-ap-list append-ap-distrib)
also have ...
also have ...
finally show \(?case\).
qed
next
fix \(f :: (\tau \Rightarrow \sigma)\ list\) and \(x\)
show \(f \circ [x] = [\lambda f. f x] \circ f\)
  unfolding ap-list-def by simp
next
fix \(R :: (\tau \Rightarrow \sigma \Rightarrow bool)\)
show rel-fun \((\text{list-all2} R) (\lambda x. [x]) (\lambda x. [x])\)
  by transfer-prover
next
fix \(R\) and \(f :: (\tau \Rightarrow \sigma)\ list\) and \(g :: (\tau \Rightarrow \sigma)\ list\) and \(x\)
assume \((\text{transfer-rule} 1): \text{list-all2} (\text{rel-fun} (\text{eq-on} (\text{set} x)) R) f g\)
have \((\text{transfer-rule} 2): \text{list-all2} \ (\text{eq-on} \ (\text{set} x)) x x\)
  by (simp add: list-all2_same)
show \(\text{list-all2} R \ (f \circ x) \ (g \circ x)\)
  by transfer-prover
qed (simp add: cons-ap-list)

lemma \(\text{map-ap-conv}\): \(\text{map} f x = [f] \circ x\)
unfolding ap-list-def List.bind-def by simp

end
end

3 Distinct, non-empty list

theory Applicative-DNEList imports
  Applicative-List
  HOL-Library.Dlist
begin
lemma \(\text{bind-eq-Nil-iff}\): \(\text{List.bind} \ x s f = [] \iff (\forall x \in \text{set} x. f x = [])\)
  by(simp add: List.bind-def)
lemma \(\text{zip-eq-Nil-iff}\): \(\text{zip} \ x s y s = [] \iff xs = [] \lor ys = []\)
  by(cases \(x\) \(y\); rule: list.exhaust(case-product List.exhaust)) simp-all
lemma \(\text{remdups-append1}\): \(\text{remdups} \ (\text{remdups} \ x s @ y s) = \text{remdups} \ (x s @ y s)\)
  by(induction \(x s\)) simp-all
lemma \(\text{remdups-append2}\): \(\text{remdups} \ (x s @ \text{remdups} y s) = \text{remdups} \ (x s @ y s)\)
  by(induction \(x s\)) simp-all
lemma \(\text{remdups-append1-drop}\): \(\text{set} x \subseteq \text{set} y s \Longrightarrow \text{remdups} \ (x s @ y s) = \text{remdups} y s\)

12
by (induction xs) auto

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

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

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

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

setup-lifting type-definition-dnelist

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

lift-bnf (no-warn-transfer, no-warn-wits) 'a dnelist via dnelist-subtype-dlist for
map: map
  by (auto simp: dlist-eq-iff)
hide-const (open) map

context begin
qualified lemma map-def: Applicative-DNELList.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.pcr-cr-eq cr-dnelist-def list-of-dnelist[simplified]

13
qualified lift-definition single :: 'a ⇒ 'a dnelist is \( \lambda x. [x] \) by simp

qualified lift-definition insert :: 'a ⇒ 'a dnelist ⇒ 'a dnelist is \( \lambda x. xs. \) if \( x \in \) set xs then xs else \( x \# xs \) by auto

qualified lift-definition append :: 'a dnelist ⇒ 'a dnelist ⇒ 'a dnelist is \( \lambda x. xs. \) if \( x \in \) set xs then xs else \( x \# xs \) by auto

qualified lift-definition bind :: 'a dnelist ⇒ ('a ⇒ 'b dnelist) ⇒ 'b dnelist is \( \lambda x. \) if \( x \in \) set xs then xs else \( x \# xs \) 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 \( \lambda f x. \) 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)

close context includes applicative-syntax begin

lemma ap-pure-dlist: 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 (\( \lambda x. x \)) ⋄ x = x for x :: 'a dnelist
by transfer simp

have *: remdups (remdups (remdups ([\lambda g 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 (\( \lambda f x. f (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 (\( \lambda f x. f (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 (λ g. map g (concat (map (λ f. map f x) f)))) g)))
  using list.pure-B-centric[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 (λ g 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 *)
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 (λ f. f x) ∘ 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 ≠ []
  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 (λ f. map f y) (map (λ x 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-remdups-replicate)
  finally show ?thesis .
qed

show pure-dnelist (λ x 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 (λ f 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
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, 'c) monoid-ap})))
)

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

applicative monoid-add
  for pure: pure-monoid-add
  ap: ap-monoid-add
subgoal by(simp add: pure-monoid-add-def)
subgoal for g f x by(cases g f x rule: monoid-ap.exhaust|case-product monoid-ap.exhaust,
  case-product monoid-ap.exhaust|)(simp add: pure-monoid-add-def add.assoc)
subgoal for x by(cases x)(simp add: pure-monoid-add-def)
subgoal for f x 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|
apply(rule monoid-add.pure-I-conv)
done

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

16
applicative idemp-monoid-add \( W \)
  for pure: pure-monoid-add :: - \( \Rightarrow \) (- :: idemp-monoid-add, -) monoid-ap
  ap: ap-monoid-add :: (- :: idemp-monoid-add, -) monoid-ap \( \Rightarrow \)
  apply(rule monoid-add.homomorphism monoid-add.pure-B-conv monoid-add.pure-I-conv)+
subgoal for \( f x \) by(cases \( f x \) 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 (+) \( \circ \) (\( x :: (\text{nat, int}) \) monoid-ap) \( \circ \) y = pure (+) \( \circ \) y \( \circ \) 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 :: \('a \Rightarrow \text{'a filter}\) where
  pure-filter \( x \) = principal \{\( x \}\}
definition ap-filter :: (\('a \Rightarrow \text{'b filter}\) \Rightarrow \text{'a filter} \Rightarrow \text{'b filter}\) where
  ap-filter \( F \) \( X \) = filtermap (\( \lambda (f, x). f x \)) (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 \( W \) is holds not available
for
  pure: pure-filter
  ap: ap-filter
  rel: rel-filter
proof
  show ap-filter (pure-filter \( f \)) (pure-filter \( x \)) = pure-filter \( f \) \( x \) for \( f :: \('a \Rightarrow \text{'b} \)
and \( x \)

17
by (simp add: ap-filter-def pure-filter-def principal-prod-principal)

show ap-filter (ap-filter (pure-filter (λ 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 (λ x. 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 (pure-filter (λ f x y. f y x) f) x) y =
ap-filter (ap-filter f y) x for f :: ('b ⇒ 'a ⇒ 'c) filter and x y


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

done

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
\[
\begin{align*}
\text{shd} \ (\text{ap-stream} \ f \ x) &= \text{shd} \ f \ (\text{shd} \ x) \\
| \stl \ (\text{ap-stream} \ f \ x) &= \text{ap-stream} \ (\stl \ f) \ (\stl \ x)
\end{align*}
\]

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

context includes lifting-syntax applicative-syntax
begin

lemma ap-stream-id: pure \((\lambda x. x) \circ x = x\)
by (coinduction arbitrary: \(x\)) simp

lemma ap-stream-homo: pure \(f \circ pure \ x = pure \ (f \ x)\)
by coinduction simp

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

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

applicative stream \((S, K)\)
for
pure: sconst
ap: ap-stream
rel: stream-all2
set: sset
proof –
fix g :: \((\Rightarrow) \Rightarrow \Rightarrow) \ stream\ and\ \ f\ x\ 
show pure \((\lambda g f x. g \ (f \ x)) \circ g \circ f \circ x = g \circ (f \circ x)\)
by (coinduction arbitrary: \(g \ f \ x\)) auto
next
fix x :: \(\Rightarrow) \ stream\ and\ \ y :: \(\Rightarrow) \ stream\ 
show pure \((\lambda x y. x) \circ x \circ y = x\)
by (coinduction arbitrary: \(x \ y\)) auto
next
fix R :: \(\Rightarrow) \ \Rightarrow 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 :: \((\Rightarrow) \ \Rightarrow) \ stream\ and\ \ g :: \((\Rightarrow) \ \Rightarrow) \ stream\ and\ \ x\ 
assume \([\text{transfer-rule}]\): \(stream-all2 \ (eq-on \ (sset \ x)) == R \ f \ g\)
have \([\text{transfer-rule}]\): \(stream-all2 \ (eq-on \ (sset \ x)) \ x \ x\ by(simp add: \text{stream.rel-refl-strong})\)
show stream-all2 R (f ∘ x) (g ∘ x) by transfer-prover
qed (rule ap-stream-homo)

lemma smap-applicative: smap f x = pure f ∘ x
unfolding ap-stream-def by (coinduction arbitrary: x) auto

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

end
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 \( \text{ap-pmf } f \ x = \text{map-pmf} (\lambda f, x). f x \) (\( \text{pair-pmf } f \ x \))

ad hoc overloading\( \text{Applicative.ap } \text{ap-pmf} \)

context includes\( \text{applicative-syntax} \)

begin

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

lemma \( \text{ap-pmf-comp: } \text{pure-pmf} \circ u \odot v \odot w = u \odot (v \odot w) \) by (simp add: \( \text{ap-pmf-def pair-return-pmf1 pair-map-pmf1 pair-map-pmf2 pmf.map-comp o-def split-def pair-pair-pmf} \))

lemma \( \text{ap-pmf-homo: } \text{pure-pmf } f \odot \text{pure-pmf } x = \text{pure-pmf} (f x) \) by (simp add: \( \text{ap-pmf-def pair-return-pmf1} \))

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

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

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

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

applicative \( \text{pmf} (C, K) \)
for
pure: \( \text{pure-pmf} \)
ap: \( \text{ap-pmf} \)
rel: \( \text{rel-pmf} \)
set: \( \text{set-pmf} \)

proof  
fix \( R :: 'a \Rightarrow 'b \Rightarrow \text{bool} \)
show \( \text{rel-fun } R \ (\text{rel-pmf } R) \ \text{pure-pmf pure-pmf} \) by transfer-prover

next
fix \( R \) and \( f :: ('a \Rightarrow 'b) \text{ pmf} \) and \( g :: ('a \Rightarrow 'c) \text{ pmf} \) and \( x \)
assume [(transfer-rule): \( \text{rel-pmf} (\text{rel-fun (eq-on (set-pmf x))}) R \) \( f \) \( g \)]
have [(transfer-rule): \( \text{rel-pmf} (eq-on (set-pmf x)) \) \( x \) \( x \) by (simp add: \( \text{pmf.rel-refl-strong} \))]

21
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 (λ. sum-list (f x)) xs)
by(induction xs) simp-all

context includes applicative-syntax begin

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

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

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

setup-lifting type-definition-pfp

lift-definition pure-pfp :: ('a ⇒ 'b) pfp ⇒ 'a pfp
is λfs xs. [λ(f, p) (x, 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) ⋆ 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 (λx. x) ★ x = x for x :: 'a pfp
    by transfer(simp add: ap-list-def List.bind-def)
  show pure-pfp f ★ pure-pfp x = pure-pfp (f x) for f :: 'a ⇒ 'b and x
    by transfer(applicative-lifting; simp)
  show pure-pfp (λg f x. g (f x)) ★ g ★ f ★ x = g ★ (f ★ x)
    for g :: ('b ⇒ 'c) pfp and f :: ('a ⇒ 'b) pfp and x
    by transfer(applicative-lifting; clarsimp)
  show f ★ pure-pfp x = pure-pfp (λf. f x) ★ f for f :: ('a ⇒ '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-of x = star-of (f x) for f x by(fact Ifun-star-of)
qued(transfer; rule refl)+

end

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

definition pure-vec :: 'a ⇒ ('a, 'b :: finite) vec
  where pure-vec x = (x - . x)
definition \( \text{ap-vec} :: (\alpha \Rightarrow \beta, \gamma :: \text{finite}) \text{ vec} \Rightarrow (\alpha, \gamma) \text{ vec} \Rightarrow (\beta, \gamma) \text{ vec} \)

where

\[
\text{ap-vec} f x = (\chi i. (f \$ i) (x \$ i))
\]

adhoc-overloading \( \text{Applicative}.ap \ \text{ap-vec} \)

applicative \( \text{vec} (K, W) \)

for

pure: \( \text{pure-vec} \)
ap: \( \text{ap-vec} \)

by \( \text{(auto simp add: pure-vec-def ap-vec-def vec-nth-inverse)} \)

lemma \( \text{pure-vec-nth [simp]}: \text{pure-vec} x \$ i = x \)

by \( \text{(simp add: pure-vec-def)} \)

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

by \( \text{(simp add: ap-vec-def)} \)

end

theory \text{Applicative-Functor} imports

\text{Applicative-Environment}
\text{Applicative-Option}
\text{Applicative-Sum}
\text{Applicative-Set}
\text{Applicative-List}
\text{Applicative-DNELList}
\text{Applicative-Monoid}
\text{Applicative-Filter}
\text{Applicative-State}
\text{Applicative-Stream}
\text{Applicative-Open-State}
\text{Applicative-PMF}
\text{Applicative-Probability-List}
\text{Applicative-Star}
\text{Applicative-Vector}
begin

print-applicative

end

4 Examples of applicative lifting

4.1 Algebraic operations for the environment functor

theory \text{Applicative-Environment-Algebra} imports

\text{Applicative-Environment}

24
HOL - Library.Function-Division

begin

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

category includes applicative-syntax
begin

lemma plus-fun-af [applicative-unfold]: f + g = pure (+) o f o g
unfolding plus-fun-def const-def apf-def ..

lemma zero-fun-af [applicative-unfold]: 0 = pure 0
unfolding zero-fun-def const-def ..

lemma times-fun-af [applicative-unfold]: f * g = pure (*) o f o g
unfolding times-fun-def const-def apf-def ..

lemma one-fun-af [applicative-unfold]: 1 = pure 1
unfolding one-fun-def const-def ..

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

lemma inverse-fun-af [applicative-unfold]: inverse f = pure inverse o f
unfolding inverse-fun-def o-def const-def apf-def ..

lemma divide-fun-af [applicative-unfold]: divide f g = pure divide o f o g
unfolding divide-fun-def const-def apf-def ..

end

end

4.2 Pointwise arithmetic on streams

theory Stream-Algebra
imports Applicative-Stream
begin

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

instantiation stream :: (one) one begin
definition [applicative-unfold]: 1 = sconst 1
instance .. end
instantiation stream :: (plus) plus begin
context includes applicative-syntax begin
definition [applicative-unfold]: \( x + y = \text{pure} \ (\text{+}) \circ x \circ (y :: 'a \text{ stream}) \)
end
instance ..
end

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

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

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

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

instantiation stream :: (modulo) modulo begin
context includes applicative-syntax begin
definition [applicative-unfold]: \( x \text{ div} y = \text{pure} \ (\text{div}) \circ x \circ (y :: 'a \text{ stream}) \)
definition [applicative-unfold]: \( x \text{ mod} y = \text{pure} \ (\text{mod}) \circ x \circ (y :: 'a \text{ stream}) \)
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
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 correctness property states that the list of labels in the obtained tree is distinct. As observed by Gibbons and Bird [1], this breaks the abstraction of the state monad, because the relabeling function must be run. Although Hutton and Fulger are careful to reason in point-free style, they nevertheless unfold the implementation of the state monad operations. Gibbons and Hinze [2] suggest to state the correctness in an effectful way using an exception-state
monad. Thereby, they lose the applicative structure and have to resort to a full monad.

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

```plaintext
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-::'a. 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 

```plaintext
definition labels :: 'a tree ⇒ 'a list
where labels = fold-tree (λx. [x]) append
lemma labels-simps [simp]:
  labels (Leaf x) = [x]
  labels (Node l r) = labels l @ labels r
by(simp-all add: labels-def)
locale labelling =
  fixes fresh :: ('s, 'x) state
begin
declare [[show-variants]]
definition label-tree :: 'a tree ⇒ ('s, 'x tree) state
where label-tree = fold-tree (λ-::'a. pure Leaf o fresh) (λl r. pure Node o l o r)
lemma label-tree-simps [simp]:
  label-tree (Leaf x) = pure Leaf o fresh
  label-tree (Node l r) = pure Node o label-tree l o label-tree r
by(simp-all add: label-tree-def)
primrec label-list :: 'a list ⇒ ('s, 'x list) state
where
  label-list [] = pure []
```

30
\[ \text{label-list} \ (x \ # \ xs) = \text{pure} \ (\#) \circ \text{fresh} \circ \text{label-list} \ xs \]

**Lemma** \( \text{label-append} \): 
\[ \text{label-list} \ (a \ @ \ b) = \text{pure} \ (\@) \circ \text{label-list} \ a \circ \text{label-list} \ b \]
— The proof lifts the defining equations of \((@)\) to the state monad.

**Proof** (induction \( a \))

- **Case** Nil
  - **Show** \( ?\text{case} \)
    - **Unfolding** \text{append.simps} label-list.simps
      - **By** applicative-nf simp

- **Next**
  - **Case** \((\text{Cons} \ a1 \ a2)\)
    - **Show** \( ?\text{case} \)
      - **Unfolding** \text{append.simps} label-list.simps Cons.IH
      - **By** applicative-nf simp

**Qed**

**Lemma** \( \text{label-tree-list} \): 
\[ \text{pure} \ \text{labels} \circ \text{label-tree} \ t = \text{label-list} \ (\text{labels} \ t) \]

**Proof** (induction \( t \))

- **Case** Leaf
  - **Show** \( ?\text{case} \)
    - **Unfolding** \text{label-tree-simps} \text{labels-simps} \text{label-list.simps}
      - **By** applicative-nf simp

- **Next**
  - **Case** Node
    - **Show** \( ?\text{case} \)
      - **Unfolding** \text{label-tree-simps} \text{labels-simps} \text{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** \( t :: 'a \text{ tree} \)
- **Assumes** \( \text{distinct} :: \bigwedge xs :: 'a \text{ list. distinct} \ (\text{fst} \ (\text{run-state} \ (\text{label-list} \ xs) \ s)) \)
- **Shows** \( \text{distinct} \ (\text{labels} \ (\text{fst} \ (\text{run-state} \ (\text{label-tree} \ t) \ s))) \)

**Using** \( \text{label-tree-list[of} \ t, \ \text{THEN arg-cong, of} \ \lambda f. \ \text{run-state} \ f \ s] \ \text{assms[of labels t]}

**By** (cases \( \text{run-state} \ (\text{label-list} \ (\text{labels} \ t)) \ s) \text{(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 = \text{UNIV :: 'a set morphisms un-B B by blast}

**Setup-lifting** type-definition-dual

**Lift-definition** pure-dual :: \('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b dual

**Is** \( \lambda \text{pure. pure} \)

**Lift-definition** ap-dual :: \(('a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b) \Rightarrow 'af1) \Rightarrow ('af1 \Rightarrow 'af2 \Rightarrow 'af13) \Rightarrow ('af13 \Rightarrow 'af2 \Rightarrow 'af) \Rightarrow 'af2 dual \Rightarrow 'af3 dual \Rightarrow 'af dual

31
is λpure ap1 ap2 f x. ap2 (ap1 (pure (λx f x)) x) f .

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

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

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

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

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

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

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

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

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

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

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

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

lemma ap-state-rev-rev-B: B f ∘ B x = B (pure-state-rev (λx f x) ∘ x ∘ f)
unfolding ap-state-rev-rev-def by (fact ap-dual.abs-eq)
lemma ap-state-rev-rev-pure-B: pure f ⬤ B x = B (pure-state-rev f ⬤ x)
unfolding ap-state-rev-rev-def pure-state-rev-rev-def
by transfer(applicative-nf, rule refl)

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

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 ⬤ State-Monad bind y f = State-Monad bind y ((pure x) ⬤ f)
by(simp add: ap-conv-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 (stl s)) (λ-. State-Monad.return (shd s)))

primrec traverse :: (′a ⇒ (′s, ′b) state) ⇒ (′s, ′b tree) state
where
traverse f (Leaf x) = pure Leaf ⬤ f x
| traverse f (Node l r) = pure Node ⬤ traverse f l ⬤ traverse f r
As we cannot abstract over the applicative functor in definitions, we define traversal on the transformed applicative function once again.

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

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

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

lemma recurse-Node:
recurse f (Node l r) = pure (λx l. Node l r) ⬤ recurse f r ⬤ recurse f l
proof
  have recurse f (Node l r) = un-B (pure Node ⬤ traverse-rev (B ⬤ f) l ⬤ traverse-rev (B ⬤ f) r)
  by(simp add: recurse-def)
also have ... = un-B (B (pure Node) ⋄ B (recurse f l) ⋄ B (recurse f r))
  by(simp add: un-B-inverse recurse-def pure-state-rev-def pure-dual-def)
also have ... = pure (λx f. f x) ⋄ recurse f r ⋄ (pure (λx f. f x) ⋄ recurse f l ⋄ pure Node)
  by(simp add: ap-state-rev-B B-inverse)
also have ... = pure (λr l. Node l r) ⋄ recurse f r ⋄ recurse f l
  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 . . . = un-B ∘ un-B ∘ traverse-rev-rev (B ∘ B ∘ f) unfolding traverse-B-B
by (simp add: o-assoc)
also have . . . = ?rhs by (simp add: recurse-rev-def o-assoc)
finally show ?thesis .
qed

lemma recurse-traverse:
assumes f · g = pure
shows recurse f · traverse g = pure
— Gibbons and Bird impose this as an additional requirement on traversals, but they write that they have not found a way to derive this fact from other axioms. So we prove it directly.
proof
fix t
from assms have *: ∀x. State-Monad.bind (g x) f = State-Monad.return x
by (simp add: fun-eq-iff)
hence **: ∀x h. State-Monad.bind (g x) (λx. State-Monad.bind (f x) h) = h x
by (fold State-Monad.bind-assoc)(simp)
show (recurse f ∘ traverse g) t = pure t unfolding kleisli-state-def
proof (induction t)
  case (Leaf x)
  show ?case
  by (simp add: ap-conv-bind-state recurse-Leaf **)
next
  case (Node l r)
  show ?case
qed
qed

Apply traversals to labelling

definition strip :: 'a × 'b ⇒ ('b stream, 'a) state
where strip = (λ(a, b). State-Monad.bind (State-Monad.update (SCons b)) (λ-.
State-Monad.return a))

definition adorn :: 'a ⇒ ('b stream, 'a × 'b) state
where adorn a = pure (Pair a) ∘ fetch

abbreviation label :: 'a tree ⇒ ('b stream, ('a × 'b) tree) state
where label ≡ traverse adorn
abbreviation unlabel :: ('a × 'b) tree ⇒ ('b stream, 'a tree) state
where unlabel ≡ recurse strip

lemma strip-adorn: strip • adorn = pure

lemma correctness-monadic: unlabel • label = pure
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 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]:
\[
\begin{align*}
\text{dlabels} \left( \text{Leaf } x \right) &= \text{pure } [x] \\
\text{dlabels} \left( \text{Node } l \ r \right) &= \text{pure } \left( \text{case-prod append} \odot \left( \text{assert } \text{case-prod disjoint} \right) \left( \text{pure } \text{Pair} \odot \text{dlabels } l \odot \text{dlabels } r \right) \right) \\
\text{by} \left( \text{simp-all add: dlabels-def} \right)
\end{align*}
\]

**lemma** correctness-applicative:

**assumes** distinct: \( \forall n. \text{pure } \left( \text{assert } \text{distinct} \right) \odot \text{symbols } n = \text{symbols } n \)

**shows** State-Monad.return dlabels \odot \text{label-tree } t = \text{symbols } \left( \text{leaves } t \right)  

**proof**(induction \( t \))

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

**unfolding** label-tree-simps leaves-simps repeatM.simps by applicative-nf simp

next

fix \( l \ r :: 'a \text{ tree} \)

**assume** IH: pure dlabels \odot \text{label-tree } l = \text{symbols } \left( \text{leaves } l \right) \text{ pure dlabels } \odot \text{label-tree } r = \text{symbols } \left( \text{leaves } r \right)

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

let \(?f = \lambda l \ r. \text{pure } ?\text{cat} \odot (\text{assert } ?\text{disj} ) (\text{pure } \text{Pair} \odot l \odot r)\)

**have** State-Monad.return dlabels \odot \text{label-tree } \left( \text{Node } l \ r \right) =

pure \( f \odot (\text{pure dlabels } \odot \text{label-tree } l) \odot (\text{pure dlabels } \odot \text{label-tree } r) \)

**unfolding** label-tree-simps by applicative-nf simp

**also have** \ldots = pure \( f \odot (\text{pure } \text{assert distinct} ) \odot \text{symbols } \left( \text{leaves } l \right) \odot (\text{pure } \text{assert distinct} ) \odot \text{symbols } \left( \text{leaves } r \right) \)

**unfolding** IH distinct \ldots

**also have** \ldots = pure (\text{assert distinct} ) \odot \text{symbols } \left( \text{leaves } \left( \text{Node } l \ r \right) \right)

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

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

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

qed

end

4.3.4 Probabilistic tree relabelling

**primrec** mirror :: 'a \text{ tree } \Rightarrow 'a \text{ tree}  

**where**

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

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

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

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

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

**where**

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

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

| subtree \left( \text{Leaf } x \right) = \text{Leaf } x  

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

**by** pat-completeness auto
termination by lexicographic-order

adhoc-overloading Applicative.pure pure-pmf

context fixes p :: 'a ⇒ 'b pmf begin

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

lemma plabel-mirror: plabel (mirror t) = pure mirror ⊗ plabel t
proof (induction t)
  case (Leaf x)
  show ?case unfolding plabel.simps mirror.simps by (applicative-lifting) simp
next
  case (Node t1 t2)
  show ?case unfolding plabel.simps mirror.simps Node.IH by (applicative-lifting)
  simp
qed

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

end

end

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


5 Formalisation of idiomatic terms and lifting

5.1 Immediate joinability under a relation

theory Joinable
imports Main
begin

5.1.1 Definition and basic properties

definition joinable :: ('a × 'b) set ⇒ ('a × 'a) set
where joinable R = \{ (x, y). \exists 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 and (y, z) ∈ R
using assms unfolding joinable-simp by blast

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

lemma refl-joinable-iff: (∀x. \exists 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 $R \subseteq \text{joinable } R$
proof (rule subrelI)
  fix $x \ y$
  assume $(x, y) \in R$
  moreover from $\langle \text{refl } R \rangle$ 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 $R\text{-subst}: \forall x \ y. \ (x, y) \in R \Rightarrow (P x, P y) \in R$
  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 $R\text{-subst } xz$ have $(P x, P z) \in R$ .
  moreover from $R\text{-subst } yz$ have $(P y, P z) \in R$ .
  ultimately show $?\text{thesis}$ by (rule joinableI)
qed

5.1.2 Confluence

definition confluent :: 'a rel \Rightarrow bool
where confluent $R$ \iff $(\forall x \ y \ y'. \ (x, y) \in R \Rightarrow (x, y') \in R \Rightarrow (y, y') \in \text{joinable } R)$

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

lemma confluentE:
  assumes $\text{confluent } R \text{ and } (x, y) \in R \text{ and } (x, y') \in R$
  obtains $z$ where $(y, z) \in R \text{ and } (y', z) \in R$
  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 $(x, y) \in \text{joinable } R$
where \( \text{joinablep } P \ x \ y \)

**Definition**

5.1.4 Predicate version

**Theorem**

\[\text{joinable-eq-rtscl} \]

**Proof**

\[\text{rtrancl-converseI} \]

\[\text{qed} \]

\[\text{next} \]

\[\text{proof} \]

\[\text{lemm}\text{a joinable-le-rtscl} \]

5.1.3 Relation to reflexive transitive symmetric closure

**Lemma** \( \text{joinable-le-rtsc}\ell: \text{joinable } (R^*) \subseteq (R \cup R^{-1})^* \)

**Proof** (rule subrelI)

\[\text{fix } x \ y \]

\[\text{assume } (x, y) \in \text{joinable } (R^*) \]

\[\text{then obtain } z \text{ where } xz: (x, z) \in R^* \text{ and } yz: (y, z) \in R^* \text{ by (rule joinableE)} \]

\[\text{from } xz \text{ have } (x, z) \in (R \cup R^{-1})^* \text{ by (blast intro: in-rtrancl-UnI)} \]

\[\text{moreover from } yz \text{ have } (z, w) \in (R \cup R^{-1})^* \text{ by (blast intro: in-rtrancl-UnI rtrancl-converseI)} \]

\[\text{ultimately show } (x, y) \in (R \cup R^{-1})^* \text{ by (rule rtrancl-trans)} \]

**Qed**

**Theorem** \( \text{joinable-eq-rtscl} \): \( \text{joinable } (R^*) = (R \cup R^{-1})^* \)

**Proof**

\[\text{show } \text{joinable } (R^*) \subseteq (R \cup R^{-1})^* \text{ using } \text{joinable-le-rtsc}\ell \ . \]

**Next**

\[\text{show } \text{joinable } (R^*) \supseteq (R \cup R^{-1})^* \text{ proof (rule subrelI)} \]

\[\text{fix } x \ y \]

\[\text{assume } (x, y) \in (R \cup R^{-1})^* \]

\[\text{thus } (x, y) \in \text{joinable } (R^*) \text{ proof (induction set: rtrancl)} \]

\[\text{case base} \]

\[\text{show } (x, x) \in \text{joinable } (R^*) \text{ using } \text{joinable-refl refl-rtrancl} \ . \]

**Next**

\[\text{case } (\text{step } y \ z) \]

\[\text{have } R \subseteq \text{joinable } (R^*) \text{ using refl-le-joinable refl-rtrancl by fast} \]

\[\text{with } (y, z) \in R \cup R^{-1}, \text{ have } (y, z) \in \text{joinable } (R^*) \text{ using } \text{joinable-sym by fast} \]

\[\text{with } (x, y) \in \text{joinable } (R^*) \text{ show } (x, z) \in \text{joinable } (R^*) \]

\[\text{using trans-joinable trans-rtrancl } \text{confluent } (R^*) \text{ by (blast dest: transD)} \]

**Qed**

**Qed**

5.1.4 Predicate version

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

**Where** \( \text{joinablep } P \ x \ y \iff (\exists z. P \ x \ z \land P \ y \ z) \)
lemma joinablep-joinable[pred-set-conv]:
joinablep (λx y. (x, y) ∈ R) = (λx y. (x, y) ∈ joinable R)
by (fastforce simp only: joinablep_def joinable-simp)

lemma reflp-joinablep: reflp P ⟹ reflp (joinablep P)
by (blast intro: reflpI joinable-refl[to-pred] refl-onI[to-pred] dest: reflpD)

lemma joinablep-refl: reflp P ⟹ joinablep P x x
using reflp-joinablep by (rule reflpD)

lemma reflp-le-joinablep: reflp P ⟹ P ≤ joinablep P
by (blast intro: refl-le-joinable[to-pred] refl-onI[to-pred] dest: reflpD))

end

5.2 Combined beta and eta reduction of lambda terms

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

5.2.1 Auxiliary lemmas

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

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

5.2.2 Reduction

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

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

lemma beta-into-beta-eta-reds: s →_β t ⟹ s →_βη∗ t
by auto

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

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

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

lemma beta-eta-appL[intro]: \( s \to_{\beta\eta}^* s' \implies s \circ t \to_{\beta\eta}^* s' \circ t \)
by (induction set: rtranclp) (auto intro: rtranclp.\( rtrancl\)-into-rtrancl)

lemma beta-eta-appR[intro]: \( t \to_{\beta\eta}^* t' \implies s \circ t \to_{\beta\eta}^* s \circ t' \)
by (induction set: rtranclp) (auto intro: rtranclp.\( rtrancl\)-into-rtrancl)

lemma beta-eta-abs[intro]: \( t \to_{\beta\eta}^* t' \implies \Abs t \to_{\beta\eta}^* \Abs t' \)
by (induction set: rtranclp) (auto intro: rtranclp.\( rtrancl\)-into-rtrancl)

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

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

5.2.3 Equivalence

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

definition term-equiv :: dB \( \Rightarrow \) dB \( \Rightarrow \) bool (infixl \( \leftrightarrow \) 50)
where term-equiv = joinablep beta-eta-reds

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

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

43
lemma reds-into-equiv[elim]: \( s \rightarrow_{\eta} t \implies s \leftrightarrow t \)
by (blast intro: term-eqI)

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

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

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

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

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

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

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

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

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

lemma equiv-app: \( s \leftrightarrow t \implies t \leftrightarrow t' \implies s \circ t \leftrightarrow s' \circ t' \)
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: \( s \leftrightarrow t \implies \text{lift } s \leftrightarrow \text{lift } t \)
by (auto intro: term-equivI beta-eta-lift elim: term-equivE)

lemma equiv-liftn: \( s \leftrightarrow t \implies \text{liftn } n \, s \leftrightarrow \text{liftn } n \, t \)
by (induction n) (auto intro: equiv-lift)

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

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

5.3 Combinators defined as closed lambda terms

theory Combinators
imports Beta-Eta
begin

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

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

lemma I-equiv[iff]: \( I ° x \leftrightarrow x \)
using I-eval ..

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

lemma B-eval1: \( B ° g \rightarrow_\beta \text{Abs } (\text{Abs } (\text{lift } g \, 0) ° (\text{Var 1} ° \text{Var 0})) \)
proof
  have \( B ° g \rightarrow_\beta \text{Abs } (\text{Abs } (\text{Var 2} ° (\text{Var 1} ° \text{Var 0}))) [g/0] \) unfolding B-def ..
  then show ?thesis by (simp add: numerals)
qed

lemma B-eval2: \( B ° g ° f \rightarrow_\beta \text{Abs } (\text{lift } g \, 0 ° (\text{lift } f \, 0 ° \text{Var 0}))) ° f \)
proof
  have \( B ° g ° f \rightarrow_\beta \text{Abs } (\text{Abs } (\text{lift } g \, 0) ° (\text{Var 1} ° \text{Var 0}))) ° \text{lift } f \)
  using B-eval1 by blast

45
also have $\rightarrow_\beta Abs (\text{lift} \ g \ 0 \ ° \ (\text{Var} \ 1 \ ° \ \text{Var} \ 0)) [f/0]$ ..
also have $\ldots = Abs (\text{lift} \ g \ 0 \ ° \ (\text{lift} \ f \ 0 \ ° \ \text{Var} \ 0))$ by simp
finally show $?thesis$.
qed

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

lemma $B\text{-equiv}[iff]$:
$B \ ° \ g \ ° \ f \ ° \ x \leftrightarrow g \ ° \ (f \ ° \ x)$
using $B\text{-eval}$ ..

lemma $B\text{-closed}[simp]$:
$lift n B \ k = B$
unfolding $B\text{-def}$ by simp

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

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

lemma $T\text{-equiv}[iff]$:
$T \ ° \ x \ ° \ f \leftrightarrow f \ ° \ x$
using $T\text{-eval}$ ..

lemma $T\text{-closed}[simp]$:
$lift n T \ k = T$
unfolding $T\text{-def}$ 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 arbi-
trary 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 (∀y''. x = Pure x'' → (∃y'. y = Pure y' ∧ x'' ↔ y')) ∧
\[(\forall x'', y = Pure x'' \rightarrow (\exists y'. x = Pure y' \land x'' \leftrightarrow y'))\]

**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**

**qed**

**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 \(x \equiv y\) unfolding x-def.

then have \((\forall x''. x = Opaque x'' \rightarrow (\exists y'. y = Opaque y' \land x'' = y'))\) ∧ \((\forall x''. y = Opaque x'' \rightarrow (\exists y'. x = Opaque y' \land x'' = y'))\)

by induction fast+

with **that show thesis unfolding x-def by blast**

**qed**

**lemma ap-similarE:**

assumes \(x1 \odot x2 \equiv y\)

obtains **y1 y2 where y = y1 \odot y2 and x1 \equiv y1 and x2 \equiv y2**

**proof**

from **assms**

have \((\forall x1', x2'. x1 \odot x2 = x1' \odot x2' \rightarrow (\exists y1 y2. y = y1 \odot y2 \land x1' \equiv y1 \land x2' \equiv y2))\) ∧ \((\forall x1' x2'. x = x1' \odot x2' \rightarrow (\exists y1 y2. x1 \odot x2 = y1 \odot y2 \land x1' \equiv y1 \land x2' \equiv y2))\)

**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 I ◦ 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 $x \simeq y \equiv \text{itrm-cong idiom-rule } x y$

**Lemma** idiom-rule-into-equiv: idiom-rule $x y \implies x \simeq y$.

**Lemmas**
- $\text{itrm-id} = \text{idiom-id}[\text{THEN idiom-rule-into-equiv}]
- \text{itrm-comp} = \text{idiom-comp}[\text{THEN idiom-rule-into-equiv}]
- \text{itrm.hom} = \text{idiom-hom}[\text{THEN idiom-rule-into-equiv}]
- \text{itrm-xchng} = \text{idiom-xchng}[\text{THEN idiom-rule-into-equiv}]

**Lemma** similar-into-equiv: $x \bowtie y \implies x \simeq y$

by (induction pred: idrm-cong) (auto intro: ap-cong idrm-sym idrm-trans)

**Lemma** opaque-equiv: $x \simeq y \implies \text{opaque } x = \text{opaque } y$

**Proof** (induction pred: idrm-cong)
  - case (into-idrm-cong $x y$)
  - thus ?case by induction auto

**Qed** simp-all

**Lemma** iorder-equiv: $x \simeq y \implies \text{iorder } x = \text{iorder } y$

by (auto dest: opaque-equiv)

**Locale** special-idiom =
  - fixes extra-rule :: $'a$ itrm $\Rightarrow 'a$ itrm $\Rightarrow \text{bool}$
begin

**Definition** idiom-ext-rule = sup idiom-rule extra-rule

**Abbreviation**
- $\text{itrn-ext-equiv} :: 'a$ itrm $\Rightarrow 'a$ itrm $\Rightarrow \text{bool}$ (infixl $\simeq+$ 50)

**Where** $x \simeq y \equiv \text{itrn-cong idiom-ext-rule } x y$

**Lemma** equiv-into-ext-equiv: $x \simeq y \implies x \simeq y$

**Unfolding** idiom-ext-rule-def
by (induction pred: idrm-cong)
  - (auto intro: into-idrm-cong ap-cong idrm-sym idrm-trans)

**Lemmas**
- $\text{itrn-ext-id} = \text{itrn-id}[\text{THEN equiv-into-ext-equiv}]
- \text{itrn-ext-comp} = \text{itrn-comp}[\text{THEN equiv-into-ext-equiv}]
- \text{itrn-ext-hom} = \text{itrn-hom}[\text{THEN equiv-into-ext-equiv}]
- \text{itrn-ext-xchng} = \text{itrn-xchng}[\text{THEN equiv-into-ext-equiv}]

end

### 5.4.2 Syntactic unlifting

With generalisation of variables

**Primrec**

unlift' :: $\text{nat} \Rightarrow 'a$ itrm $\Rightarrow \text{nat}$

$\Rightarrow dB$

**Where**

unlift' $n$ (Opaque -) $i$ = Var $i$

| unlift' $n$ (Pure $x$) $i$ = liftn $n$ $x$ $0$
unlift' \ n \ (f \circ x) \ i = \text{unlift'} \ n \ f \ (i + \text{iorder} \ x) \circ \text{unlift'} \ n \ x \ i

abbreviation \text{unlift} \ x \equiv (\text{Abs} \ ^{\text{iorder}} \ x) \ (\text{unlift'} \ (\text{iorder} \ x) \ x \ 0)

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

lemma \text{absn-cong[intro]}: s \leftrightarrow t \Longrightarrow (\text{Abs} \ ^n) \ s \leftrightarrow (\text{Abs} \ ^n) \ t
by (induction n) auto

lemma \text{free-unlift}: \text{free} \ (\text{unlift'} \ n \ x \ i) \ j \Longrightarrow j \geq n \lor (j \geq i \land j < i + \text{iorder} \ x)
proof (induction x arbitrary: i)
  case (Opaque x)
  thus ?case by simp
next
  case (Pure x)
  thus ?case using \text{free-liftn} by simp
next
  case (IAp x y)
  hence \ j \leq i + \text{iorder} \ y \ by simp
with IAp show ?case by auto
qed

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

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

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

\textbf{Preserving variables} \quad \textbf{primrec} \text{unlift-vars :: nat \Rightarrow nat \ itrm \Rightarrow dB}
where \( \text{unlift-vars n (Opaque i) = Var i} \)
unlift-vars n (Pure x) = liftn n x 0
unlift-vars n (x ⋄ y) = unl lift-vars n x ° unl lift-vars n y

lemma all-pure-unlift-vars: opaque x = [] ⇒ x ∼ Pure (unlift-vars 0 x)
proof (induction x)
  case (Opaque x) then show ?case by simp
next
  case (Pure x) then show ?case by simp
next
  case (IAp x y) then have no-opaque: opaque x = [] opaque y = [] by simp+
  then have unl lift-ap: unl lift-vars 0 (x ⋄ y) = unl lift-vars 0 x ° unl lift-vars 0 y
  by simp
  from no-opaque IAp.IH have x ⋄ y ∼ Pure (unlift-vars 0 x) ⋄ Pure (unlift-vars 0 y)
  by (blast intro: ap-cong)
  also have ... ∼ Pure (unlift-vars 0 x ° unl lift-vars 0 y) by (rule itrm-hom)
  also have ... = Pure (unlift-vars 0 (x ⋄ y)) by (simp only: unl lift-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 ↔ unl lift 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 \((f \odot \text{Opaque } x)\) = \(\text{Abs}^{\sim}?n\) \((\text{unlift}' \ ?n f \ 1 \circ \text{Var } 0)\)
  by simp
also have ... = \(\text{Abs}^{\sim}\)\(iorder f\) \((\text{Abs} (\text{unlift}' \ ?n f 1 \circ \text{Var } 0))\)
  using \text{unlift-Suc-inside} by simp
also have ... \(\leftrightarrow\) \(\text{unlift} f\)
proof
  have \(\neg\) \(\text{free} (\text{unlift}' \ ?n f 1)\) \(0\)
  using \text{free-unlift} by fastforce
  hence \(\text{Abs} (\text{unlift}' \ ?n f 1 \circ \text{Var } 0) \rightarrow (\text{unlift}' \ ?n f 1)[\text{Var } 0/0]\) ..
  also have ... = \(\text{unlift}' (\text{iorder f}) f 0\)
  using \text{unlift-subst} by (metis \text{One-nat-def} Suc-eq-plus1 le0)
finally show \(?thesis\)
  by (simp add: \(r\)-into-\(r\)-tranclp absn-cong eta-into-equiv)
qed
finally show \(?case\)
  using ap-cf.\text{IH} by (auto intro: term-sym term-trans)
qed

lemma \text{cf-similarI}:
  assumes \(x \in \text{CF} \ y \in \text{CF}\)
  and \(\text{opaque } x = \text{opaque } y\)
  and \(\text{CF-pure } x \leftrightarrow \text{CF-pure } y\)
  shows \(x \equiv y\)
using \text{assms} proof (induction arbitrary: \(y\))
case \(\text{pure-cf } x\)
hence \(\text{opaque } y = [\] by auto
with \(y \in \text{CF}\) obtain \(y'\) where \(y = \text{Pure } y'\)
by cases auto
with \text{pure-cf.prems} show \(?case\) by auto
next
case \(\text{ap-cf } f \ x\)
from \(\langle \text{opaque } (f \odot \text{Opaque } x) = \text{opaque } y\rangle\)
obtain \(y1 \ y2\) where \(\text{opaque } y = y1 @ y2\)
  and \(\text{opaque } f = y1\) and \(\text{[x]} = y2\)
  by fastforce
from \(\langle [x] = y2\rangle\) obtain \(y'\) where \(y2 = [y']\) and \(x = y'\)
  by auto
with \(y \in \text{CF}\) and \(\langle \text{opaque } y = y1 @ y2\rangle\) obtain \(g\)
  where \(\text{opaque } g = y1\) and \(\text{y-split: } y = g \odot \text{Opaque } y' \ g \in \text{CF}\)
by cases auto
with \text{ap-cf.prems} \(\text{opaque } f = y1\)
have \(\text{opaque } f = \text{opaque } g\) \(\text{CF-pure } f \leftrightarrow \text{CF-pure } g\)
by auto
with \text{ap-cf.IH} \(\langle g \in \text{CF}\rangle\) have \(f \equiv g\)
by simp
with \text{ap-cf.prems} \(\text{y-split } \langle x = y'\rangle\) show \(?case\)
by (auto intro: ap-cong)
qed

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

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

**Lemma cf-unique:**
assumes in-cf: \( x \in CF \) \( y \in CF \)
and equiv: \( x \simeq y \)
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** \( \text{norm-pn} :: \ dB \Rightarrow \ 'a \ itrm \Rightarrow \ 'a \ itrm \)
where
\[
\begin{align*}
\text{norm-pn} f \ (\text{Opaque} \ x) &= \text{undefined} \\
\text{norm-pn} f \ (\text{Pure} \ x) &= \text{Pure} \ (f \circ x) \\
\text{norm-pn} f \ (n \circ x) &= \text{norm-pn} \ (\text{B} \circ f) \ n \circ x
\end{align*}
\]

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

**Primrec** \( \text{norm} :: \ 'a \ itrm \Rightarrow \ 'a \ itrm \)
where
\[
\begin{align*}
\text{norm} \ (\text{Opaque} \ x) &= \text{Pure} \ I \circ \text{Opaque} \ x \\
\text{norm} \ (\text{Pure} \ x) &= \text{Pure} \ x \\
\text{norm} \ (f \circ x) &= \text{norm-nn} \ (\text{norm} \ f) \ (\text{norm} \ x)
\end{align*}
\]

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

**Lemma** \( \text{norm-nn-in-cf} :: \)
assumes \( n \in CF \) \( n' \in CF \)
shows \( \text{norm-nn} n \ n' \in CF \)
using assms(2,1)
by (induction \( n' \) arbitrary; \( x \)) (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 \( x \in \text{CF} \)
shows \( \text{norm-pn } f \cdot x \simeq \text{Pure } f \cdot x \)
using assms proof (induction \( x \) arbitrary: \( f \))
case (pure-cf \( x \))
  have \( \text{Pure } (f \circ x) \simeq \text{Pure } f \cdot \text{Pure } x \) using \( \text{itrn-hom[symmetric]} \).
then show \?case by simp
next
case (ap-cf \( n \cdot x \))
from ap-cf.IH have \( \text{norm-pn } (B \circ f) \cdot n \simeq \text{Pure } (B \circ f) \cdot n \).
then have \( \text{norm-pn } (B \circ f) \cdot n \circ \text{Opaque } x \simeq \text{Pure } (B \circ f) \cdot n \circ \text{Opaque } x \).
also have \( \ldots \simeq \text{Pure } B \circ \text{Pure } f \circ n \circ \text{Opaque } x \)
  using \( \text{itrn-hom[symmetric]} \) by blast.
also have \( \ldots \simeq \text{Pure } f \circ (n \circ \text{Opaque } x) \) using \( \text{itrn-comp} \).
finally show \?case by simp
qed

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

55
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 \(\text{norm-nn} \, (\text{norm} \, f) \, (\text{norm} \, x) \simeq \text{norm} \, f \, \circ \, \text{norm} \, x\)
by (rule norm-nn-equiv)
also have \(\ldots \simeq f \, \circ \, x\) using IAp.IH 
finally show \(?case\) by simp
qed

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

5.4.5 Lifting with normal forms

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

theorem \texttt{nf-lifting}:
assumes \(\text{opaque}: \text{opaque} \, x = \text{opaque} \, y\)
and \(\text{base-eq}: \text{unlift} \, x \leftrightarrow \text{unlift} \, y\)
s Worms \(x \simeq y\)
proof –
obtain \(n\) where \(nf-x: n \simeq x\) \(n \in CF\) by (rule \texttt{normal-form})
obtain \(n'\) where \(nf-y: n' \simeq y\) \(n' \in CF\) by (rule \texttt{normal-form})
from \(nf-x\) have \(CF\)-pure \(n \leftrightarrow \text{unlift} \, x\) by (rule \texttt{nf-unlift})
also note \(base-eq\)
also from \(nf-y\) have \(\text{unlift} \, y \leftrightarrow CF\)-pure \(n'\) by (rule \texttt{nf-unlift}[THEN \texttt{term-sym}])
finally have \(\text{pure-eq}: CF\)-pure \(n \leftrightarrow CF\)-pure \(n'\).
from \(nf-x(1)\) have \(\text{opaque} \, n = \text{opaque} \, x\) by (rule \texttt{opaque-equiv})
also note \(opaque\)
also from \(nf-y(1)\) have \(\text{opaque} \, y = \text{opaque} \, n'\) by (rule \texttt{opaque-equiv}[THEN \texttt{sym}])
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 \(nf-x\) \(nf-y\) \(\text{pure-eq}\) \(\text{opaque-eq}\)
by (blast intro: \texttt{similar-into-equiv cf-similarI})
also from $\text{nf-y}(1)$ have $n' \simeq y$.
finally show $x \simeq y$.
qed

5.4.6 Bracket abstraction, twice

Preliminaries: Sequential application of variables
definition $\text{frees} :: dB \Rightarrow \text{nat set}$
where $\text{simp}: \text{frees } t = \{ i. \text{ free } i \}$
definition $\text{var-dist} :: \text{nat list } \Rightarrow dB \Rightarrow dB$
where $\text{var-dist} = \text{fold} (\lambda i t. t \circ \text{Var } i)$

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

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

lemma $\text{var-dist-append1} [\text{simp}]: \text{var-dist vs } @ [v] t = \text{var-dist vs } t \circ \text{Var } v$
unfolding $\text{var-dist-def}$ 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: $t$) auto

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

lemma $\text{var-dist-subst-gt}:
\forall v \in \text{set vs}. v < i \Rightarrow (\text{var-dist vs } s)[t/i] = \text{var-dist } (s[t/i])$
by (induction vs arbitrary: $s$) 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 $\text{vsubst-subst} [\text{simp}]: (\text{Var } u)[\text{Var } v/w] = \text{Var } (\text{vsubst } u v w)$
unfolding $\text{vsubst-def}$ by simp

lemma $\text{vsubst-subst-lt} [\text{simp}]: u < w \Rightarrow \text{vsubst } u v w = u$
unfolding $\text{vsubst-def}$ by simp

lemma $\text{var-dist-subst-Var}:
(\text{var-dist vs } s)[\text{Var } i/j] = \text{var-dist } (\text{map} (\lambda v. \text{vsubst } v i j) vs) (s[\text{Var } i/j])$
by (induction vs arbitrary: $s$) 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: $s \leftrightarrow t$) auto

57
Preliminaries: Eta reductions with permuted variables  

lemma absn-subst: 

\((\text{Abs} \, \tilde{n}) \, s)[t/k] = (\text{Abs} \, \tilde{n}) \, (s[liftn \, n \, t \, 0/k+i\, n])\)

by (induction \(n\) arbitrary: \(t \, k\)) (simp-all add: liftn-lift-swap)

lemma absn-beta-equiv: \((\text{Abs} \, \tilde{n}) \, s \, \circ \, t \iff (\text{Abs} \, \tilde{n}) \, (s[liftn \, n \, t \, 0/n])\)

proof –
  have \((\text{Abs} \, \tilde{n}) \, s \, \circ \, t = \text{Abs} \, ((\text{Abs} \, \tilde{n}) \, s) \, \circ \, t\) by simp
  also have \(\ldots \iff ((\text{Abs} \, \tilde{n}) \, s)[t/0]\) by (rule beta-into-equiv) (rule beta beta)
  also have \(\ldots = (\text{Abs} \, \tilde{n}) \, (s[liftn \, n \, t \, 0/n])\) by (simp add: absn-subst)
  finally show \(?\text{thesis}\).

qed

lemma absn-dist-eta: \((\text{Abs} \, \tilde{n}) \, (\text{var-dist} \, (\text{rev} \, [0..<n])) \, (\text{liftn} \, n \, t \, 0)) \iff t\)

proof (induction \(n\))
  case 0 show \(?\text{case} \, \text{by simp}\)

next
  case (Suc \(n\))
  let \(?\text{dist-range} = \lambda a \, k. \, \text{var-dist} \, (\text{rev} \, [a..<k]) \, (\text{liftn} \, k \, t \, 0)\)
  have append: \(\text{rev} \, [0..<n] = \text{rev} \, [1..<\text{Suc} \, n] \, @ \, [0]\) by (simp add: upt-rec)
  have dist-last: \(\text{dist-range} \, 0 \, (\text{Suc} \, n) = \text{dist-range} \, 1 \, (\text{Suc} \, n) \, \circ \, \text{Var} \, 0\)

  unfolding append var-dist-append1 ..

  have \(\neg \, \text{free} \, (\text{dist-range} \, 1 \, (\text{Suc} \, n)) \, \emptyset\) proof –
    have \(\text{frees} \, (\text{dist-range} \, 1 \, (\text{Suc} \, n)) = \text{frees} \, (\text{liftn} \, (\text{Suc} \, n) \, t \, 0) \cup \{1..\}\)

    unfolding var-dist-frees by fastforce
  then have \(0 \not\in \, \text{frees} \, (\text{dist-range} \, 1 \, (\text{Suc} \, n))\) by simp
  then show \(?\text{thesis} \, \text{by simp}\)

qed

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

  unfolding dist-last by (rule eta)
  also have \(\ldots = \text{var-dist} \, (\text{rev} \, [0..<n]) \, ((\text{liftn} \, (\text{Suc} \, n) \, t \, 0)[\text{Var} \, 0/0])\) proof –
    have \(\forall \, v \in \text{set} \, (\text{rev} \, [1..<\text{Suc} \, n]). \, 0 < v\) by auto
    moreover have \(\text{rev} \, [0..<n] = \text{map} \, (\lambda v. \, v - 1) \, (\text{rev} \, [1..<\text{Suc} \, n])\) by (induction \(n\)) simp-all
    ultimately show \(?\text{thesis}\) by (simp only: var-dist-subst-It)

qed

then have \(\text{Abs} \, (\text{dist-range} \, 0 \, n) \, \text{using subst-liftn}[\text{of} \, 0 \, n \, 0 \, \text{t} \, \text{Var} \, 0] \) by simp

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

then have \((\text{Abs} \, \text{Suc} \, n) \, (\text{dist-range} \, 0 \, (\text{Suc} \, n)) \iff (\text{Abs} \, \text{Suc} \, n) \, (\text{dist-range} \, 0 \, n)\)

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

also from \(\text{Suc.} \text{IH}\) have \(\ldots \, \Rightarrow t\).

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

qed

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

where

\[\text{strip-context} \, n \, (\text{Var} \, i) \, k = \, \text{if} \, i \, < \, k \, \text{then} \, \text{Var} \, i \, \text{else} \, \text{Var} \, (i - n)\]
| \(\text{strip-context} \, n \, (\text{Abs} \, t) \, k = \text{Abs} \, (\text{strip-context} \, n \, t \, (\text{Suc} \, k))\)
| \(\text{strip-context} \, n \, (s \, \circ \, t) \, k = \, \text{strip-context} \, n \, (s \, \circ \, t) \, k \, \circ \, \text{strip-context} \, n \, t \, k\)
lemma strip-context-liftn: strip-context n (liftn (m + n) t k) k = liftn m t k
by (induction t arbitrary: k) simp-all

lemma liftn-strip-context:
  assumes \( \forall i \in \text{frees } t. \ i < k \lor k + n \leq i \)
  shows liftn n (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 ?thesis
        by arith
    qed simp
  qed
  with Abs.IH show ?case
    by simp
qed auto

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

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

lemma perm-vars-distinct:
  perm-vars n vs \( \leftrightarrow \) distinct vs
unfolding perm-vars-def by simp

lemma perm-vars-length:
  perm-vars n vs \( \leftrightarrow \) length vs = n
unfolding perm-vars-def using distinct-card by force

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

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

lemma perm-vars-inj-on-nth:
  assumes perm-vars n vs
  shows inj-on (nth vs) \{0..<n\}
proof (rule inj-onI)
  fix i j
  assume i ∈ {0..<n} and j ∈ {0..<n}
  with assms have i < length vs and j < length vs
    using perm-vars-length by simp+
  moreover from assms have distinct vs by (rule perm-vars-distinct)
  moreover assume vs ! i = vs ! j
  ultimately show i = j using 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 \( \forall t. \forall i ∈ \text{frees } t. \text{n} \leq i \implies \) 
  \( \text{(Abs}^{\text{n}}) \text{(var-dist } vs’ \text{)((Abs}^{\text{n}}) \text{(var-dist vs (liftn n t 0)))} \leftrightarrow \text{strip-context n t 0} \)
  proof –
  define vsubsts where vsubsts n vs ′ vs = 
    map λv.
      if v < n − length vs ′ then v
      else if v < n then vs ′ ! ((n − v − 1) + (n − length vs ′))
      else v − length vs ′ for n vs ′ vs

  let ?app-vars = λt n vs ′ vs. var-dist vs ′ ((Abs}^{\text{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}^{\text{n}}) (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: \( m' = n - \text{length}(v \# vs') \) 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 \( \leftrightarrow \) var-dist vs' ((Abs``Suc n') (var-dist vs (liftn (Suc n') t 0)))
  \( \text{Var}\ v)\)
unfolding var-dist-Cons ..
also have \( \leftrightarrow ?app-vars\ t n' vs' (vsubsts\ n\ [v] vs)\) proof (rule var-dist-cong)
  have map (\( \lambda\ v.\ vsubsts\ v (v + n') n' \) vs = vsubsts\ n\ [v] vs)
  unfolding Suc-n'[symmetric] vssubsts-def vsubsts-def
  by (auto cong: if-cong)
then have \( (\text{var-dist vs (liftn (Suc n') t 0))}(\text{liftn n' (Var v) 0/n'})
  = \text{var-dist (vsubsts\ n\ [v] vs) (liftn n' t 0)}\)
using var-dist-subst-Var subst-liftn by simp
then show \( (\text{Abs``Suc n'}) (\text{var-dist vs (liftn (Suc n') t 0))} \leftrightarrow \) Var v
  \( (\text{Abs``n'}) (\text{var-dist (vsubsts\ n\ [v] vs) (liftn n' t 0)})\)
by (fastforce intro: absn-beta-equiv[THEN term-trans])
qed
also have \( \leftrightarrow (\text{Abs``?m'}) (\text{var-dist (vsubsts\ n' vs' (vsubsts\ n\ [v] vs)}) (\text{liftn ?m' t 0}))\)
  using vs'-length Cons.IH by blast
also have \( = (\text{Abs``?m'}) (\text{var-dist (vsubsts\ n\ (v \# vs') vs) (liftn ?m' t 0)})\)
proof
  have vsubsts\ n' vs' (vsubsts\ (Suc n') [v] vs) = vsubsts\ (Suc n') (v \# vs') vs
  unfolding vsubsts-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 (\( \lambda_i.\ n - \text{perm-vars-inv}\ n\ vs\ (n - i - 1) - 1\))
[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 (\( \lambda v'\ vs'\ ! (n - v - 1) = v'\) 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

61
unfolding \( vs'\)-def by simp
also from \( \langle i < n \rangle \) have ... = \( n - i - 1 \) using perm-vars perm-vars-inv-nth

by simp
also from \( \langle i < n \rangle \) 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: vsubsts n vs' vs = rev [0..<n]
unfolding vsubsts-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 \( \?app-vars t n \) \( \leftrightarrow \) \( \?appd-vars t n \) for \( t \)
using partial-appd[of \( vs' \)] vs'-length vs'-vs by simp
then have \( (Abs^{\sim}n) \) \( \langle \?app-vars t n \rangle \) \( \leftrightarrow \) \( (Abs^{\sim}n) \) \( \langle \?appd-vars t n \rangle \)
by (rule absn-cong)
also have ... \( \leftrightarrow \) strip-context \( n \) \( \) \( \) \( t \) 0
using not-free by (rule absn-dist-eta-free)
finally have \( (Abs^{\sim}n) \) \( \langle \?app-vars t n \rangle \) \( \leftrightarrow \) strip-context \( n \) \( \) \( t \) 0).
\}
with that show \?thesis.

qed

lemma liftn-absn: \( \text{liftn} n \) \( \langle \langle Abs^{\sim}m \rangle \rangle \) \( t \) \( k \) = \( \langle Abs^{\sim}m \rangle \) \( \langle \text{liftn} n \rangle \) \( t \) \( (k + m) \)
by (induction \( m \) arbitrary; \( k \)) auto

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

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

qed
then have \( \forall i. \ 0 < i \land \text{free } t \ i \rightarrow i - 1 < k \lor k' \leq i - 1 \) by simp
then have \( \forall i. \ 0 < i \land \text{free } t \ i \rightarrow i < \text{Suc } k \lor \text{Suc } k' \leq i \) by auto
with \( i \neq 0 \land i \in \text{frees } t \) show \( \text{thesis} \) by simp
qed
qed
with \( \text{Abs}, \text{IH} \) Abs.prems(1) show \( \text{case} \) 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: \( \text{perm-vars } n \ vs \)
and not-free: \( \forall i \in \text{frees } s. \ n \leq i \land \forall i \in \text{frees } t. \ n \leq i \)
shows \( \text{strip-context } n \ s \ 0 \leftrightarrow \text{strip-context } n \ t \ 0 \)
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:
\( \forall t. \ \forall i \in \text{frees } t. \ n \leq i \implies (\text{Abs} \sim n) (\text{var-dist } vs' (((\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ t \ 0)))))) \leftrightarrow \text{strip-context } n \ t \ 0 \)
using perm-vars dist-perm-eta by blast
have \( \text{strip-context } n \ s \ 0 \leftrightarrow (\text{Abs} \sim n) (\text{var-dist } vs' (((\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ s \ 0)))))) \)
using etas[THEN term-sym] not-free(1)
also have \( \ldots \leftrightarrow (\text{Abs} \sim n) (\text{var-dist } vs' (((\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ s \ 0)))))) \)
proof (rule absn-cong, rule var-dist-cong)
have \( (\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ s \ 0)) = (\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ s \ n)) \)
using not-free(1) liftn-liftn0[of \( s \ n \)] by simp
also have \( \ldots = (\text{Abs} \sim n) (\text{liftn } n \ (\text{var-dist } vs \ s) \ n) \)
using vs-lt-n liftn-var-dist-lt by simp
also have \( \ldots = (\text{liftn } n \ ((\text{Abs} \sim n) (\text{var-dist } vs \ s)) \ 0) \)
using liftn-absn by simp
also have \( \ldots \leftrightarrow \text{liftn } n \ ((\text{Abs} \sim n) (\text{var-dist } vs \ t) \ 0) \)
using perm-equiv by (rule equiv-liftn)
also have \( \ldots = (\text{Abs} \sim n) (\text{liftn } n \ (\text{var-dist } vs \ t) \ n) \)
using liftn-absn by simp
also have \( \ldots = (\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ t \ n)) \)
using vs-lt-n liftn-var-dist-lt by simp
also have \( \ldots = (\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ t \ 0)) \)
using not-free(2) liftn-liftn0[of \( t \ n \)] by simp
finally show \( (\text{Abs} \sim n) (\text{var-dist } vs (\text{liftn } n \ s \ 0)) \leftrightarrow \ldots \).
qed
also have \( \ldots \leftrightarrow \text{strip-context } n \ t \ 0 \)
using etas not-free(2).
finally show \( \text{thesis} \).
 qed
General notion of bracket abstraction for lambda terms

definition foldr-option :: ('a ⇒ 'b ⇒ 'b option) ⇒ 'a list ⇒ 'b ⇒ 'b option
where foldr-option f xs e = foldr (λa b. Option.bind b (f a)) xs (Some e)

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

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

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

locale bracket-abstraction =
  fixes term-bracket :: nat ⇒ dB ⇒ dB option
  assumes bracket-app: term-bracket i s = Some s' ⇒ s ° Var i ↔ s
  assumes bracket-frees: term-bracket i s = Some s' ⇒ frees s' = frees s - {i}
begin

definition term-brackets :: nat list ⇒ dB ⇒ 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 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-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 var-dist vs t' ↔ t
proof -
  from assms have ∀ t''. t' ↔ t'' → var-dist vs t'' ↔ t
  proof (induction vs arbitrary: t')
    case Nil then show ?case by (simp add: term-sym)
  next

64
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'' ° Var v ↔ u by (rule bracket-app)
show ?case proof rule+
  fix t'' assume t' ↔ t''
with red1 have red: t'' ° Var v ↔ u
using term-sym term-trans by blast
have var-dist (v ≠ vs) t'' = var-dist vs (t'' ° Var v) by simp
also have ... ↔ t using Cons.IH[OF inner] red[symmetric] by blast
finally show var-dist (v ≠ vs) t'' ↔ t.
qed

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

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

lemma unlift-vars-frees: ∀ i ∈ frees (unlift-vars n x). i ∈ set (opaque x) ∨ n ≤ i
by (induction x) (auto simp add: free-liftn)

locale itrm-abstraction = special-idiom extra-rule for extra-rule :: nat itrm ⇒ - +
fixes itrm-bracket :: nat ⇒ nat itrm ⇒ nat itrm option
assumes itrm-bracket-ap: itrm-bracket i x = Some x' ⇒ x' ⊑ Opaque i ⊑+ x
assumes itrm-bracket-opaque:
  itrm-bracket i x = Some x' ⇒ set (opaque x') = set (opaque x) - {i}
begin

definition itrm-brackets = foldr-option itrm-bracket

lemma itrm-brackets-Nil[simp]: itrm-brackets [] x = Some x
unfolding itrm-brackets-def by simp

lemma itrm-brackets-Cons-SomeE:
  assumes itrm-brackets (v#vs) x = Some x'
  obtains y' where itrm-brackets vs x = Some y' and itrm-bracket v y' = Some x'
using assms unfolding itrm-brackets-def by (elim foldr-option-Cons-SomeE)
definition opaque-dist = fold (\(\lambda i y. y \diamond \text{Opaque } i\))

lemma opaque-dist-cong: \(x \simeq^+ y \Rightarrow \text{opaque-dist } x \simeq^+ \text{opaque-dist } y\)
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 \text{Opaque } v \simeq^+ y' by (elim itrm-bracket-ap)
  then have x'' \circ \text{Opaque } v \simeq^+ y'
    using Cons.prems(2) by (blast intro: itrm-sym itrm-trans)
  note this[symmetric]
  with defined' have fold (\(\lambda i y. y \circ \text{Opaque } i\)) vs (x'' \circ \text{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'

66
shows opaque $x' = []$

proof
from defined have set (opaque $x'$) = set (opaque $x$) − set vs
by (rule itrm-brackets-opaque)
with all-opaque have set (opaque $x'$) = {} by simp
then show ?thesis by simp
qed

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

end

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

lemma brackets-unlift-vars-swap:
assumes all-opaque: set (opaque $x$) ⊆ {0..<n}
and vs-bound: set vs ⊆ {0..<n}
and defined: itrm-brackets vs $x$ = Some $x'$
shows term-brackets vs (unlift-vars n $x$) = Some (unlift-vars n $x'$)
using vs-bound defined proof (induction vs arbitrary: $x'$)
case Nil
then show ?case by simp
next
case (Cons v vs)
then obtain $y'$
where ivs: itrm-brackets vs $x$ = Some $y'$
and iv: itrm-bracket v $y'$ = Some $x'$
by (elim itrm-brackets-Cons-SomeE)
with Cons have term-brackets vs (unlift-vars n $x$) = Some (unlift-vars n $y'$)
by auto
moreover have
have Some (unlift-vars n $x$) = map-option (unlift-vars n) (itrm-bracket v $y'$)
unfolding iv by simp
moreover have set (opaque $y'$) ⊆ {0..<n}
using all-opaque ivs by (auto dest: itrm-brackets-opaque)
moreover have \( v < n \) using Cons.prems by simp
ultimately have \( \text{term-bracket} \ v \ (\text{unlift-vars} \ n \ y') = \text{Some} \ (\text{unlift-vars} \ n \ x') \)
using bracket-compat by auto
}
ultimately show \(?\text{case}\) by (rule term-brackets-ConsI)
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: \( \text{itrm-brackets} \ vs \ x = \text{Some} \ x' \) \( \text{itrm-brackets} \ vs \ y = \text{Some} \ y' \)
and base-eq: \( (\text{Abs}^{\sim} n) \ (\text{unlift-vars} \ n \ x) \leftrightarrow (\text{Abs}^{\sim} n) \ (\text{unlift-vars} \ n \ y) \)
shows \( x \simeq\) \( y \)
proof –
from perm-vars have set-vs: set vs = \{0..<\( n \)\}
unfolding perm-vars-def by simp
have \( x\text{-swap} \) term-brackets vs (unlift-vars \( n \) \( x \)) = Some (unlift-vars \( n \) \( x' \))
using all-vars set-vs defined(1) by (auto intro: brackets-unlift-vars-swap)
have \( y\text{-swap} \) term-brackets vs (unlift-vars \( n \) \( y \)) = Some (unlift-vars \( n \) \( y' \))
using all-vars set-vs defined(2) by (auto intro: brackets-unlift-vars-swap)

from all-vars have set (opaque \( x \)) \( \subseteq \) set vs unfolding set-vs by simp
then have complete-x: opaque \( x' = [] \)
using defined(1) itrm-brackets-all by blast
then have \( x\text{-frees} \): \( \forall i \in \text{frees} \) (unlift-vars \( n \) \( x' \)), \( n \leq i \)
using unlift-vars-frees by fastforce

from all-vars have set (opaque \( y \)) \( \subseteq \) set vs unfolding set-vs by simp
then have complete-y: opaque \( y' = [] \)
using defined(2) itrm-brackets-all by blast
then have \( y\text{-frees} \): \( \forall i \in \text{frees} \) (unlift-vars \( n \) \( y' \)), \( n \leq i \)
using unlift-vars-frees by fastforce

have \( x \simeq^+ \) opaque-dist vs \( x' \)
using defined(1) by (rule itrm-brackets-dist[\text{symmetric}])
also have \( ... \simeq^+ \) opaque-dist vs (Pure (unlift-vars \( 0 \) \( x' \)))
using all-vars set-vs defined(1)
by (auto intro: opaque-dist-cong itrm-brackets-all-unlift-vars)
also have \( ... \simeq^+ \) opaque-dist vs (Pure (unlift-vars \( 0 \) \( y' \)))
proof (rule opaque-dist-cong, rule pure-cong)
have \( (\text{Abs}^{\sim} n) \ (\text{var-dist} \ vs \ (\text{unlift-vars} \ n \ x')) \leftrightarrow (\text{Abs}^{\sim} n) \ (\text{unlift-vars} \ n \ x) \)
using \( x\text{-swap} \) term-brackets-dist by auto
also have \( \ldots \leftrightarrow (\text{Abs}^{\sim} n) \ (\text{unlift-vars} \ n \ y) \) using base-eq .
also have \( \ldots \leftrightarrow (\text{Abs}^{\sim} n) \ (\text{var-dist} \ vs \ (\text{unlift-vars} \ n \ y')) \)
using \( y\text{-swap} \) term-brackets-dist[THEN term-sym] by auto
finally have \( \text{strip-context} \ n \ (\text{unlift-vars} \ n \ x') \ 0 \leftrightarrow \text{strip-context} \ n \ (\text{unlift-vars} \ n \ y') \ 0 \)
using perm-vars \( x\text{-frees} \) \( y\text{-frees} \)
by (intro dist-perm-eta-equiv)
then show unlift-vars 0 x' ↔ unlift-vars 0 y'
  using strip-unlift-vars complete-x complete-y by simp
qed
also have ... ≃ opaque-dist vs y' proof (rule opaque-dist-cong)
  show Pure (unlift-vars 0 y') ≃+ y'
    using all-vars set-vs defined(2) itrm-brackets-all-unlift-vars[THEN itrm-sym]
    by blast
qed
also have ... ≃+ y using defined(2) by (rule itrm-brackets-dist)
finally show ?thesis .
qed

References


