Applicative Lifting

Andreas Lochbihler Joshua Schneider

October 27, 2022

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

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

1.1 Equality restricted to a set

definition *eq-on* :: $'a\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where [*simp*]: $eq\text{-}on\ A = (\lambda x\ y.\ x \in A \wedge x = y)$

lemma *rel-fun-eq-onI*: $(\bigwedge x.\ x \in A \Longrightarrow R\ (f\ x)\ (g\ x)) \Longrightarrow rel\text{-}fun\ (eq\text{-}on\ A)\ R\ f\ g$
by *auto*

lemma *rel-fun-map-fun2*: $rel\text{-}fun\ (eq\text{-}on\ (range\ h))\ A\ f\ g \Longrightarrow rel\text{-}fun\ (BNF\text{-}Def.Grp\ UNIV\ h)^{-1-1}\ A\ f\ (map\text{-}fun\ h\ id\ g)$
by(*auto simp add*: *rel-fun-def Grp-def eq-onp-def*)

lemma *rel-fun-refl-eq-onp*:
$(\bigwedge z.\ z \in f\ `\ X \Longrightarrow A\ z\ z) \Longrightarrow rel\text{-}fun\ (eq\text{-}on\ X)\ A\ f\ f$
by(*auto simp add*: *rel-fun-def eq-onp-def*)

lemma *eq-onE*: $[\![\ eq\text{-}on\ X\ a\ b;\ [\![\ b \in X;\ a = b\ ]\!] \Longrightarrow thesis\ ]\!] \Longrightarrow thesis$ **by** *auto*

lemma *Domainp-eq-on* [*simp*]: $Domainp\ (eq\text{-}on\ X) = (\lambda x.\ x \in X)$
by *auto*

1.2 Proof automation

lemma *arg1-cong*: $x = y \Longrightarrow f\ x\ z = f\ y\ z$
by (*rule arg-cong*)

lemma *UNIV-E*: $x \in UNIV \Longrightarrow P \Longrightarrow P$ **.**

context begin

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

private definition $B\ g\ f\ x \equiv g\ (f\ x)$
private definition $C\ f\ x\ y \equiv f\ y\ x$
private definition $I\ x \equiv x$
private definition $K\ x\ y \equiv x$
private definition $S\ f\ g\ x \equiv (f\ x)\ (g\ x)$
private definition $T\ x\ f \equiv f\ x$
private definition $W\ f\ x \equiv f\ x\ x$

lemmas [*abs-def*, *combinator-unfold*] = *B-def C-def I-def K-def S-def T-def W-def*
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*))))) *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*

end

2 Common applicative functors

2.1 Environment functor

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

definition *const* $x = (\lambda\text{-}.\ x)$
definition *apf* $x\ y = (\lambda z.\ x\ z\ (y\ z))$

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* $(\lambda f\ x\ y.\ f\ y\ x) \diamond f \diamond x \diamond y = f \diamond y \diamond x$
by *applicative-lifting simp*

end

2.2 Option

theory *Applicative-Option* **imports**
  Applicative
  HOL−Library.Adhoc-Overloading
begin

fun *ap-option* :: $('a \Rightarrow 'b)$ *option* $\Rightarrow$ $'a$ *option* $\Rightarrow$ $'b$ *option*
where
    ap-option (*Some f*) (*Some x*) = *Some* (*f x*)
  | *ap-option* - - = *None*

abbreviation (*input*) *pure-option* :: $'a \Rightarrow 'a$ *option*
where *pure-option* $\equiv$ *Some*

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

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

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

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

applicative *option* (*C*, *W*)
for
pure: *Some*
ap: *ap-option*
rel: *rel-option*
set: *set-option*
proof −
include *applicative-syntax*
{ fix *x* :: *'a option*
show *pure* (λ*x*. *x*) ⋄ *x* = *x* **by** (*cases x*) *simp-all*
next
fix *g* :: (*'b* ⇒ *'c*) *option* **and** *f* :: (*'a* ⇒ *'b*) *option* **and** *x*
show *pure* (λ*g 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* (λ*f 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* ⇒ *'a* ⇒ *'b*) *option* **and** *x*
show *pure* (λ*f 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*
}

qed (*simp add*: *some-ap-option ap-some-option*)

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

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

end

2.3 Sum types

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

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

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

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

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

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

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

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

lemma (**in** *semigroup*) *ap-sum-comp*:

```
  ap-sum f (ap-sum f (ap-sum f (Inl (o)) h) g) x = ap-sum f h (ap-sum f g x)
by(cases h g x rule: sum.exhaust[case-product sum.exhaust, case-product sum.exhaust])
  (simp-all add: local.assoc)

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

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

applicative either (W)
for
  pure: Inl
  ap: ap-either
  rel: λA. rel-sum A B
proof −
  include applicative-syntax
  { fix f :: ('c ⇒ 'c ⇒ 'd) + 'a and x
    show pure (λf x. f x x) ⋄ f ⋄ x = f ⋄ x ⋄ x
      by (cases f x rule: sum.exhaust[case-product sum.exhaust]) simp-all
  next
    interpret semigroup λx y. x by(rule semigroup-const)
    fix g :: ('d ⇒ 'e) + 'a and f :: ('c ⇒ 'd) + 'a and x
    show pure (λg f x. g (f x)) ⋄ g ⋄ f ⋄ x = g ⋄ (f ⋄ x)
      by(rule ap-sum-comp[simplified comp-def[abs-def]])
  next
    fix R and f :: ('c ⇒ 'd) + 'b and g :: ('c ⇒ 'e) + 'b and x
    assume rel-sum (rel-fun (eq-on UNIV) R) B f g
    then show rel-sum R B (f ⋄ x) (g ⋄ x)
      by (cases f g x rule: sum.exhaust[case-product sum.exhaust, case-product
sum.exhaust])
      (auto intro: B-refl[THEN reflpD] elim: rel-funE)
  }
qed (auto intro: ap-sum-id[simplified id-def] ap-sum-ichng)

end

interpretation either-af (=) by unfold-locales simp

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

by *auto*

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

end

2.4 Set with Cartesian product

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

definition *ap-set* :: $('a \Rightarrow 'b)$ *set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'b$ *set*
where *ap-set F X* = $\{f\ x \mid f\ x.\ f \in F \wedge x \in X\}$

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: $\lambda x.\ \{x\}$
ap: *ap-set*
rel: *rel-set*
set: $\lambda x.\ x$
proof −
fix *R* :: $'a \Rightarrow 'b \Rightarrow$ *bool*
show *rel-fun R* (*rel-set R*) ($\lambda x.\ \{x\}$) ($\lambda x.\ \{x\}$) **by** (*auto intro*: *rel-setI*)
next
fix *R* **and** *f* :: $('a \Rightarrow 'b)$ *set* **and** *g* :: $('a \Rightarrow 'c)$ *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* ($\lambda f.$ *List.bind xs* ($\lambda x.$ [*f x*]))

adhoc-overloading *Applicative.ap ap-list*

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

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

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

context includes *applicative-syntax*
begin

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

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

applicative *list*
for
pure: λ*x*. [*x*]
ap: *ap-list*
rel: *list-all2*
set: *set*
proof −
fix *x* :: *'a list*
show [λ*x*. *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* = λ*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 $[?B] \diamond (g \# gs) \diamond f \diamond x = [?B\ g] \diamond f \diamond x \mathbin{@} [?B] \diamond gs \diamond f \diamond x$
by (*simp add*: *cons-ap-list append-ap-distrib*)
also have $\ldots = [g] \diamond (f \diamond x) \mathbin{@} gs \diamond (f \diamond x)$ **using** *g-comp Cons.IH* **by** *simp*
also have $\ldots = (g \# gs) \diamond (f \diamond x)$ **by** (*simp add*: *cons-ap-list*)
finally show *?case* **.**
qed
next
fix $f :: ('a \Rightarrow 'b)\ list$ **and** x
show $f \diamond [x] = [\lambda f.\ f\ x] \diamond f$ **unfolding** *ap-list-def* **by** *simp*
next
fix $R :: 'a \Rightarrow 'b \Rightarrow bool$
show *rel-fun R* (*list-all2 R*) ($\lambda x.\ [x]$) ($\lambda x.\ [x]$) **by** *transfer-prover*
next
fix R **and** $f :: ('a \Rightarrow 'b)\ list$ **and** $g :: ('a \Rightarrow 'c)\ list$ **and** x
assume [*transfer-rule*]: *list-all2* (*rel-fun* (*eq-on* (*set x*)) *R*) *f g*
have [*transfer-rule*]: *list-all2* (*eq-on* (*set x*)) *x x* **by** (*simp add*: *list-all2-same*)
show *list-all2 R* ($f \diamond x$) ($g \diamond x$) **by** *transfer-prover*
qed (*simp add*: *cons-ap-list*)

lemma *map-ap-conv*[*applicative-unfold*]: *map f x* = $[f] \diamond x$
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 *bind-eq-Nil-iff* [*simp*]: *List.bind xs f* = [] $\longleftrightarrow$ ($\forall x{\in}set\ xs.\ f\ x$ = [])
by(*simp add*: *List.bind-def*)

lemma *zip-eq-Nil-iff* [*simp*]: *zip xs ys* = [] $\longleftrightarrow$ *xs* = [] $\vee$ *ys* = []
by(*cases xs ys rule*: *list.exhaust*[*case-product list.exhaust*]) *simp-all*

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

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

lemma *remdups-append1-drop*: *set xs* $\subseteq$ *set ys* $\Longrightarrow$ *remdups* (*xs* @ *ys*) = *remdups ys*

by(*induction xs*) *auto*

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

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

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

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

setup-lifting *type-definition-dnelist*

lemma *dnelist-subtype-dlist*:
type-definition (λ*x*. *Dlist* (*list-of-dnelist x*)) (λ*x*. *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-DNEList.map* = *map-fun id* (*map-fun list-of-dnelist Abs-dnelist*) (λ*f 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* (=))) (λ*f xs*. *remdups* (*map f xs*)) *Applicative-DNEList.map*
by(*simp add*: *map-def rel-fun-def dnelist.pcr-cr-eq cr-dnelist-def list-of-dnelist*[*simplified*]

Abs-dnelist-inverse)

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

abbreviation (*input*) *pure-dnelist* :: *'a* ⇒ *'a dnelist*
where *pure-dnelist* ≡ *single*

end

lift-definition *ap-dnelist* :: (*'a* ⇒ *'b*) *dnelist* ⇒ *'a dnelist* ⇒ *'b dnelist*
is λ*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*)

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* (λ*x*. *x*) ⋄ *x* = *x* **for** *x* :: *'a dnelist*
    by *transfer simp*

  have ∗: *remdups* (*remdups* (*remdups* ([λ*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* (λ*f*. *map f x*) (*remdups* (*concat* (*map* (λ*x*. *map* (λ*f y*. *x* (*f y*)) *f*) *g*)))))
      unfolding *ap-list-def List.bind-def*
    by(*subst* (*2*) *remdups-concat-remdups*[*symmetric*])(*simp add*: *o-def remdups-map-remdups remdups-concat-remdups*)
    also have ... = *remdups* (*concat* (*map* (λ*f*. *map f x*) (*concat* (*map* (λ*x*. *map* (λ*f y*. *x* (*f y*)) *f*) *g*))))
    by(*subst* (*1*) *remdups-concat-remdups*[*symmetric*])(*simp add*: *remdups-map-remdups*

remdups-concat-remdups)
also have ... = *remdups* (*concat* (*map remdups* (*map* (λ*g*. *map g* (*concat* (*map* (λ*f*. *map f x*) *f*))) *g*)))
using *list.pure-B-conv*[*of g f x*] **unfolding** *remdups-concat-map*
by(*simp add*: *ap-list-def List.bind-def o-def*)
also have ... = *?rhs* **unfolding** *ap-list-def List.bind-def*
by(*subst* (*2*) *remdups-concat-map*[*symmetric*])(*simp add*: *o-def remdups-map-remdups*)
finally show *?thesis* **.**
qed
show *pure-dnelist* (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 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-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*, '*b*) *monoid-ap* ⇒ ('*a*, '*c*) *monoid-ap*}))]
›

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

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

applicative *comm-monoid-add* (*C*)
for *pure*: *pure-monoid-add* :: - ⇒ (- :: *comm-monoid-add*, -) *monoid-ap*
ap: *ap-monoid-add* :: (- :: *comm-monoid-add*, -) *monoid-ap* ⇒ -
apply(*rule monoid-add.homomorphism monoid-add.pure-B-conv monoid-add.interchange*)+
subgoal for *f x y* **by**(*cases f x y rule*: *monoid-ap.exhaust*[*case-product monoid-ap.exhaust*, *case-product monoid-ap.exhaust*])(*simp add*: *pure-monoid-add-def add-ac*)
apply(*rule monoid-add.pure-I-conv*)
done

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

applicative *idemp-monoid-add* (*W*)
for *pure*: *pure-monoid-add* :: - ⇒ (- :: *idemp-monoid-add*, -) *monoid-ap*
ap: *ap-monoid-add* :: (- :: *idemp-monoid-add*, -) *monoid-ap* ⇒ -
apply(*rule monoid-add.homomorphism monoid-add.pure-B-conv monoid-add.pure-I-conv*)+
subgoal for *f x* **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* (+) ⋄ (*x* :: (*nat*, *int*) *monoid-ap*) ⋄ *y* = *pure* (+) ⋄ *y* ⋄ *x*
by(*applicative-lifting comm-monoid-add*) *simp*

end

3.2 Filters

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

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

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

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

declare *filtermap-parametric*[*transfer-rule*]

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

applicative *filter* (*C*)
— K is available for not-*bot* filters and W isholds not available
for
pure: *pure-filter*
ap: *ap-filter*
rel: *rel-filter*
proof −
show *ap-filter* (*pure-filter f*) (*pure-filter x*) = *pure-filter* (*f x*) **for** *f* :: *'a* ⇒ *'b*
and *x*

by(*simp add*: *ap-filter-def pure-filter-def principal-prod-principal*)
show *ap-filter* (*ap-filter* (*ap-filter* (*pure-filter* (λ*g 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*
by(*simp add*: *ap-filter-def pure-filter-def filtermap-filtermap prod-filtermap1 prod-filtermap2 apfst-def case-prod-map-prod prod-filter-assoc prod-filter-principal-singleton split-beta*)
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* (*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(*simp add*: *ap-filter-def pure-filter-def filtermap-filtermap prod-filter-principal-singleton2 prod-filter-principal-singleton prod-filtermap1 prod-filtermap2 prod-filter-assoc split-beta*)
apply(*subst* (*2*) *prod-filter-commute*)
apply(*simp add*: *filtermap-filtermap prod-filtermap1 prod-filtermap2*)
done
show *rel-fun R* (*rel-filter R*) *pure-filter pure-filter* **for** *R* :: '*a* ⇒ '*b* ⇒ *bool*
by(*rule pure-filter-parametric*)
show *rel-filter R* (*ap-filter f x*) (*ap-filter g x*) **if** *rel-filter* (*rel-fun* (*eq-on UNIV*) *R*) *f g*
for *R* **and** *f* :: ('*a* ⇒ '*b*) *filter* **and** *g* :: ('*a* ⇒ '*c*) *filter* **and** *x*
supply *that*[*unfolded eq-on-UNIV*, *transfer-rule*] **by** *transfer-prover*
qed

end

3.3 State monad

theory *Applicative-State*
imports
Applicative
HOL−Library.State-Monad
begin

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

end

3.4 Streams as an applicative functor

theory *Applicative-Stream* **imports**
Applicative
HOL−Library.Stream
HOL−Library.Adhoc-Overloading
begin

primcorec (*transfer*) *ap-stream* :: ('*a* ⇒ '*b*) *stream* ⇒ '*a stream* ⇒ '*b stream*

```
where
    shd (ap-stream f x) = shd f (shd x)
  | stl (ap-stream f x) = ap-stream (stl f) (stl x)

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

context includes lifting-syntax applicative-syntax
begin

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

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

lemma ap-stream-interchange: f ⋄ pure x = pure (λf. f x) ⋄ f
by (coinduction arbitrary: f) auto

lemma ap-stream-composition: pure (λg f x. g (f x)) ⋄ g ⋄ f ⋄ x = g ⋄ (f ⋄ x)
by (coinduction arbitrary: g f x) auto

applicative stream (S, K)
for
  pure: sconst
  ap: ap-stream
  rel: stream-all2
  set: sset
proof −
  fix g :: ('b ⇒ 'a ⇒ 'c) stream and f x
  show pure (λg f x. g x (f x)) ⋄ g ⋄ f ⋄ x = g ⋄ x ⋄ (f ⋄ x)
    by (coinduction arbitrary: g f x) auto
next
  fix x :: 'b stream and y :: 'a stream
  show pure (λx y. x) ⋄ x ⋄ y = x
    by (coinduction arbitrary: x y) auto
next
  fix R :: 'a ⇒ 'b ⇒ bool
  show (R ===> stream-all2 R) pure pure
  proof
    fix x y
    assume R x y
    then show stream-all2 R (pure x) (pure y)
      by coinduction simp
  qed
next
  fix R and f :: ('a ⇒ 'b) stream and g :: ('a ⇒ 'c) stream and x
  assume [transfer-rule]: stream-all2 (eq-on (sset x) ===> R) f g
 have [transfer-rule]: stream-all2 (eq-on (sset x)) x x by(simp add: stream.rel-refl-strong)
```

show *stream-all2 R* (*f* ⋄ *x*) (*g* ⋄ *x*) **by** *transfer-prover*
qed (*rule ap-stream-homo*)

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

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

end

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* = (λ*s*. *case f s of* (*g*, *s*′) ⇒ *case x s*′ *of* (*y*, *s*′′) ⇒ (*g y*, *s*′′))

abbreviation (*input*) *pure-state* ≡ *Pair*

adhoc-overloading *Applicative.ap ap-state*

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

end

3.6 Probability mass functions

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

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

definition *ap-pmf* :: ('*a* ⇒ '*b*) *pmf* ⇒ '*a pmf* ⇒ '*b pmf*

where *ap-pmf f x* = *map-pmf* (λ(*f*, *x*). *f x*) (*pair-pmf f x*)

adhoc-overloading *Applicative.ap ap-pmf*

context includes *applicative-syntax*
begin

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

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

lemma *ap-pmf-homo*: *pure-pmf f* ⋄ *pure-pmf x* = *pure-pmf* (*f x*)
by(*simp add*: *ap-pmf-def pair-return-pmf1*)

lemma *ap-pmf-interchange*: *u* ⋄ *pure-pmf x* = *pure-pmf* (λ*f*. *f x*) ⋄ *u*
by(*simp add*: *ap-pmf-def pair-return-pmf1 pair-return-pmf2 pmf.map-comp o-def*)

lemma *ap-pmf-K*: *return-pmf* (λ*x* -. *x*) ⋄ *x* ⋄ *y* = *x*
by(*simp add*: *ap-pmf-def pair-map-pmf1 pmf.map-comp pair-return-pmf1 o-def split-def map-fst-pair-pmf*)

lemma *ap-pmf-C*: *return-pmf* (λ*f x y*. *f y x*) ⋄ *f* ⋄ *x* ⋄ *y* = *f* ⋄ *y* ⋄ *x*
apply(*simp add*: *ap-pmf-def pair-map-pmf1 pmf.map-comp pair-return-pmf1 pair-pair-pmf o-def split-def*)
apply(*subst* (*2*) *pair-commute-pmf*)
apply(*simp add*: *pair-map-pmf2 pmf.map-comp o-def split-def*)
done

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

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

show *rel-pmf R* (*ap-pmf f x*) (*ap-pmf g x*) **by** *transfer-prover*
qed(*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*x*. *sum-list* (*f x*)) *xs*)
by(*induction xs*) *simp-all*

context includes *applicative-syntax* **begin**

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

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

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

setup-lifting *type-definition-pfp*

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

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

qed

adhoc-overloading *Applicative.ap ap-pfp*

applicative *pfp*
for *pure*: *pure-pfp*
ap: *ap-pfp*
proof −
show *pure-pfp* (λ*x*. *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* (λ*g 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* (λ*f*. *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*)
qed(*transfer*; *rule refl*)+

end

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

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

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

adhoc-overloading *Applicative.ap ap-vec*

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

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

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

end

theory *Applicative-Functor* **imports**
Applicative-Environment
Applicative-Option
Applicative-Sum
Applicative-Set
Applicative-List
Applicative-DNEList
Applicative-Monoid
Applicative-Filter
Applicative-State
Applicative-Stream
Applicative-Open-State
Applicative-PMF
Applicative-Probability-List
Applicative-Star
Applicative-Vector
begin

print-applicative

end

4 Examples of applicative lifting

4.1 Algebraic operations for the environment functor

theory *Applicative-Environment-Algebra* **imports**
Applicative-Environment

HOL−Library.Function-Division
begin

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

context includes *applicative-syntax*
begin

lemma *plus-fun-af* [*applicative-unfold*]: $f + g = pure\ (+) \diamond f \diamond g$
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*]: $f * g = pure\ (*) \diamond f \diamond g$
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*]: $of\text{-}nat\ n = pure\ (of\text{-}nat\ n)$
unfolding *of-nat-fun const-def* **..**

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

lemma *divide-fun-af* [*applicative-unfold*]: $divide\ f\ g = pure\ divide \diamond f \diamond g$
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*]: $x + y = pure\ (+) \diamond x \diamond (y :: 'a\ stream)$
end
instance ..
end

instantiation *stream* :: (*minus*) *minus* **begin**
context includes *applicative-syntax* **begin**
definition [*applicative-unfold*]: $x - y = pure\ (-) \diamond x \diamond (y :: 'a\ stream)$
end
instance ..
end

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

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

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

instantiation *stream* :: (*modulo*) *modulo* **begin**
context includes *applicative-syntax* **begin**
definition [*applicative-unfold*]: $x\ div\ y = pure\ (div) \diamond x \diamond (y :: 'a\ stream)$
definition [*applicative-unfold*]: $x\ mod\ y = pure\ (mod) \diamond x \diamond (y :: 'a\ stream)$
end
instance ..
end

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

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

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

instance *stream* :: (*ab-semigroup-mult*) *ab-semigroup-mult*

using *mult.commute* **by** *intro-classes applicative-lifting*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

instance *stream* :: (*ring*) *ring* **..**

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

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

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

instance *stream* :: (*numeral*) *numeral* **..**

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

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

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

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

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

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

definition leaves :: 'a tree ⇒ nat
where leaves = fold-tree (λ-. 1) (+)

lemma leaves-simps [simp]:
  leaves (Leaf x) = Suc 0
  leaves (Node l r) = leaves l + leaves r
by(simp-all add: leaves-def)
```

4.3.1 Pure correctness statement

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

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

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

declare [[show-variants]]

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

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

primrec label-list :: 'a list ⇒ ('s, 'x list) state
where
    label-list [] = pure []
```

```
| label-list (x # xs) = pure (#) ⋄ fresh ⋄ label-list xs

lemma label-append: label-list (a @ b) = pure (@) ⋄ label-list a ⋄ label-list b
  — The proof lifts the defining equations of (@) to the state monad.
proof (induction a)
  case Nil
  show ?case
    unfolding append.simps label-list.simps
    by applicative-nf simp
next
  case (Cons a1 a2)
  show ?case
    unfolding append.simps label-list.simps Cons.IH
    by applicative-nf simp
qed

lemma label-tree-list: pure labels ⋄ label-tree t = label-list (labels t)
proof (induction t)
  case Leaf show ?case unfolding label-tree-simps labels-simps label-list.simps
    by applicative-nf simp
next
 case Node show ?case unfolding label-tree-simps labels-simps label-append Node.IH[symmetric]
    by applicative-nf simp
qed
```

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

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

end
```

4.3.2 Correctness via monadic traversals

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

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

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

lift-definition ap-dual :: (('a ⇒ ('a ⇒ 'b) ⇒ 'b) ⇒ 'af1) ⇒ ('af1 ⇒ 'af3 ⇒
'af13) ⇒ ('af13 ⇒ 'af2 ⇒ 'af) ⇒ 'af2 dual ⇒ 'af3 dual ⇒ 'af dual
```

is $\lambda pure\ ap1\ ap2\ f\ x.\ ap2\ (ap1\ (pure\ (\lambda x\ f.\ f\ x))\ x)\ f$ **.**

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

definition $pure\text{-}state\text{-}rev :: 'a \Rightarrow ('s, 'a)\ state\text{-}rev$
where $pure\text{-}state\text{-}rev = pure\text{-}dual\ State\text{-}Monad.return$

definition $ap\text{-}state\text{-}rev :: ('s, 'a \Rightarrow 'b)\ state\text{-}rev \Rightarrow ('s, 'a)\ state\text{-}rev \Rightarrow ('s, 'b)\ state\text{-}rev$
where $ap\text{-}state\text{-}rev = ap\text{-}dual\ State\text{-}Monad.return\ State\text{-}Monad.ap\ State\text{-}Monad.ap$

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

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

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

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

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

lemma $ap\text{-}state\text{-}rev\text{-}rev\text{-}B: B\ f \diamond B\ x = B\ (pure\text{-}state\text{-}rev\ (\lambda x\ f.\ f\ x) \diamond x \diamond f)$
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.

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

lemma *ap-pure-bind-state*: *pure x* ⋄ *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* (**infixl** · 55)
where [*simp*]: *kleisli-state g f a* = *State-Monad.bind* (*f a*) *g*

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

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

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

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

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

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

lemma *recurse-Node*:
  recurse f (*Node l r*) = *pure* (λ*r 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* (λ*x f*. *f x*) ⋄ *recurse f r* ⋄ (*pure* (λ*x f*. *f x*) ⋄ *recurse f l* ⋄ *pure Node*)
by(*simp add*: *ap-state-rev-B B-inverse*)
also have ... = *pure* (λ*r l*. *Node l r*) ⋄ *recurse f r* ⋄ *recurse f l*
— This step expands to 13 steps in [1]
by(*applicative-nf*) *simp*
finally show *?thesis* **.**
qed

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

B ∘ *B* is an idiom morphism

lemma *B-pure*: *pure x* = *B* (*State-Monad.return x*)
unfolding *pure-state-rev-def* **by** *transfer simp*

lemma *BB-pure*: *pure x* = *B* (*B* (*pure x*))
unfolding *pure-state-rev-rev-def B-pure*[*symmetric*] **by** *transfer*(*rule refl*)

lemma *BB-ap*: *B* (*B f*) ⋄ *B* (*B x*) = *B* (*B* (*f* ⋄ *x*))
proof −
have *B* (*B f*) ⋄ *B* (*B x*) = *B* (*B* (*pure* (λ*x f*. *f x*) ⋄ *f* ⋄ (*pure* (λ*x f*. *f x*) ⋄ *x* ⋄ *pure* (λ*x 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*
by(*simp add*: *ap-conv-bind-state recurse-Node*)(*simp add*: *State-Monad.bind-assoc*[*symmetric*] *Node.IH*)
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*
by(*simp add*: *strip-def adorn-def fun-eq-iff fetch-def*[*abs-def*] *ap-conv-bind-state*)

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* (λ*x*. *pure* [*x*])
    (λ*l r*. *pure* (*case-prod append*) ⋄ (*assert* (*case-prod disjoint*) (*pure Pair* ⋄ *l* ⋄ *r*)))

lemma *dlabels-simps* [*simp*]:

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

lemma correctness-applicative:
  assumes distinct: ⋀n. pure (assert distinct) ⋄ symbols n = symbols n
  shows State-Monad.return dlabels ⋄ label-tree t = symbols (leaves t)
proof(induction t)
  show pure dlabels ⋄ label-tree (Leaf x) = symbols (leaves (Leaf x)) for x :: 'a
    unfolding label-tree-simps leaves-simps repeatM.simps by applicative-nf simp
next
  fix l r :: 'a tree
  assume IH: pure dlabels ⋄ label-tree l = symbols (leaves l) pure dlabels ⋄ label-tree
r = symbols (leaves r)
  let ?cat = case-prod append and ?disj = case-prod disjoint
  let ?f = λl r. pure ?cat ⋄ (assert ?disj (pure Pair ⋄ l ⋄ r))
  have State-Monad.return dlabels ⋄ label-tree (Node l r) =
        pure ?f ⋄ (pure dlabels ⋄ label-tree l) ⋄ (pure dlabels ⋄ label-tree r)
    unfolding label-tree-simps by applicative-nf simp
  also have ... = pure ?f ⋄ (pure (assert distinct) ⋄ symbols (leaves l)) ⋄ (pure
(assert distinct) ⋄ symbols (leaves r))
    unfolding IH distinct ..
  also have ... = pure (assert distinct) ⋄ symbols (leaves (Node l r))
    unfolding leaves-simps repeatM-plus by applicative-nf simp
  also have ... = symbols (leaves (Node l r)) by(rule distinct)
  finally show pure dlabels ⋄ label-tree (Node l r) = symbols (leaves (Node l r)) .
qed

end
```

4.3.4 Probabilistic tree relabelling

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

datatype dir = Left | Right

hide-const (open) path

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

termination by *lexicographic-order*

adhoc-overloading *Applicative.pure pure-pmf*

context fixes $p :: 'a \Rightarrow 'b\ pmf$ **begin**

primrec *plabel* :: $'a\ tree \Rightarrow 'b\ tree\ pmf$
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

end

5 Formalisation of idiomatic terms and lifting

5.1 Immediate joinability under a relation

theory *Joinable*
imports *Main*
begin

5.1.1 Definition and basic properties

definition *joinable* :: $('a \times 'b)$ *set* $\Rightarrow$ $('a \times 'a)$ *set*
where *joinable* $R = \{(x, y).\ \exists z.\ (x, z) \in R \wedge (y, z) \in R\}$

lemma *joinable-simp*: $(x, y) \in$ *joinable* $R \longleftrightarrow (\exists z.\ (x, z) \in R \wedge (y, z) \in R)$
unfolding *joinable-def* **by** *simp*

lemma *joinableI*: $(x, z) \in R \Longrightarrow (y, z) \in R \Longrightarrow (x, y) \in$ *joinable* R
unfolding *joinable-simp* **by** *blast*

lemma *joinableD*: $(x, y) \in$ *joinable* $R \Longrightarrow \exists z.\ (x, z) \in R \wedge (y, z) \in R$
unfolding *joinable-simp* **.**

lemma *joinableE*:
  assumes $(x, y) \in$ *joinable* R
  obtains z **where** $(x, z) \in R$ **and** $(y, z) \in R$
using *assms* **unfolding** *joinable-simp* **by** *blast*

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

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

lemma *refl-joinable*: *refl* $R \Longrightarrow$ *refl* (*joinable* R)
using *refl-joinable-iff* **by** (*blast dest*: *refl-onD*)

lemma *joinable-refl*: *refl* $R \Longrightarrow (x, x) \in$ *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*: $(x, y) \in$ *joinable* $R \Longrightarrow (y, x) \in$ *joinable* R
using *sym-joinable* **by** (*rule symD*)

lemma *joinable-mono*: $R \subseteq S \Longrightarrow$ *joinable* $R \subseteq$ *joinable* S
by (*rule subrelI*) (*auto simp only*: *joinable-simp*)

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

lemma *joinable-subst*:
assumes *R-subst*: $\bigwedge x\ y.\ (x, y) \in R \Longrightarrow (P\ x, P\ y) \in R$
assumes *joinable*: $(x, y) \in$ *joinable R*
shows $(P\ x, P\ y) \in$ *joinable R*
proof −
from *joinable* **obtain** *z* **where** *xz*: $(x, z) \in R$ **and** *yz*: $(y, z) \in R$ **by** (*rule joinableE*)
from *R-subst xz* **have** $(P\ x, P\ z) \in R$ **.**
moreover from *R-subst yz* **have** $(P\ y, P\ z) \in R$ **.**
ultimately show *?thesis* **by** (*rule joinableI*)
qed

5.1.2 Confluence

definition *confluent* :: *'a rel* ⇒ *bool*
where *confluent R* ⟷ $(\forall x\ y\ y'.\ (x, y) \in R \wedge (x, y') \in R \longrightarrow (y, y') \in$ *joinable R*)

lemma *confluentI*:
$(\bigwedge x\ y\ y'.\ (x, y) \in R \Longrightarrow (x, y') \in R \Longrightarrow \exists z.\ (y, z) \in R \wedge (y', z) \in R) \Longrightarrow$ *confluent R*
unfolding *confluent-def* **by** (*blast intro*: *joinableI*)

lemma *confluentD*:
confluent R $\Longrightarrow (x, y) \in R \Longrightarrow (x,y') \in R \Longrightarrow (y, y') \in$ *joinable R*
unfolding *confluent-def* **by** *blast*

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

lemma *trans-joinable*:
assumes *trans R* **and** *confluent R*
shows *trans* (*joinable R*)
proof (*rule transI*)
fix *x y z*
assume $(x, y) \in$ *joinable R*

then obtain u **where** xu: $(x,\ u) \in R$ **and** yu: $(y,\ u) \in R$ **by** (*rule joinableE*)
assume $(y,\ z) \in$ *joinable* R
then obtain v **where** yv: $(y,\ v) \in R$ **and** zv: $(z,\ v) \in R$ **by** (*rule joinableE*)
from yu yv ‹*confluent* R› **obtain** w **where** uw: $(u,\ w) \in R$ **and** vw: $(v,\ w) \in R$
by (*blast elim*: *confluentE*)
from xu uw ‹*trans* R› **have** $(x,\ w) \in R$ **by** (*blast elim*: *transE*)
moreover from zv vw ‹*trans* R› **have** $(z,\ w) \in R$ **by** (*blast elim*: *transE*)
ultimately show $(x,\ z) \in$ *joinable* R **by** (*rule joinableI*)
qed

5.1.3 Relation to reflexive transitive symmetric closure

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

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

5.1.4 Predicate version

definition *joinablep* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where *joinablep* $P\ x\ y \longleftrightarrow (\exists z.\ P\ x\ z \wedge P\ y\ z)$

lemma *joinablep-joinable*[*pred-set-conv*]:
  joinablep (λ*x y*. (*x*, *y*) ∈ *R*) = (λ*x 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 *liftn-lift-swap*: *liftn n* (*lift t k*) *k* = *lift* (*liftn n t k*) *k*
by (*induction n*) *simp-all*

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

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

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

5.2.2 Reduction

abbreviation *beta-eta* :: *dB* ⇒ *dB* ⇒ *bool* (**infixl** →$_{\beta\eta}$ *50*)
where *beta-eta* ≡ *sup beta eta*

abbreviation *beta-eta-reds* :: *dB* ⇒ *dB* ⇒ *bool* (**infixl** →$_{\beta\eta}{}^*$ *50*)
where *s* →$_{\beta\eta}{}^*$ *t* ≡ (*beta-eta*)** *s t*

lemma *beta-into-beta-eta-reds*: *s* →$_\beta$ *t* ⟹ *s* →$_{\beta\eta}{}^*$ *t*

by *auto*

lemma *eta-into-beta-eta-reds*: $s \to_\eta t \Longrightarrow s \to_{\beta\eta}{}^* t$
by *auto*

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

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

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

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

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

lemma *beta-eta-lift*: $s \to_{\beta\eta}{}^* t \Longrightarrow lift\ s\ k \to_{\beta\eta}{}^* lift\ t\ k$
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* ⇒ *dB* ⇒ *bool* (**infixl** ↔ *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*])

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

lemma *beta-into-equiv*[*elim*]: $s \rightarrow_{\beta}\ t \Longrightarrow s \leftrightarrow t$
by (*rule reds-into-equiv*) (*rule beta-into-beta-eta-reds*)

lemma *eta-into-equiv*[*elim*]: $s \rightarrow_{\eta}\ t \Longrightarrow s \leftrightarrow t$
by (*rule reds-into-equiv*) (*rule eta-into-beta-eta-reds*)

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

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

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

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

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

lemma *term-trans*[*trans*]: $s \leftrightarrow t \Longrightarrow t \leftrightarrow u \Longrightarrow s \leftrightarrow u$
unfolding *term-equiv-def*
using *trans-joinable*[*to-pred*] *trans-rtrancl*[*to-pred*] *confluent-beta-eta-reds*
by (*blast elim*: *transpE*)

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

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

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

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

lemma *equiv-app*: $s \leftrightarrow s' \Longrightarrow t \leftrightarrow t' \Longrightarrow s \circ t \leftrightarrow s' \circ t'$
by (*blast intro*: *term-trans*)

lemma *equiv-abs*[*intro*]: $t \leftrightarrow t' \Longrightarrow Abs\ t \leftrightarrow Abs\ t'$
unfolding *term-equiv-def* **using** *beta-eta-abs*

by (*iprover intro*: *joinable-subst*[*to-pred*])

lemma *equiv-lift*: $s \leftrightarrow t \Longrightarrow \mathit{lift}\ s\ k \leftrightarrow \mathit{lift}\ t\ k$
by (*auto intro*: *term-equivI beta-eta-lift elim*: *term-equivE*)

lemma *equiv-liftn*: $s \leftrightarrow t \Longrightarrow \mathit{liftn}\ n\ s\ k \leftrightarrow \mathit{liftn}\ n\ t\ k$
by (*induction n*) (*auto intro*: *equiv-lift*)

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

lemma *equiv-eq-rtscl-reds*: $\mathit{term\text{-}equiv} = (\mathit{sup}\ \mathit{beta\text{-}eta}\ \mathit{beta\text{-}eta}^{-1-1})^{**}$
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*: $\mathcal{I} = \mathit{Abs}\ (\mathit{Var}\ 0)$
definition *B-def*: $\mathcal{B} = \mathit{Abs}\ (\mathit{Abs}\ (\mathit{Abs}\ (\mathit{Var}\ 2 \circ (\mathit{Var}\ 1 \circ \mathit{Var}\ 0))))$
definition *T-def*: $\mathcal{T} = \mathit{Abs}\ (\mathit{Abs}\ (\mathit{Var}\ 0 \circ \mathit{Var}\ 1))$ — reverse application

lemma *I-eval*: $\mathcal{I} \circ x \rightarrow_\beta x$
proof −
  have $\mathcal{I} \circ x \rightarrow_\beta \mathit{Var}\ 0[x/0]$ **unfolding** *I-def* **..**
  then show *?thesis* **by** *simp*
qed

lemma *I-equiv*[*iff*]: $\mathcal{I} \circ x \leftrightarrow x$
using *I-eval* **..**

lemma *I-closed*[*simp*]: $\mathit{liftn}\ n\ \mathcal{I}\ k = \mathcal{I}$
unfolding *I-def* **by** *simp*

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

lemma *B-eval2*: $\mathcal{B} \circ g \circ f \rightarrow_\beta^* \mathit{Abs}\ (\mathit{lift}\ g\ 0 \circ (\mathit{lift}\ f\ 0 \circ \mathit{Var}\ 0))$
proof −
  have $\mathcal{B} \circ g \circ f \rightarrow_\beta^* \mathit{Abs}\ (\mathit{Abs}\ (\mathit{lift}\ (\mathit{lift}\ g\ 0)\ 0 \circ (\mathit{Var}\ 1 \circ \mathit{Var}\ 0))) \circ f$
    using *B-eval1* **by** *blast*

also have ... $\rightarrow_\beta$ *Abs* (*lift* (*lift g 0*) *0* $\circ$ (*Var 1* $\circ$ *Var 0*)) [*f*/*0*] **..**
also have ... = *Abs* (*lift g 0* $\circ$ (*lift f 0* $\circ$ *Var 0*)) **by** *simp*
finally show *?thesis* **.**
qed

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

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

lemma *B-closed*[*simp*]: *liftn n* $\mathcal{B}$ *k* = $\mathcal{B}$
unfolding *B-def* **by** *simp*

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

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

lemma *T-equiv*[*iff*]: $\mathcal{T} \circ x \circ f \leftrightarrow f \circ x$
using *T-eval* **..**

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

end

5.4 Idiomatic terms – Properties and operations

theory *Idiomatic-Terms*
imports *Combinators*
begin

This theory proves the correctness of the normalisation algorithm for 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 (∀ *x''*. *x* = *Pure x''* ⟶ (∃ *y'*. *y* = *Pure y'* ∧ *x''* ↔ *y'*)) ∧

```
    (∀ x''. y = Pure x'' ⟶ (∃ y'. x = Pure y' ∧ x'' ↔ y'))
  proof (induction)
    case pure-cong thus ?case by (auto intro: term-sym)
  next
    case itrm-trans thus ?case by (fastforce intro: term-trans)
  qed simp-all
  with that show thesis unfolding x-def by blast
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 ≅ y unfolding x-def .
  then have (∀ x''. x = Opaque x'' ⟶ (∃ y'. y = Opaque y' ∧ x'' = y')) ∧
    (∀ x''. y = Opaque x'' ⟶ (∃ y'. x = Opaque y' ∧ x'' = y'))
  by induction fast+
  with that show thesis unfolding x-def by blast
qed

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

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

```
inductive idiom-rule :: 'a itrm ⇒ 'a itrm ⇒ bool
where
    idiom-id: idiom-rule (Pure 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 $x \simeq y \equiv$ *itrm-cong idiom-rule x y*

lemma *idiom-rule-into-equiv*: *idiom-rule x y* $\Longrightarrow$ $x \simeq y$ **..**

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

lemma *similar-into-equiv*: $x \cong y \Longrightarrow x \simeq y$
by (*induction pred*: *itrm-cong*) (*auto intro*: *ap-cong itrm-sym itrm-trans*)

lemma *opaque-equiv*: $x \simeq y \Longrightarrow$ *opaque x* = *opaque y*
proof (*induction pred*: *itrm-cong*)
  case (*into-itrm-cong x y*)
  thus *?case* **by** *induction auto*
qed *simp-all*

lemma *iorder-equiv*: $x \simeq y \Longrightarrow$ *iorder x* = *iorder y*
by (*auto dest*: *opaque-equiv*)

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

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

abbreviation *itrm-ext-equiv* :: *'a itrm* $\Rightarrow$ *'a itrm* $\Rightarrow$ *bool* (**infixl** $\simeq^+$ *50*)
where $x \simeq^+ y \equiv$ *itrm-cong idiom-ext-rule x y*

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

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

end

5.4.2 Syntactic unlifting

With generalisation of variables **primrec** *unlift'* :: *nat* $\Rightarrow$ *'a itrm* $\Rightarrow$ *nat* $\Rightarrow$ *dB*
where
    unlift' n (*Opaque* -) *i* = *Var i*
  | *unlift' n* (*Pure x*) *i* = *liftn n x 0*

```
  | unlift' n (f ⋄ x) i = unlift' n f (i + iorder x) ∘ unlift' n x i

abbreviation unlift x ≡ (Abs^^iorder x) (unlift' (iorder x) x 0)

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

lemma absn-cong[intro]: s ↔ t ⟹ (Abs^^n) s ↔ (Abs^^n) t
by (induction n) auto

lemma free-unlift: free (unlift' n x i) j ⟹ j ≥ n ∨ (j ≥ i ∧ j < i + iorder x)
proof (induction x arbitrary: i)
  case (Opaque x)
  thus ?case by simp
next
  case (Pure x)
  thus ?case using free-liftn by simp
next
  case (IAp x y)
  thus ?case by fastforce
qed

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

lemma unlift'-equiv: x ≃ y ⟹ unlift' n x i ↔ unlift' n y i
proof (induction arbitrary: n i pred: itrm-cong)
  case (into-itrm-cong x y) thus ?case
  proof induction
    case (idiom-id x)
    show ?case using I-equiv[symmetric] by simp
  next
    case (idiom-comp g f x)
    let ?G = unlift' n g (i + iorder f + iorder x)
    let ?F = unlift' n f (i + iorder x)
    let ?X = unlift' n x i
    have unlift' n (g ⋄ (f ⋄ x)) i = ?G ∘ (?F ∘ ?X)
      by (simp add: add.assoc)
```

```
    moreover have unlift' n (Pure 𝓑 ⋄ g ⋄ f ⋄ x) i = 𝓑 ∘ ?G ∘ ?F ∘ ?X
      by (simp add: add.commute add.left-commute)
    moreover have ?G ∘ (?F ∘ ?X) ↔ 𝓑 ∘ ?G ∘ ?F ∘ ?X using B-equiv[symmetric]
.
    ultimately show ?case by simp
  next
    case (idiom-hom f x)
    show ?case by auto
  next
    case (idiom-xchng f x)
    let ?F = unlift' n f i
    let ?X = liftn n x 0
    have unlift' n (f ⋄ Pure x) i = ?F ∘ ?X by simp
    moreover have unlift' n (Pure (𝓣 ∘ x) ⋄ f) i = 𝓣 ∘ ?X ∘ ?F by simp
    moreover have ?F ∘ ?X ↔ 𝓣 ∘ ?X ∘ ?F using T-equiv[symmetric] .
    ultimately show ?case by simp
  qed
next
  case pure-cong
  thus ?case by (auto intro: equiv-liftn)
next
  case (ap-cong f f' x x')
  from ‹x ≃ x'› have iorder-eq: iorder x = iorder x' by (rule iorder-equiv)
  have unlift' n (f ⋄ x) i = unlift' n f (i + iorder x) ∘ unlift' n x i by simp
  moreover have unlift' n (f' ⋄ x') i = unlift' n f' (i + iorder x) ∘ unlift' n x' i
    using iorder-eq by simp
  ultimately show ?case using ap-cong.IH by (auto intro: equiv-app)
next
  case itrm-refl
  thus ?case by simp
next
  case itrm-sym
  thus ?case using term-sym by simp
next
  case itrm-trans
  thus ?case using term-trans by blast
qed

lemma unlift-equiv: x ≃ y ⟹ unlift x ↔ unlift y
proof −
  assume x ≃ y
  then have unlift' (iorder y) x 0 ↔ unlift' (iorder y) y 0 by (rule unlift'-equiv)
  moreover from ‹x ≃ y› have iorder x = iorder y by (rule iorder-equiv)
  ultimately show ?thesis by auto
qed

Preserving variables  primrec unlift-vars :: nat ⇒ nat itrm ⇒ dB
where
  unlift-vars n (Opaque i) = Var i
```

```
  | unlift-vars n (Pure x) = liftn n x 0
  | unlift-vars n (x ◇ y) = unlift-vars n x ° unlift-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 unlift-ap: unlift-vars 0 (x ◇ y) = unlift-vars 0 x ° unlift-vars 0 y
    by simp
  from no-opaque IAp.IH have x ◇ y ≃ Pure (unlift-vars 0 x) ◇ Pure (unlift-vars
0 y)
    by (blast intro: ap-cong)
  also have ... ≃ Pure (unlift-vars 0 x ° unlift-vars 0 y) by (rule itrm-hom)
  also have ... = Pure (unlift-vars 0 (x ◇ y)) by (simp only: unlift-ap)
  finally show ?case .
qed
```

5.4.3 Canonical forms

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

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

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

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

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

lemma cf-unlift:
  assumes x ∈ CF
    shows CF-pure x ↔ unlift x
using assms proof (induction set: CF)
  case (pure-cf x)
  show ?case by simp
```

next
case (*ap-cf f x*)
let *?n* = *iorder f* + *1*
have *unlift* (*f* ◇ *Opaque x*) = (*Abs*^^*?n*) (*unlift′ ?n f 1* ∘ *Var 0*)
by *simp*
also have ... = (*Abs*^^*iorder f*) (*Abs* (*unlift′ ?n f 1* ∘ *Var 0*))
using *funpow-Suc-inside* **by** *simp*
also have ... ↔ *unlift f* **proof** −
have ¬ *free* (*unlift′ ?n f 1*) *0* **using** *free-unlift* **by** *fastforce*
hence *Abs* (*unlift′ ?n f 1* ∘ *Var 0*) →$_\eta$ (*unlift′ ?n f 1*)[*Var 0*/*0*] **..**
also have ... = *unlift′* (*iorder f*) *f 0*
using *unlift-subst* **by** (*metis One-nat-def Suc-eq-plus1 le0*)
finally show *?thesis*
by (*simp add*: *r-into-rtranclp absn-cong eta-into-equiv*)
qed
finally show *?case*
using *ap-cf.IH* **by** (*auto intro*: *term-sym term-trans*)
qed

lemma *cf-similarI*:
assumes *x* ∈ *CF y* ∈ *CF*
and *opaque x* = *opaque y*
and *CF-pure x* ↔ *CF-pure y*
shows *x* ≅ *y*
using *assms* **proof** (*induction arbitrary*: *y*)
case (*pure-cf x*)
hence *opaque y* = [] **by** *auto*
with ‹*y* ∈ *CF*› **obtain** *y′* **where** *y* = *Pure y′* **by** *cases auto*
with *pure-cf.prems* **show** *?case* **by** *auto*
next
case (*ap-cf f x*)
from ‹*opaque* (*f* ◇ *Opaque x*) = *opaque y*›
obtain *y1 y2* **where** *opaque y* = *y1* @ *y2*
and *opaque f* = *y1* **and** [*x*] = *y2* **by** *fastforce*
from ‹[*x*] = *y2*› **obtain** *y′* **where** *y2* = [*y′*] **and** *x* = *y′*
by *auto*
with ‹*y* ∈ *CF*› **and** ‹*opaque y* = *y1* @ *y2*› **obtain** *g*
where *opaque g* = *y1* **and** *y-split*: *y* = *g* ◇ *Opaque y′ g* ∈ *CF* **by** *cases auto*
with *ap-cf.prems* ‹*opaque f* = *y1*›
have *opaque f* = *opaque g CF-pure f* ↔ *CF-pure g* **by** *auto*
with *ap-cf.IH* ‹*g* ∈ *CF*› **have** *f* ≅ *g* **by** *simp*
with *ap-cf.prems y-split* ‹*x* = *y′*› **show** *?case* **by** (*auto intro*: *ap-cong*)
qed

lemma *cf-similarD*:
assumes *in-cf*: *x* ∈ *CF y* ∈ *CF*
and *similar*: *x* ≅ *y*
shows *CF-pure x* ↔ *CF-pure y* ∧ *opaque x* = *opaque y*
using *assms*

by (*blast intro*!: *similar-into-equiv opaque-equiv cf-unlift unlift-equiv*
intro: *term-trans term-sym*)

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

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

5.4.4 Normalisation of idiomatic terms

primrec *norm-pn* :: $dB \Rightarrow 'a\ itrm \Rightarrow 'a\ itrm$
where
norm-pn f (*Opaque x*) = *undefined*
| *norm-pn f* (*Pure x*) = *Pure* ($f \circ x$)
| *norm-pn f* ($n \diamond x$) = *norm-pn* ($\mathcal{B} \circ f$) $n \diamond x$

primrec *norm-nn* :: $'a\ itrm \Rightarrow 'a\ itrm \Rightarrow 'a\ itrm$
where
norm-nn n (*Opaque x*) = *undefined*
| *norm-nn n* (*Pure x*) = *norm-pn* ($\mathcal{T} \circ x$) *n*
| *norm-nn n* ($n' \diamond x$) = *norm-nn* (*norm-pn* $\mathcal{B}$ *n*) $n' \diamond x$

primrec *norm* :: $'a\ itrm \Rightarrow 'a\ itrm$
where
norm (*Opaque x*) = *Pure* $\mathcal{I} \diamond$ *Opaque x*
| *norm* (*Pure x*) = *Pure x*
| *norm* ($f \diamond x$) = *norm-nn* (*norm f*) (*norm x*)

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

lemma *norm-nn-in-cf*:
assumes $n \in CF\ n' \in CF$
shows *norm-nn n n'* $\in CF$
using *assms*(*2*,*1*)

by (*induction* n' *arbitrary*: n) (*auto intro*: *norm-pn-in-cf*)

lemma *norm-in-cf*: $norm\ x \in CF$
by (*induction* x) (*auto intro*: *norm-nn-in-cf*)

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

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

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

next
case (*Pure x*)
show *?case* **by** *simp*
next
case (*IAp f x*)
have *norm f* ∈ *CF* **and** *norm x* ∈ *CF* **by** (*rule norm-in-cf*)+
then have *norm-nn* (*norm f*) (*norm x*) ≃ *norm f* ⋄ *norm x*
by (*rule norm-nn-equiv*)
also have ... ≃ *f* ⋄ *x* **using** *IAp.IH* **..**
finally show *?case* **by** *simp*
qed

lemma *normal-form*: **obtains** *n* **where** *n* ≃ *x* **and** *n* ∈ *CF*
using *norm-equiv norm-in-cf* **..**

5.4.5 Lifting with normal forms

lemma *nf-unlift*:
assumes *equiv*: *n* ≃ *x* **and** *cf*: *n* ∈ *CF*
shows *CF-pure n* ↔ *unlift x*
proof −
from *cf* **have** *CF-pure n* ↔ *unlift n* **by** (*rule cf-unlift*)
also from *equiv* **have** *unlift n* ↔ *unlift x* **by** (*rule unlift-equiv*)
finally show *?thesis* **.**
qed

theorem *nf-lifting*:
assumes *opaque*: *opaque x* = *opaque y*
and *base-eq*: *unlift x* ↔ *unlift y*
shows *x* ≃ *y*
proof −
obtain *n* **where** *nf-x*: *n* ≃ *x n* ∈ *CF* **by** (*rule normal-form*)
obtain *n′* **where** *nf-y*: *n′* ≃ *y n′* ∈ *CF* **by** (*rule normal-form*)

from *nf-x* **have** *CF-pure n* ↔ *unlift x* **by** (*rule nf-unlift*)
also note *base-eq*
also from *nf-y* **have** *unlift y* ↔ *CF-pure n′* **by** (*rule nf-unlift*[*THEN term-sym*])
finally have *pure-eq*: *CF-pure n* ↔ *CF-pure n′* **.**

from *nf-x*(*1*) **have** *opaque n* = *opaque x* **by** (*rule opaque-equiv*)
also note *opaque*
also from *nf-y*(*1*) **have** *opaque y* = *opaque n′* **by** (*rule opaque-equiv*[*THEN sym*])
finally have *opaque-eq*: *opaque n* = *opaque n′* **.**

from *nf-x*(*1*) **have** *x* ≃ *n* **..**
also have *n* ≃ *n′*
using *nf-x nf-y pure-eq opaque-eq*
by (*blast intro*: *similar-into-equiv cf-similarI*)

also from *nf-y(1)* **have** $n' \simeq y$ **.**
finally show $x \simeq y$ **.**
qed

5.4.6 Bracket abstraction, twice

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

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

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

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

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

lemma *var-dist-frees*: $frees\ (var\text{-}dist\ vs\ t) = frees\ t \cup set\ vs$
by (*induction vs arbitrary*: *t*) *auto*

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

lemma *var-dist-subst-gt*:
$\forall\, v{\in}set\ vs.\ v < i \Longrightarrow (var\text{-}dist\ vs\ s)[t/i] = var\text{-}dist\ vs\ (s[t/i])$
by (*induction vs arbitrary*: *s*) *simp-all*

definition *vsubst* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where $vsubst\ u\ v\ w = (if\ u < w\ then\ u\ else\ if\ u = w\ then\ v\ else\ u - 1)$

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

lemma *vsubst-subst-lt*[*simp*]: $u < w \Longrightarrow vsubst\ u\ v\ w = u$
unfolding *vsubst-def* **by** *simp*

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

lemma *var-dist-cong*: $s \leftrightarrow t \Longrightarrow var\text{-}dist\ vs\ s \leftrightarrow var\text{-}dist\ vs\ t$
by (*induction vs arbitrary*: *s t*) *auto*

Preliminaries: Eta reductions with permuted variables **lemma** *absn-subst*:
$((Abs^{\frown\frown}n)\ s)[t/k] = (Abs^{\frown\frown}n)\ (s[liftn\ n\ t\ 0/k+n])$
by (*induction n arbitrary*: *t k*) (*simp-all add*: *liftn-lift-swap*)

lemma *absn-beta-equiv*: $(Abs^{\frown\frown}Suc\ n)\ s \circ t \leftrightarrow (Abs^{\frown\frown}n)\ (s[liftn\ n\ t\ 0/n])$
proof −
have $(Abs^{\frown\frown}Suc\ n)\ s \circ t = Abs\ ((Abs^{\frown\frown}n)\ s) \circ t$ **by** *simp*
also have ... $\leftrightarrow ((Abs^{\frown\frown}n)\ s)[t/0]$ **by** (*rule beta-into-equiv*) (*rule beta.beta*)
also have ... $= (Abs^{\frown\frown}n)\ (s[liftn\ n\ t\ 0/n])$ **by** (*simp add*: *absn-subst*)
finally show *?thesis* **.**
qed

lemma *absn-dist-eta*: $(Abs^{\frown\frown}n)\ (var\text{-}dist\ (rev\ [0..<n])\ (liftn\ n\ t\ 0)) \leftrightarrow t$
proof (*induction n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*)
let *?dist-range* $= \lambda a\ k.\ var\text{-}dist\ (rev\ [a..<k])\ (liftn\ k\ t\ 0)$
have *append*: $rev\ [0..<Suc\ n] = rev\ [1..<Suc\ n]\ @\ [0]$ **by** (*simp add*: *upt-rec*)
have *dist-last*: *?dist-range 0* (*Suc n*) = *?dist-range 1* (*Suc n*) ∘ *Var 0*
unfolding *append var-dist-append1* **..**

have ¬ *free* (*?dist-range 1* (*Suc n*)) *0* **proof** −
have *frees* (*?dist-range 1* (*Suc n*)) = *frees* (*liftn* (*Suc n*) *t 0*) ∪ {*1..n*}
unfolding *var-dist-frees* **by** *fastforce*
then have *0* ∉ *frees* (*?dist-range 1* (*Suc n*)) **by** *simp*
then show *?thesis* **by** *simp*
qed
then have *Abs* (*?dist-range 0* (*Suc n*)) $\rightarrow_\eta$ (*?dist-range 1* (*Suc n*))[*Var 0/0*]
unfolding *dist-last* **by** (*rule eta*)
also have ... = *var-dist* (*rev* [*0..<n*]) ((*liftn* (*Suc n*) *t 0*)[*Var 0/0*]) **proof** −
have ∀ *v*∈*set* (*rev* [*1..<Suc n*]). *0* < *v* **by** *auto*
moreover have *rev* [*0..<n*] = *map* (λ*v*. *v* − *1*) (*rev* [*1..<Suc n*]) **by** (*induction n*) *simp-all*
ultimately show *?thesis* **by** (*simp only*: *var-dist-subst-lt*)
qed
also have ... = *?dist-range 0 n* **using** *subst-liftn*[*of 0 n 0 t Var 0*] **by** *simp*
finally have *Abs* (*?dist-range 0* (*Suc n*)) ↔ *?dist-range 0 n* **..**
then have $(Abs^{\frown\frown}Suc\ n)$ (*?dist-range 0* (*Suc n*)) ↔ $(Abs^{\frown\frown}n)$ (*?dist-range 0 n*)
unfolding *funpow-Suc-inside* **by** (*rule absn-cong*)
also from *Suc.IH* **have** ... ↔ *t* **.**
finally show *?case* **.**
qed

primrec *strip-context* :: *nat* ⇒ *dB* ⇒ *nat* ⇒ *dB*
where
strip-context n (*Var i*) *k* = (*if i* < *k then Var i else Var* (*i* − *n*))
| *strip-context n* (*Abs t*) *k* = *Abs* (*strip-context n t* (*Suc k*))
| *strip-context n* (*s* ∘ *t*) *k* = *strip-context n s k* ∘ *strip-context n t k*

lemma *strip-context-liftn*: *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 ∀ *i*∈*frees t*. *i* < *k* ∨ *k* + *n* ≤ *i*
    shows *liftn n* (*strip-context n t k*) *k* = *t*
using *assms* **proof** (*induction t arbitrary*: *k*)
  case (*Abs t*)
  have ∀ *i*∈*frees t*. *i* < *Suc k* ∨ *Suc k* + *n* ≤ *i* **proof**
    fix *i* **assume** *free*: *i* ∈ *frees t*
    show *i* < *Suc k* ∨ *Suc k* + *n* ≤ *i* **proof** (*cases i* > *0*)
      assume *i* > *0*
      with *free Abs.prems* **have** *i*−*1* < *k* ∨ *k* + *n* ≤ *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 ∀ *i*∈*frees t*. *n* ≤ *i*
  shows (*Abs*⁀*n*) (*var-dist* (*rev* [*0*..<*n*]) *t*) ↔ *strip-context n t 0* (**is** *?lhs t* ↔ *?rhs*)
proof −
  have *?lhs* (*liftn n ?rhs 0*) ↔ *?rhs* **by** (*rule absn-dist-eta*)
  moreover have *liftn n ?rhs 0* = *t*
    using *assms* **by** (*auto intro*: *liftn-strip-context*)
  ultimately show *?thesis* **by** *simp*
qed

definition *perm-vars* :: *nat* ⇒ *nat list* ⇒ *bool*
where *perm-vars n vs* ⟷ *distinct vs* ∧ *set vs* = {*0*..<*n*}

lemma *perm-vars-distinct*: *perm-vars n vs* ⟹ *distinct vs*
unfolding *perm-vars-def* **by** *simp*

lemma *perm-vars-length*: *perm-vars n vs* ⟹ *length vs* = *n*
unfolding *perm-vars-def* **using** *distinct-card* **by** *force*

lemma *perm-vars-lt*: *perm-vars n vs* ⟹ ∀ *i*∈*set vs*. *i* < *n*
unfolding *perm-vars-def* **by** *simp*

lemma *perm-vars-nth-lt*: *perm-vars n vs* ⟹ *i* < *n* ⟹ *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 $i \in \{0..<n\}$ **and** $j \in \{0..<n\}$
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** $\bigwedge t.\ \forall\, i \in$*frees t*. $n \le i \Longrightarrow$
(*Abs*^^*n*) (*var-dist vs′* ((*Abs*^^*n*) (*var-dist vs* (*liftn n t 0*)))) ↔ *strip-context n t 0*
proof −
define *vsubsts* **where** *vsubsts n vs′ vs* =
map (λ*v*.
if v < *n* − *length vs′ then v*
else if v < *n then vs′* ! (*n* − *v* − *1*) + (*n* − *length vs′*)
else v − *length vs′*) *vs* **for** *n vs′ vs*

let *?app-vars* = λ*t n vs′ vs*. *var-dist vs′* ((*Abs*^^*n*) (*var-dist vs* (*liftn n t 0*)))
{
fix *t* :: *dB* **and** *vs′* :: *nat list*
assume *partial*: *length vs′* ≤ *n*

let *?m* = *n* − *length vs′*
have *?app-vars t n vs′ vs* ↔ (*Abs*^^*?m*) (*var-dist* (*vsubsts n vs′ vs*) (*liftn ?m t 0*))
using *partial* **proof** (*induction vs′ arbitrary*: *vs n*)
case *Nil*
then have *vsubsts n* [] *vs* = *vs* **unfolding** *vsubsts-def* **by** (*auto intro*: *map-idI*)
then show *?case* **by** *simp*
next
case (*Cons v vs′*)
define *n′* **where** *n′* = *n* − *1*
have *Suc-n′*: *Suc n′* = *n* **unfolding** *n′-def* **using** *Cons.prems* **by** *simp*
have *vs′-length*: *length vs′* ≤ *n′* **unfolding** *n′-def* **using** *Cons.prems* **by** *simp*
let *?m′* = *n′* − *length vs′*

```
    have m'-conv: ?m' = n − length (v # vs') unfolding n'-def by simp

    have ?app-vars t n (v # vs') vs = ?app-vars t (Suc n') (v # vs') vs
      unfolding Suc-n' ..
    also have ... ↔ var-dist vs' ((Abs^^Suc n') (var-dist vs (liftn (Suc n') t 0))
° Var v)
      unfolding var-dist-Cons ..
   also have ... ↔ ?app-vars t n' vs' (vsubsts n [v] vs) proof (rule var-dist-cong)
      have map (λvv. vsubst vv (v + n') n') vs = vsubsts n [v] vs
        unfolding Suc-n'[symmetric] vsubsts-def vsubst-def
        by (auto cong: if-cong)
      then have (var-dist vs (liftn (Suc n') t 0))[liftn n' (Var v) 0/n']
            = var-dist (vsubsts n [v] vs) (liftn n' t 0)
        using var-dist-subst-Var subst-liftn by simp
      then show (Abs^^Suc n') (var-dist vs (liftn (Suc n') t 0)) ° Var v
            ↔ (Abs^^n') (var-dist (vsubsts n [v] vs) (liftn n' t 0))
        by (fastforce intro: absn-beta-equiv[THEN term-trans])
    qed
    also have ... ↔ (Abs^^?m') (var-dist (vsubsts n' vs' (vsubsts n [v] vs)) (liftn
?m' t 0))
      using vs'-length Cons.IH by blast
    also have ... = (Abs^^?m') (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 (λi. n − perm-vars-inv n vs (n − i − 1) − 1)
[0..<n]

 from perm-vars have vs-length: length vs = n by (rule perm-vars-length)
 have vs'-length: length vs' = n unfolding vs'-def by simp

 have map (λv. vs' ! (n − v − 1)) vs = rev [0..<n] proof −
   have length vs = length (rev [0..<n])
     unfolding vs-length by simp
   then have list-all2 (λv v'. vs' ! (n − v − 1) = v') vs (rev [0..<n]) proof
     fix i assume i < length vs
     then have i < n unfolding vs-length .
     then have vs ! i < n using perm-vars perm-vars-nth-lt by simp
     with ‹i < n› have vs' ! (n − vs ! i − 1) = n − perm-vars-inv n vs (vs ! i)
− 1
```

```
      unfolding vs'-def by simp
    also from ‹i < n› have ... = n − i − 1 using perm-vars perm-vars-inv-nth
by simp
    also from ‹i < n› have ... = rev [0..<n] ! i by (simp add: rev-nth)
    finally show vs' ! (n − vs ! i − 1) = rev [0..<n] ! i .
  qed
  then show ?thesis
    unfolding list.rel-eq[symmetric]
    using list.rel-map
    by auto
 qed
 then have vs'-vs: 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 = λt n. var-dist (rev [0..<n]) t
 {
  fix t
  assume not-free: ∀ i∈frees t. n ≤ i
  have ?app-vars t n vs' vs ↔ ?appd-vars t n for t
   using partial-appd[of vs'] vs'-length vs'-vs by simp
  then have (Abs^^n) (?app-vars t n vs' vs) ↔ (Abs^^n) (?appd-vars t n)
   by (rule absn-cong)
  also have ... ↔ strip-context n t 0
   using not-free by (rule absn-dist-eta-free)
  finally have (Abs^^n) (?app-vars t n vs' vs) ↔ strip-context n t 0 .
 }
 with that show ?thesis .
qed

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

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

lemma liftn-context-conv: k ≤ k' ⟹ ∀ i∈frees t. i < k ∨ k' ≤ i ⟹ liftn n t k =
liftn n t k'
proof (induction t arbitrary: k k')
 case (Abs t)
 have ∀ i∈frees t. i < Suc k ∨ Suc k' ≤ i proof
  fix i assume i ∈ frees t
  show i < Suc k ∨ Suc k' ≤ i proof (cases i = 0)
   assume i = 0 then show ?thesis by simp
  next
   assume i ≠ 0
   from Abs.prems(2) have ∀ i. free t (Suc i) ⟶ i < k ∨ k' ≤ i by auto
```

```
    then have ∀ i. 0 < i ∧ free t i ⟶ i − 1 < k ∨ k′ ≤ i − 1 by simp
    then have ∀ i. 0 < i ∧ free t i ⟶ i < Suc k ∨ Suc k′ ≤ i by auto
    with ‹i ≠ 0› ‹i ∈ frees t› show ?thesis by simp
   qed
  qed
  with Abs.IH Abs.prems(1) show ?case by auto
qed auto

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

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

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

General notion of bracket abstraction for lambda terms **definition**
foldr-option :: $('a \Rightarrow 'b \Rightarrow 'b\ option) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b\ option$
where *foldr-option f xs e* = *foldr* ($\lambda a\ b.$ *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* $s' \Longrightarrow s' \circ Var\ i \leftrightarrow s$
assumes *bracket-frees*: *term-bracket i s* = *Some* $s' \Longrightarrow$ *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* (*v*#*vs*) *t* = *Some* t'
obtains s' **where** *term-brackets vs t* = *Some* s' **and** *term-bracket v* s' = *Some* t'
using *assms* **unfolding** *term-brackets-def* **by** (*elim foldr-option-Cons-SomeE*)

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

lemma *term-brackets-dist*:
assumes *term-brackets vs t* = *Some* t'
shows *var-dist vs* $t' \leftrightarrow t$
proof −
from *assms* **have** $\forall t''.\ t' \leftrightarrow t'' \longrightarrow$ *var-dist vs* $t'' \leftrightarrow t$
proof (*induction vs arbitrary*: t')
case *Nil* **then show** *?case* **by** (*simp add*: *term-sym*)
next

case (*Cons v vs*)
from *Cons.prems* **obtain** *u* **where**
inner: *term-brackets vs t* = *Some u* **and**
step: *term-bracket v u* = *Some t′*
by (*auto elim*: *term-brackets-Cons-SomeE*)
from *step* **have** *red1*: *t′* ∘ *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
qed
then show *?thesis* **by** *blast*
qed

end

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

lemma *strip-unlift-vars*:
assumes *opaque x* = []
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* (λ*i y*. *y* ⋄ *Opaque i*)

lemma *opaque-dist-cong*: *x* ≃⁺ *y* ⟹ *opaque-dist vs x* ≃⁺ *opaque-dist vs y*
unfolding *opaque-dist-def*
by (*induction vs arbitrary*: *x y*) (*simp-all add*: *ap-congL*)

lemma *itrm-brackets-dist*:
  assumes *defined*: *itrm-brackets vs x* = *Some x′*
    shows *opaque-dist vs x′* ≃⁺ *x*
proof −
  define *x″* **where** *x″* = *x′*
  have *x′* ≃⁺ *x″* **unfolding** *x″-def* **..**
  with *defined* **show** *opaque-dist vs x″* ≃⁺ *x*
    unfolding *opaque-dist-def*
    proof (*induction vs arbitrary*: *x′ x″*)
        case *Nil* **then show** *?case* **unfolding** *itrm-brackets-def* **by** (*simp add*: *itrm-sym*)
    next
      case (*Cons v vs*)
      from *Cons.prems*(*1*) **obtain** *y′*
        where *defined′*: *itrm-brackets vs x* = *Some y′*
          and *itrm-bracket v y′* = *Some x′*
        by (*rule itrm-brackets-Cons-SomeE*)
      then have *x′* ⋄ *Opaque v* ≃⁺ *y′* **by** (*elim itrm-bracket-ap*)
      then have *x″* ⋄ *Opaque v* ≃⁺ *y′*
        using *Cons.prems*(*2*) **by** (*blast intro*: *itrm-sym itrm-trans*)
      note *this*[*symmetric*]
      with *defined′* **have** *fold* (λ*i y*. *y* ⋄ *Opaque i*) *vs* (*x″* ⋄ *Opaque v*) ≃⁺ *x*
        using *Cons.IH* **by** *blast*
      then show *?case* **by** *simp*
    qed
qed

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

lemma *itrm-brackets-all*:
  assumes *all-opaque*: *set* (*opaque x*) ⊆ *set vs*
      and *defined*: *itrm-brackets vs x* = *Some x′*

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

lemma *itrm-brackets-all-unlift-vars*:
assumes *all-opaque*: *set* (*opaque* x) ⊆ *set vs*
and *defined*: *itrm-brackets vs* x = *Some* x'
shows $x' \simeq^+$ *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 *Some* (*unlift-vars* n x') = *map-option* (*unlift-vars* n) (*itrm-bracket* v y')
unfolding *iv* **by** *simp*
moreover have *set* (*opaque* y') ⊆ {0..<n}
using *all-opaque ivs* **by** (*auto dest*: *itrm-brackets-opaque*)

```
    moreover have v < n using Cons.prems by simp
    ultimately have term-bracket v (unlift-vars n y') = Some (unlift-vars n x')
      using bracket-compat by auto
  }
  ultimately show ?case by (rule term-brackets-ConsI)
qed

theorem bracket-lifting:
  assumes all-vars: set (opaque x) ∪ set (opaque y) ⊆ {0..<n}
      and perm-vars: perm-vars n vs
      and defined: itrm-brackets vs x = Some x' itrm-brackets vs y = Some y'
      and base-eq: (Abs^^n) (unlift-vars n x) ↔ (Abs^^n) (unlift-vars n y)
    shows x ≃+ y
proof −
  from perm-vars have set-vs: set vs = {0..<n}
    unfolding perm-vars-def by simp

  have x-swap: term-brackets vs (unlift-vars n x) = Some (unlift-vars n x')
    using all-vars set-vs defined(1) by (auto intro: brackets-unlift-vars-swap)
  have y-swap: term-brackets vs (unlift-vars n y) = Some (unlift-vars n y')
    using all-vars set-vs defined(2) by (auto intro: brackets-unlift-vars-swap)

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

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

  have x ≃+ opaque-dist vs x'
    using defined(1) by (rule itrm-brackets-dist[symmetric])
  also have ... ≃+ opaque-dist vs (Pure (unlift-vars 0 x'))
    using all-vars set-vs defined(1)
    by (auto intro: opaque-dist-cong itrm-brackets-all-unlift-vars)
  also have ... ≃+ opaque-dist vs (Pure (unlift-vars 0 y'))
  proof (rule opaque-dist-cong, rule pure-cong)
    have (Abs^^n) (var-dist vs (unlift-vars n x')) ↔ (Abs^^n) (unlift-vars n x)
      using x-swap term-brackets-dist by auto
    also have ... ↔ (Abs^^n) (unlift-vars n y) using base-eq .
    also have ... ↔ (Abs^^n) (var-dist vs (unlift-vars n y'))
      using y-swap term-brackets-dist[THEN term-sym] by auto
    finally have strip-context n (unlift-vars n x') 0 ↔ strip-context n (unlift-vars
n y') 0
      using perm-vars ux-frees uy-frees
```

```
    by (intro dist-perm-eta-equiv)
  then show unlift-vars 0 x' ↔ unlift-vars 0 y'
    using strip-unlift-vars complete-x complete-y by simp
 qed
 also have ... ≃+ opaque-dist vs y' proof (rule opaque-dist-cong)
  show Pure (unlift-vars 0 y') ≃+ y'
   using all-vars set-vs defined(2) itrm-brackets-all-unlift-vars[THEN itrm-sym]
   by blast
 qed
 also have ... ≃+ y using defined(2) by (rule itrm-brackets-dist)
 finally show ?thesis .
qed

end

end
```

References

[1] J. Gibbons and R. Bird. Be kind, rewind: A modest proposal about traversal. May 2012.

[2] J. Gibbons and R. Hinze. Just do it: Simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, pages 2–14. ACM, 2011.

[3] R. Hinze. Lifting operators and laws. 2010.

[4] G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. In *Trends in Functional Programming (TFP 2008)*, 2008.

[5] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.