Applicative Lifting

Andreas Lochbihler Joshua Schneider

March 17, 2025

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*: $(\bigwedge x.\ x \in A \Longrightarrow R\ (f\ x)\ (g\ x)) \Longrightarrow rel\text{-}fun\ (eq\text{-}on\ A)\ R\ f\ g$
⟨*proof*⟩

lemma *rel-fun-map-fun2*: $rel\text{-}fun\ (eq\text{-}on\ (range\ h))\ A\ f\ g \Longrightarrow rel\text{-}fun\ (BNF\text{-}Def.Grp\ UNIV\ h)^{-1-1}\ A\ f\ (map\text{-}fun\ h\ id\ g)$
⟨*proof*⟩

lemma *rel-fun-refl-eq-onp*:
$(\bigwedge z.\ z \in f\ `\ X \Longrightarrow A\ z\ z) \Longrightarrow rel\text{-}fun\ (eq\text{-}on\ X)\ A\ f\ f$
⟨*proof*⟩

lemma *eq-onE*: $\llbracket\ eq\text{-}on\ X\ a\ b;\ \llbracket\ b \in X;\ a = b\ \rrbracket \Longrightarrow thesis\ \rrbracket \Longrightarrow thesis$ ⟨*proof*⟩

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

1.2 Proof automation

lemma *arg1-cong*: $x = y \Longrightarrow f\ x\ z = f\ y\ z$
⟨*proof*⟩

lemma *UNIV-E*: $x \in UNIV \Longrightarrow P \Longrightarrow P$ ⟨*proof*⟩

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*
⟨*proof*⟩

⟨*ML*⟩

lemma [*combinator-eq*]: *B* ≡ *S* (*K S*) *K* ⟨*proof*⟩
lemma [*combinator-eq*]: *C* ≡ *S* (*S* (*K* (*S* (*K S*) *K*)) *S*) (*K K*) ⟨*proof*⟩
lemma [*combinator-eq*]: *I* ≡ *W K* ⟨*proof*⟩
lemma [*combinator-eq*]: *I* ≡ *C K* () ⟨*proof*⟩
lemma [*combinator-eq*]: *S* ≡ *B* (*B W*) (*B B C*) ⟨*proof*⟩
lemma [*combinator-eq*]: *T* ≡ *C I* ⟨*proof*⟩
lemma [*combinator-eq*]: *W* ≡ *S S* (*S K*) ⟨*proof*⟩

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*
⟨*proof*⟩

end

⟨*ML*⟩

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
begin

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

adhoc-overloading *Applicative.pure* $\rightleftharpoons$ *const*
adhoc-overloading *Applicative.ap* $\rightleftharpoons$ *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*
$\langle proof \rangle$

lemma
  includes *applicative-syntax*
  shows *const* $(\lambda f\ x\ y.\ f\ y\ x) \diamond f \diamond x \diamond y = f \diamond y \diamond x$
$\langle proof \rangle$

end

2.2 Option

theory *Applicative-Option* **imports**
  Applicative
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* $\rightleftharpoons$ *pure-option*
adhoc-overloading *Applicative.ap* $\rightleftharpoons$ *ap-option*

lemma *some-ap-option*: *ap-option* (*Some* f) x = *map-option* $f\ x$
$\langle proof \rangle$

lemma *ap-some-option*: *ap-option* f (*Some* x) = *map-option* $(\lambda g.\ g\ x)\ f$
$\langle proof \rangle$

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*
⟨*proof*⟩

applicative *option* (C, W)
for
pure: *Some*
ap: *ap-option*
rel: *rel-option*
set: *set-option*
⟨*proof*⟩
include *applicative-syntax*
⟨*proof*⟩

lemma *map-option-ap-conv*[*applicative-unfold*]: *map-option* f x = *ap-option* (*pure* f) x
⟨*proof*⟩

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

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

abbreviation (*input*) *pure-sum* **where** *pure-sum* $\equiv$ *Inl*
adhoc-overloading *Applicative.pure* $\rightleftharpoons$ *pure-sum*
adhoc-overloading *Applicative.ap* $\rightleftharpoons$ *ap-either*

lemma *ap-sum-id*: *ap-sum c* (*Inl id*) $x = x$
⟨*proof*⟩

lemma *ap-sum-ichng*: *ap-sum c f* (*Inl x*) = *ap-sum c* (*Inl* ($\lambda f.\ f\ x$)) *f*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *semigroup-const*: *semigroup* ($\lambda x\ y.\ x$)
⟨*proof*⟩

locale *either-af* =
fixes $B :: 'b \Rightarrow 'b \Rightarrow bool$
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*
⟨*proof*⟩

end

interpretation *either-af* (=) ⟨*proof*⟩

applicative *semigroup-sum*
for
pure: *Inl*
ap: *ap-plus*
⟨*proof*⟩

no-adhoc-overloading *Applicative.pure* $\rightleftharpoons$ *pure-sum*

end

2.4 Set with Cartesian product

theory *Applicative-Set* **imports**
Applicative

begin

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

adhoc-overloading *Applicative.ap* $\rightleftharpoons$ *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*
⟨*proof*⟩

applicative *set* (C)
for
  pure: $\lambda x.\ \{x\}$
  ap: *ap-set*
  rel: *rel-set*
  set: $\lambda x.\ x$
⟨*proof*⟩

end

2.5 Lists

theory *Applicative-List* **imports**
  Applicative
begin

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

adhoc-overloading *Applicative.ap* $\rightleftharpoons$ *ap-list*

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

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

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*
⟨*proof*⟩

context includes *applicative-syntax*
begin

lemma *cons-ap-list*: (f # *fs*) $\diamond$ *xs* = *map f xs* @ *fs* $\diamond$ *xs*
⟨*proof*⟩

lemma *append-ap-distrib*: (*fs* @ *gs*) $\diamond$ *xs* = *fs* $\diamond$ *xs* @ *gs* $\diamond$ *xs*
⟨*proof*⟩

applicative *list*
for
  pure: $\lambda x.$ $[x]$
  ap: *ap-list*
  rel: *list-all2*
  set: *set*
$\langle proof \rangle$

lemma *map-ap-conv*[*applicative-unfold*]: *map f x* = [*f*] $\diamond$ *x*
$\langle proof \rangle$

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* = [])
$\langle proof \rangle$

lemma *zip-eq-Nil-iff* [*simp*]: *zip xs ys* = [] $\longleftrightarrow$ *xs* = [] $\vee$ *ys* = []
$\langle proof \rangle$

lemma *remdups-append1*: *remdups* (*remdups xs* @ *ys*) = *remdups* (*xs* @ *ys*)
$\langle proof \rangle$

lemma *remdups-append2*: *remdups* (*xs* @ *remdups ys*) = *remdups* (*xs* @ *ys*)
$\langle proof \rangle$

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

lemma *remdups-concat-map*: *remdups* (*concat* (*map remdups xss*)) = *remdups* (*concat xss*)
$\langle proof \rangle$

lemma *remdups-concat-remdups*: *remdups* (*concat* (*remdups xss*)) = *remdups* (*concat xss*)
$\langle proof \rangle$

lemma *remdups-replicate*: *remdups* (*replicate n x*) = (*if n* = *0 then* [] *else* [*x*])
$\langle proof \rangle$

typedef $'a$ *dnelist* = $\{xs::'a\ list.\ distinct\ xs \wedge xs \neq []\}$
morphisms *list-of-dnelist Abs-dnelist*
⟨*proof*⟩

setup-lifting *type-definition-dnelist*

lemma *dnelist-subtype-dlist*:
type-definition $(\lambda x.\ Dlist\ (list\text{-}of\text{-}dnelist\ x))\ (\lambda x.\ Abs\text{-}dnelist\ (list\text{-}of\text{-}dlist\ x))\ \{xs.\ xs \neq Dlist.empty\}$
⟨*proof*⟩

lift-bnf (*no-warn-transfer*, *no-warn-wits*) $'a$ *dnelist via dnelist-subtype-dlist* **for** *map*: *map*
⟨*proof*⟩
hide-const (**open**) *map*

context begin
qualified lemma *map-def*: *Applicative-DNEList.map* = *map-fun id* (*map-fun list-of-dnelist Abs-dnelist*) $(\lambda f\ xs.\ remdups\ (list.map\ f\ xs))$
⟨*proof*⟩ **lemma** *map-transfer* [*transfer-rule*]:
rel-fun $(=)$ (*rel-fun* (*pcr-dnelist* $(=)$) (*pcr-dnelist* $(=)$)) $(\lambda f\ xs.\ remdups\ (map\ f\ xs))$ *Applicative-DNEList.map*
⟨*proof*⟩ **lift-definition** *single* :: $'a \Rightarrow 'a$ *dnelist* **is** $\lambda x.\ [x]$ ⟨*proof*⟩ **lift-definition** *insert* :: $'a \Rightarrow 'a$ *dnelist* $\Rightarrow 'a$ *dnelist* **is** $\lambda x\ xs.$ *if* $x \in set\ xs$ *then xs else* $x\ \#\ xs$
⟨*proof*⟩ **lift-definition** *append* :: $'a$ *dnelist* $\Rightarrow 'a$ *dnelist* $\Rightarrow 'a$ *dnelist* **is** $\lambda xs\ ys.$ *remdups* (*xs* @ *ys*) ⟨*proof*⟩ **lift-definition** *bind* :: $'a$ *dnelist* $\Rightarrow ('a \Rightarrow 'b$ *dnelist*$) \Rightarrow$ $'b$ *dnelist* **is** $\lambda xs\ f.$ *remdups* (*List.bind xs f*) ⟨*proof*⟩

abbreviation (*input*) *pure-dnelist* :: $'a \Rightarrow 'a$ *dnelist*
where *pure-dnelist* $\equiv$ *single*

end

lift-definition *ap-dnelist* :: $('a \Rightarrow 'b)$ *dnelist* $\Rightarrow 'a$ *dnelist* $\Rightarrow 'b$ *dnelist*
is $\lambda f\ x.$ *remdups* (*ap-list f x*)
⟨*proof*⟩

adhoc-overloading *Applicative.ap* $\rightleftharpoons$ *ap-dnelist*

lemma *ap-pure-list* [*simp*]: *ap-list* $[f]$ *xs* = *map f xs*
⟨*proof*⟩

context includes *applicative-syntax*
begin

lemma *ap-pure-dlist*: *pure-dnelist f* $\diamond$ *x* = *Applicative-DNEList.map f x*
⟨*proof*⟩

applicative *dnelist* (K)
for *pure*: *pure-dnelist*
ap: *ap-dnelist*
⟨*proof*⟩

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

context begin
private lift-definition x :: *int dnelist* **is** [*2*,*3*] ⟨*proof*⟩ **lift-definition** y :: *int dnelist* **is** [*5*,*7*] ⟨*proof*⟩ **lemma** *pure-dnelist* $(\lambda f\ x\ y.\ f\ y\ x)$ ⋄ *pure-dnelist* $((*))$ ⋄ x ⋄ $y \neq$ *pure-dnelist* $((*))$ ⋄ y ⋄ x
⟨*proof*⟩
end

end

end

3.1 Monoid

theory *Applicative-Monoid* **imports**
Applicative
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*)

⟨*ML*⟩

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*
⟨*proof*⟩

applicative *comm-monoid-add* (C)
for *pure*: *pure-monoid-add* :: - ⇒ (- :: *comm-monoid-add*, -) *monoid-ap*
ap: *ap-monoid-add* :: (- :: *comm-monoid-add*, -) *monoid-ap* ⇒ -
⟨*proof*⟩

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* ⇒ -
⟨*proof*⟩

Test case

lemma
  includes *applicative-syntax*
  shows *pure-monoid-add* (+) ⋄ (x :: (*nat*, *int*) *monoid-ap*) ⋄ y = *pure* (+) ⋄ y ⋄ x
⟨*proof*⟩

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* ($\lambda(f, x).\ f\ x$) (*prod-filter* F X)

lemma *eq-on-UNIV*: *eq-on UNIV* = (=)
  ⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

end

3.4 Streams as an applicative functor

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

primcorec (*transfer*) *ap-stream* :: $('a \Rightarrow 'b)$ *stream* $\Rightarrow$ $'a$ *stream* $\Rightarrow$ $'b$ *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* **and** *applicative-syntax*
begin

lemma *ap-stream-id*: *pure* $(\lambda x.\ x) \diamond x = x$
⟨*proof*⟩

lemma *ap-stream-homo*: *pure* $f \diamond$ *pure* $x =$ *pure* $(f\ x)$
⟨*proof*⟩

lemma *ap-stream-interchange*: $f \diamond$ *pure* $x =$ *pure* $(\lambda f.\ f\ x) \diamond f$
⟨*proof*⟩

lemma *ap-stream-composition*: *pure* $(\lambda g\ f\ x.\ g\ (f\ x)) \diamond g \diamond f \diamond x = g \diamond (f \diamond x)$
⟨*proof*⟩

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

set: *sset*
⟨*proof*⟩

lemma *smap-applicative*[*applicative-unfold*]: $smap\ f\ x = pure\ f \diamond x$
⟨*proof*⟩

lemma *smap2-applicative*[*applicative-unfold*]: $smap2\ f\ x\ y = pure\ f \diamond x \diamond y$
⟨*proof*⟩

end

end

3.5 Open state monad

theory *Applicative-Open-State* **imports**
Applicative
begin

type-synonym $('a, 's)\ state = 's \Rightarrow 'a \times 's$

definition $ap\text{-}state\ f\ x = (\lambda s.\ case\ f\ s\ of\ (g, s') \Rightarrow case\ x\ s'\ of\ (y, s'') \Rightarrow (g\ y, s''))$

abbreviation (*input*) $pure\text{-}state \equiv Pair$

adhoc-overloading $Applicative.ap \rightleftharpoons ap\text{-}state$

applicative *state*
for
pure: *pure-state*
ap: $ap\text{-}state :: ('a \Rightarrow 'b, 's)\ state \Rightarrow ('a, 's)\ state \Rightarrow ('b, 's)\ state$
⟨*proof*⟩

end

3.6 Probability mass functions

theory *Applicative-PMF* **imports**
Applicative
HOL−Probability.Probability
begin

abbreviation (*input*) $pure\text{-}pmf :: 'a \Rightarrow 'a\ pmf$
where $pure\text{-}pmf \equiv return\text{-}pmf$

definition $ap\text{-}pmf :: ('a \Rightarrow 'b)\ pmf \Rightarrow 'a\ pmf \Rightarrow 'b\ pmf$
where $ap\text{-}pmf\ f\ x = map\text{-}pmf\ (\lambda(f, x).\ f\ x)\ (pair\text{-}pmf\ f\ x)$

adhoc-overloading $Applicative.ap \rightleftharpoons ap\text{-}pmf$

context includes *applicative-syntax*
begin

lemma *ap-pmf-id*: *pure-pmf* $(\lambda x.\ x) \diamond x = x$
⟨*proof*⟩

lemma *ap-pmf-comp*: *pure-pmf* $(\circ) \diamond u \diamond v \diamond w = u \diamond (v \diamond w)$
⟨*proof*⟩

lemma *ap-pmf-homo*: *pure-pmf* $f \diamond$ *pure-pmf* $x =$ *pure-pmf* $(f\ x)$
⟨*proof*⟩

lemma *ap-pmf-interchange*: $u \diamond$ *pure-pmf* $x =$ *pure-pmf* $(\lambda f.\ f\ x) \diamond u$
⟨*proof*⟩

lemma *ap-pmf-K*: *return-pmf* $(\lambda x\ \text{-}.\ x) \diamond x \diamond y = x$
⟨*proof*⟩

lemma *ap-pmf-C*: *return-pmf* $(\lambda f\ x\ y.\ f\ y\ x) \diamond f \diamond x \diamond y = f \diamond y \diamond x$
⟨*proof*⟩

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*
⟨*proof*⟩

applicative *pmf* (*C*, *K*)
for
pure: *pure-pmf*
ap: *ap-pmf*
rel: *rel-pmf*
set: *set-pmf*
⟨*proof*⟩

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* ($\lambda x.$ *sum-list* (*f* *x*)) *xs*)
⟨*proof*⟩

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

lemma *set-ap-list* [*simp*]: $set\ (f \diamond x) = (\lambda(f,\ x).\ f\ x)\ `\ (set\ f \times set\ x)$
⟨*proof*⟩

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 $'a\ pfp = \{xs :: ('a \times real)\ list.\ (\forall (\text{-},\ p) \in set\ xs.\ p > 0) \wedge sum\text{-}list\ (map\ snd\ xs) = 1\}$
⟨*proof*⟩

setup-lifting *type-definition-pfp*

lift-definition *pure-pfp* :: $'a \Rightarrow 'a\ pfp$ **is** $\lambda x.\ [(x,\ 1)]$ ⟨*proof*⟩

lift-definition *ap-pfp* :: $('a \Rightarrow 'b)\ pfp \Rightarrow 'a\ pfp \Rightarrow 'b\ pfp$
is $\lambda fs\ xs.\ [\lambda(f,\ p)\ (x,\ q).\ (f\ x,\ p * q)] \diamond fs \diamond xs$
⟨*proof*⟩

adhoc-overloading *Applicative.ap* ⇌ *ap-pfp*

applicative *pfp*
 for *pure*: *pure-pfp*
 ap: *ap-pfp*
⟨*proof*⟩

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*⟩

end

theory *Applicative-Vector* **imports**

Applicative
HOL−Analysis.Finite-Cartesian-Product
begin

definition *pure-vec* :: $'a \Rightarrow ('a,\ 'b :: \mathit{finite})\ \mathit{vec}$
where *pure-vec* $x = (\chi\ \text{-}\ .\ x)$

definition *ap-vec* :: $('a \Rightarrow 'b,\ 'c :: \mathit{finite})\ \mathit{vec} \Rightarrow ('a,\ 'c)\ \mathit{vec} \Rightarrow ('b,\ 'c)\ \mathit{vec}$
where *ap-vec* $f\ x = (\chi\ i.\ (f\ \$\ i)\ (x\ \$\ i))$

adhoc-overloading *Applicative.ap* $\rightleftharpoons$ *ap-vec*

applicative *vec* $(K,\ W)$
for
pure: *pure-vec*
ap: *ap-vec*
⟨*proof*⟩

lemma *pure-vec-nth* [*simp*]: *pure-vec* $x\ \$\ i = x$
⟨*proof*⟩

lemma *ap-vec-nth* [*simp*]: *ap-vec* $f\ x\ \$\ i\ = (f\ \$\ i)\ (x\ \$\ i)$
⟨*proof*⟩

end

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

print-applicative

end

4 Examples of applicative lifting

4.1 Algebraic operations for the environment functor

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

Link between applicative instance of the environment functor with the point-wise operations for the algebraic type classes

context includes *applicative-syntax*
begin

lemma *plus-fun-af* [*applicative-unfold*]: $f + g = pure\ (+) \diamond f \diamond g$
⟨*proof*⟩

lemma *zero-fun-af* [*applicative-unfold*]: $0 = pure\ 0$
⟨*proof*⟩

lemma *times-fun-af* [*applicative-unfold*]: $f * g = pure\ (*) \diamond f \diamond g$
⟨*proof*⟩

lemma *one-fun-af* [*applicative-unfold*]: $1 = pure\ 1$
⟨*proof*⟩

lemma *of-nat-fun-af* [*applicative-unfold*]: $\textit{of-nat}\ n = pure\ (\textit{of-nat}\ n)$
⟨*proof*⟩

lemma *inverse-fun-af* [*applicative-unfold*]: $inverse\ f = pure\ inverse \diamond f$
⟨*proof*⟩

lemma *divide-fun-af* [*applicative-unfold*]: $divide\ f\ g = pure\ divide \diamond f \diamond g$
⟨*proof*⟩

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 ⟨*proof*⟩
end

instantiation *stream* :: (*one*) *one* **begin**
definition [*applicative-unfold*]: *1* = *sconst 1*
instance ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
end

instance *stream* :: (*Rings.dvd*) *Rings.dvd* ⟨*proof*⟩

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 ⟨*proof*⟩
end

instance *stream* :: (*semigroup-add*) *semigroup-add*
⟨*proof*⟩

instance *stream* :: (*ab-semigroup-add*) *ab-semigroup-add*
⟨*proof*⟩

instance *stream* :: (*semigroup-mult*) *semigroup-mult*
⟨*proof*⟩

instance *stream* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
⟨*proof*⟩

instance *stream* :: (*monoid-add*) *monoid-add*
⟨*proof*⟩

instance *stream* :: (*comm-monoid-add*) *comm-monoid-add*
⟨*proof*⟩

instance *stream* :: (*comm-monoid-diff*) *comm-monoid-diff*
⟨*proof*⟩

instance *stream* :: (*monoid-mult*) *monoid-mult*
⟨*proof*⟩

instance *stream* :: (*comm-monoid-mult*) *comm-monoid-mult*
⟨*proof*⟩

lemma *plus-stream-shd*: $shd\ (x + y) = shd\ x + shd\ y$
⟨*proof*⟩

lemma *plus-stream-stl*: $stl\ (x + y) = stl\ x + stl\ y$
⟨*proof*⟩

instance *stream* :: (*cancel-semigroup-add*) *cancel-semigroup-add*
⟨*proof*⟩

instance *stream* :: (*cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
⟨*proof*⟩

instance *stream* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add* ⟨*proof*⟩

instance *stream* :: (*group-add*) *group-add*
⟨*proof*⟩

instance *stream* :: (*ab-group-add*) *ab-group-add*
⟨*proof*⟩

instance *stream* :: (*semiring*) *semiring*
⟨*proof*⟩

instance *stream* :: (*mult-zero*) *mult-zero*
⟨*proof*⟩

instance *stream* :: (*semiring-0*) *semiring-0* ⟨*proof*⟩

instance *stream* :: (*semiring-0-cancel*) *semiring-0-cancel* ⟨*proof*⟩

instance *stream* :: (*comm-semiring*) *comm-semiring*
⟨*proof*⟩

instance *stream* :: (*comm-semiring-0*) *comm-semiring-0* ⟨*proof*⟩

instance *stream* :: (*comm-semiring-0-cancel*) *comm-semiring-0-cancel* ⟨*proof*⟩

lemma *pure-stream-inject* [*simp*]: *sconst* x = *sconst* $y \longleftrightarrow x = y$
⟨*proof*⟩

instance *stream* :: (*zero-neq-one*) *zero-neq-one*
⟨*proof*⟩

instance *stream* :: (*semiring-1*) *semiring-1* ⟨*proof*⟩

instance *stream* :: (*comm-semiring-1*) *comm-semiring-1* ⟨*proof*⟩

instance *stream* :: (*semiring-1-cancel*) *semiring-1-cancel* ⟨*proof*⟩

instance *stream* :: (*comm-semiring-1-cancel*) *comm-semiring-1-cancel*
⟨*proof*⟩

instance *stream* :: (*ring*) *ring* ⟨*proof*⟩

instance *stream* :: (*comm-ring*) *comm-ring* ⟨*proof*⟩

instance *stream* :: (*ring-1*) *ring-1* ⟨*proof*⟩

instance *stream* :: (*comm-ring-1*) *comm-ring-1* ⟨*proof*⟩

instance *stream* :: (*numeral*) *numeral* ⟨*proof*⟩

instance *stream* :: (*neg-numeral*) *neg-numeral* ⟨*proof*⟩

instance *stream* :: (*semiring-numeral*) *semiring-numeral* ⟨*proof*⟩

lemma *of-nat-stream* [*applicative-unfold*]: *of-nat* n = *sconst* (*of-nat* n)
⟨*proof*⟩

instance *stream* :: (*semiring-char-0*) *semiring-char-0*
⟨*proof*⟩

lemma *pure-stream-numeral* [*applicative-unfold*]: *numeral* n = *pure* (*numeral* n)
⟨*proof*⟩

instance *stream* :: (*ring-char-0*) *ring-char-0* ⟨*proof*⟩

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*

$\langle proof \rangle$

4.3.1 Pure correctness statement

definition *labels* :: *'a tree* ⇒ *'a list*
where *labels* = *fold-tree* ($\lambda x.$ [*x*]) *append*

lemma *labels-simps* [*simp*]:
  labels (*Leaf x*) = [*x*]
  labels (*Node l r*) = *labels l* @ *labels r*
$\langle proof \rangle$

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*) ($\lambda l\ r.$ *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*
$\langle proof \rangle$

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.
$\langle proof \rangle$

lemma *label-tree-list*: *pure labels* ⋄ *label-tree t* = *label-list* (*labels t*)
$\langle proof \rangle$

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

lemma *correctness-pure*:
  fixes *t* :: *'a tree*
  assumes *distinct*: $\bigwedge$*xs* :: *'a list*. *distinct* (*fst* (*run-state* (*label-list xs*) *s*))
  shows *distinct* (*labels* (*fst* (*run-state* (*label-tree t*) *s*)))
$\langle proof \rangle$

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* ⟨*proof*⟩
setup-lifting *type-definition-dual*

lift-definition *pure-dual* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ *dual*
is $\lambda pure.\ pure$ ⟨*proof*⟩

lift-definition *ap-dual* :: $(('a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b) \Rightarrow 'af1) \Rightarrow ('af1 \Rightarrow 'af3 \Rightarrow 'af13) \Rightarrow ('af13 \Rightarrow 'af2 \Rightarrow 'af) \Rightarrow 'af2\ dual \Rightarrow 'af3\ dual \Rightarrow 'af\ dual$
is $\lambda pure\ ap1\ ap2\ f\ x.\ ap2\ (ap1\ (pure\ (\lambda x\ f.\ f\ x))\ x)\ f$ ⟨*proof*⟩

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

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

definition *ap-state-rev* :: $('s, 'a \Rightarrow 'b)$ *state-rev* $\Rightarrow ('s, 'a)$ *state-rev* $\Rightarrow ('s, 'b)$ *state-rev*
where *ap-state-rev* = *ap-dual State-Monad.return State-Monad.ap State-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*
⟨*proof*⟩

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

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

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

⟨*proof*⟩

lemma *ap-state-rev-B*: $B\ f \diamond B\ x = B\ (State\text{-}Monad.return\ (\lambda x\ f.\ f\ x) \diamond x \diamond f)$
⟨*proof*⟩

lemma *ap-state-rev-pure-B*: $pure\ f \diamond B\ x = B\ (State\text{-}Monad.return\ f \diamond x)$
⟨*proof*⟩

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

lemma *ap-state-rev-rev-pure-B*: $pure\ f \diamond B\ x = B\ (pure\text{-}state\text{-}rev\ f \diamond x)$
⟨*proof*⟩

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*: $State\text{-}Monad.ap\ f\ x = State\text{-}Monad.bind\ f\ (\lambda f.\ State\text{-}Monad.bind\ x\ (State\text{-}Monad.return \circ f))$
⟨*proof*⟩

lemma *ap-pure-bind-state*: $pure\ x \diamond State\text{-}Monad.bind\ y\ f = State\text{-}Monad.bind\ y\ ((\diamond)\ (pure\ x) \circ f)$
⟨*proof*⟩

definition *kleisli-state* :: $('b \Rightarrow ('s, 'c)\ state) \Rightarrow ('a \Rightarrow ('s, 'b)\ state) \Rightarrow 'a \Rightarrow ('s, 'c)\ state$ (**infixl** ‹•› *55*)
where [*simp*]: $kleisli\text{-}state\ g\ f\ a = State\text{-}Monad.bind\ (f\ a)\ g$

definition *fetch* :: $('a\ stream, 'a)\ state$
where $fetch = State\text{-}Monad.bind\ State\text{-}Monad.get\ (\lambda s.\ State\text{-}Monad.bind\ (State\text{-}Monad.set\ (stl\ s))\ (\lambda\text{-}.\ State\text{-}Monad.return\ (shd\ s)))$

primrec *traverse* :: $('a \Rightarrow ('s, 'b)\ state) \Rightarrow 'a\ tree \Rightarrow ('s, 'b\ tree)\ state$
where
$traverse\ f\ (Leaf\ x) = pure\ Leaf \diamond f\ x$
$|\ traverse\ f\ (Node\ l\ r) = pure\ Node \diamond traverse\ f\ l \diamond traverse\ f\ r$

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

primrec *traverse-rev* :: $('a \Rightarrow ('s, 'b)\ state\text{-}rev) \Rightarrow 'a\ tree \Rightarrow ('s, 'b\ tree)\ state\text{-}rev$
where
$traverse\text{-}rev\ f\ (Leaf\ x) = pure\ Leaf \diamond f\ x$
$|\ traverse\text{-}rev\ f\ (Node\ l\ r) = pure\ Node \diamond traverse\text{-}rev\ f\ l \diamond traverse\text{-}rev\ f\ r$

definition *recurse* :: $('a \Rightarrow ('s, 'b)\ state) \Rightarrow 'a\ tree \Rightarrow ('s, 'b\ tree)\ state$
where $recurse\ f = un\text{-}B \circ traverse\text{-}rev\ (B \circ f)$

lemma *recurse-Leaf*: $recurse\ f\ (Leaf\ x) = pure\ Leaf \diamond f\ x$

⟨*proof*⟩

lemma *recurse-Node*:
  recurse f (*Node l r*) = *pure* (λ*r l*. *Node l r*) ⋄ *recurse f r* ⋄ *recurse f l*
⟨*proof*⟩

lemma *traverse-pure*: *traverse pure t* = *pure t*
⟨*proof*⟩

$B \circ B$ is an idiom morphism

lemma *B-pure*: *pure x* = *B* (*State-Monad.return x*)
⟨*proof*⟩

lemma *BB-pure*: *pure x* = *B* (*B* (*pure x*))
⟨*proof*⟩

lemma *BB-ap*: *B* (*B f*) ⋄ *B* (*B x*) = *B* (*B* (*f* ⋄ *x*))
⟨*proof*⟩

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*⟩

lemma *traverse-recurse*: *traverse f* = *un-B* ∘ *recurse-rev* (*B* ∘ *f*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*⟩

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* :: $'a \Rightarrow ('b\ stream,\ 'a \times 'b)\ state$
where *adorn a* = *pure* (*Pair a*) ⋄ *fetch*

abbreviation *label* :: $'a\ tree \Rightarrow ('b\ stream,\ ('a \times 'b)\ tree)\ state$
where *label* ≡ *traverse adorn*

abbreviation *unlabel* :: $('a \times 'b)\ tree \Rightarrow ('b\ stream,\ 'a\ tree)\ state$
where *unlabel* ≡ *recurse strip*

lemma *strip-adorn*: *strip* • *adorn* = *pure*
⟨*proof*⟩

lemma *correctness-monadic*: *unlabel* • *label* = *pure*
⟨*proof*⟩

4.3.3 Applicative correctness statement

Repeating an effect

primrec *repeatM* :: $nat \Rightarrow ('s,\ 'x)\ state \Rightarrow ('s,\ 'x\ list)\ state$
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*
⟨*proof*⟩

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

definition *lift-state* :: $('s,\ 'a)\ state \Rightarrow ('s,\ 'a\ option)\ state$
where [*applicative-unfold*]: *lift-state x* = *pure pure* ⋄ *x*

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

fun *assert* :: $('a \Rightarrow bool) \Rightarrow 'a\ option \Rightarrow 'a\ option$
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* :: $nat \Rightarrow ('s,\ 'x\ list\ option)\ state$
where *symbols n* ≡ *lift-state* (*repeatM n fresh*)

abbreviation (*input*) *disjoint* :: $'x\ list \Rightarrow 'x\ list \Rightarrow bool$
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*))
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩
termination ⟨*proof*⟩

adhoc-overloading *Applicative.pure* ⇌ *pure-pmf*

context fixes *p* :: *'a* ⇒ *'b pmf* **begin**

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

lemma *plabel-mirror*: *plabel* (*mirror t*) = *pure mirror* ⋄ *plabel t*
⟨*proof*⟩

lemma *plabel-subtree*: $plabel\ (subtree\ path\ t) = pure\ (subtree\ path) \diamond plabel\ t$
⟨*proof*⟩

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*: $(x, y) \in joinable\ R \longleftrightarrow (\exists z.\ (x, z) \in R \wedge (y, z) \in R)$
⟨*proof*⟩

lemma *joinableI*: $(x, z) \in R \Longrightarrow (y, z) \in R \Longrightarrow (x, y) \in joinable\ R$
⟨*proof*⟩

lemma *joinableD*: $(x, y) \in joinable\ R \Longrightarrow \exists z.\ (x, z) \in R \wedge (y, z) \in R$
⟨*proof*⟩

lemma *joinableE*:
assumes $(x, y) \in joinable\ R$
obtains z **where** $(x, z) \in R$ **and** $(y, z) \in R$
⟨*proof*⟩

lemma *refl-on-joinable*: $refl\text{-}on\ \{x.\ \exists y.\ (x, y) \in R\}\ (joinable\ R)$
⟨*proof*⟩

lemma *refl-joinable-iff*: $(\forall x.\ \exists y.\ (x, y) \in R) = refl\ (joinable\ R)$
⟨*proof*⟩

lemma *refl-joinable*: *refl* $R \Longrightarrow$ *refl* (*joinable* R)
⟨*proof*⟩

lemma *joinable-refl*: *refl* $R \Longrightarrow (x, x) \in$ *joinable* R
⟨*proof*⟩

lemma *sym-joinable*: *sym* (*joinable* R)
⟨*proof*⟩

lemma *joinable-sym*: $(x, y) \in$ *joinable* $R \Longrightarrow (y, x) \in$ *joinable* R
⟨*proof*⟩

lemma *joinable-mono*: $R \subseteq S \Longrightarrow$ *joinable* $R \subseteq$ *joinable* S
⟨*proof*⟩

lemma *refl-le-joinable*:
  assumes *refl* R
  shows $R \subseteq$ *joinable* R
⟨*proof*⟩

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*⟩

5.1.2 Confluence

definition *confluent* :: $'a$ *rel* $\Rightarrow$ *bool*
where *confluent* $R \longleftrightarrow (\forall x\ y\ y'.\ (x, y) \in R \wedge (x, y') \in R \longrightarrow (y, y') \in$ *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
⟨*proof*⟩

lemma *confluentD*:
  confluent $R \Longrightarrow (x, y) \in R \Longrightarrow (x,y') \in R \Longrightarrow (y, y') \in$ *joinable* R
⟨*proof*⟩

lemma *confluentE*:
  assumes *confluent* R **and** $(x, y) \in R$ **and** $(x, y') \in R$
  obtains z **where** $(y, z) \in R$ **and** $(y', z) \in R$
⟨*proof*⟩

lemma *trans-joinable*:
  assumes *trans* R **and** *confluent* R

shows *trans* (*joinable* R)
⟨*proof*⟩

5.1.3 Relation to reflexive transitive symmetric closure

lemma *joinable-le-rtscl*: *joinable* $(R^*) \subseteq (R \cup R^{-1})^*$
⟨*proof*⟩

theorem *joinable-eq-rtscl*:
assumes *confluent* (R^*)
shows *joinable* $(R^*) = (R \cup R^{-1})^*$
⟨*proof*⟩

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 $(\lambda x\ y.\ (x,\ y) \in R) = (\lambda x\ y.\ (x,\ y) \in$ *joinable* $R)$
⟨*proof*⟩

lemma *reflp-joinablep*: *reflp* $P \Longrightarrow$ *reflp* (*joinablep* P)
⟨*proof*⟩

lemma *joinablep-refl*: *reflp* $P \Longrightarrow$ *joinablep* $P\ x\ x$
⟨*proof*⟩

lemma *reflp-le-joinablep*: *reflp* $P \Longrightarrow P \leq$ *joinablep* P
⟨*proof*⟩

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
⟨*proof*⟩

lemma *subst-liftn*:
$i \leq n + k \wedge k \leq i \Longrightarrow$ (*liftn* (*Suc* n) $s\ k$)$[t/i]$ = *liftn* $n\ s\ k$
⟨*proof*⟩

lemma *subst-lift2*[*simp*]: (*lift* (*lift* t *0*) *0*)[x/*Suc 0*] = *lift* t *0*
⟨*proof*⟩

lemma *free-liftn*:
$free\ (liftn\ n\ t\ k)\ i = (i < k \land free\ t\ i \lor k + n \leq i \land free\ t\ (i - n))$
⟨*proof*⟩

5.2.2 Reduction

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

abbreviation *beta-eta-reds* :: *dB* ⇒ *dB* ⇒ *bool* (**infixl** ‹$\to_{\beta\eta}{}^*$› *50*)
where $s \to_{\beta\eta}{}^*\ t \equiv (beta\text{-}eta)^{**}\ s\ t$

lemma *beta-into-beta-eta-reds*: $s \to_\beta\ t \Longrightarrow s \to_{\beta\eta}{}^*\ t$
⟨*proof*⟩

lemma *eta-into-beta-eta-reds*: $s \to_\eta\ t \Longrightarrow s \to_{\beta\eta}{}^*\ t$
⟨*proof*⟩

lemma *beta-reds-into-beta-eta-reds*: $s \to_\beta{}^*\ t \Longrightarrow s \to_{\beta\eta}{}^*\ t$
⟨*proof*⟩

lemma *eta-reds-into-beta-eta-reds*: $s \to_\eta{}^*\ t \Longrightarrow s \to_{\beta\eta}{}^*\ t$
⟨*proof*⟩

lemma *beta-eta-appL*[*intro*]: $s \to_{\beta\eta}{}^*\ s' \Longrightarrow s \circ t \to_{\beta\eta}{}^*\ s' \circ t$
⟨*proof*⟩

lemma *beta-eta-appR*[*intro*]: $t \to_{\beta\eta}{}^*\ t' \Longrightarrow s \circ t \to_{\beta\eta}{}^*\ s \circ t'$
⟨*proof*⟩

lemma *beta-eta-abs*[*intro*]: $t \to_{\beta\eta}{}^*\ t' \Longrightarrow Abs\ t \to_{\beta\eta}{}^*\ Abs\ t'$
⟨*proof*⟩

lemma *beta-eta-lift*: $s \to_{\beta\eta}{}^*\ t \Longrightarrow lift\ s\ k \to_{\beta\eta}{}^*\ lift\ t\ k$
⟨*proof*⟩

lemma *confluent-beta-eta-reds*: *Joinable.confluent* $\{(s, t).\ s \to_{\beta\eta}{}^*\ t\}$
⟨*proof*⟩

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 $s \leftrightarrow t$

⟨*proof*⟩

lemma *term-equivE*:
assumes $s \leftrightarrow t$
obtains u **where** $s \rightarrow_{\beta\eta}{}^* u$ **and** $t \rightarrow_{\beta\eta}{}^* u$
⟨*proof*⟩

lemma *reds-into-equiv*[*elim*]: $s \rightarrow_{\beta\eta}{}^* t \Longrightarrow s \leftrightarrow t$
⟨*proof*⟩

lemma *beta-into-equiv*[*elim*]: $s \rightarrow_\beta t \Longrightarrow s \leftrightarrow t$
⟨*proof*⟩

lemma *eta-into-equiv*[*elim*]: $s \rightarrow_\eta t \Longrightarrow s \leftrightarrow t$
⟨*proof*⟩

lemma *beta-reds-into-equiv*[*elim*]: $s \rightarrow_\beta{}^* t \Longrightarrow s \leftrightarrow t$
⟨*proof*⟩

lemma *eta-reds-into-equiv*[*elim*]: $s \rightarrow_\eta{}^* t \Longrightarrow s \leftrightarrow t$
⟨*proof*⟩

lemma *term-refl*[*iff*]: $t \leftrightarrow t$
⟨*proof*⟩

lemma *term-sym*[*sym*]: $(s \leftrightarrow t) \Longrightarrow (t \leftrightarrow s)$
⟨*proof*⟩

lemma *conversep-term* [*simp*]: *conversep* $(\leftrightarrow) = (\leftrightarrow)$
⟨*proof*⟩

lemma *term-trans*[*trans*]: $s \leftrightarrow t \Longrightarrow t \leftrightarrow u \Longrightarrow s \leftrightarrow u$
⟨*proof*⟩

lemma *term-beta-trans*[*trans*]: $s \leftrightarrow t \Longrightarrow t \rightarrow_\beta u \Longrightarrow s \leftrightarrow u$
⟨*proof*⟩

lemma *term-eta-trans*[*trans*]: $s \leftrightarrow t \Longrightarrow t \rightarrow_\eta u \Longrightarrow s \leftrightarrow u$
⟨*proof*⟩

lemma *equiv-appL*[*intro*]: $s \leftrightarrow s' \Longrightarrow s \circ t \leftrightarrow s' \circ t$
⟨*proof*⟩

lemma *equiv-appR*[*intro*]: $t \leftrightarrow t' \Longrightarrow s \circ t \leftrightarrow s \circ t'$
⟨*proof*⟩

lemma *equiv-app*: $s \leftrightarrow s' \Longrightarrow t \leftrightarrow t' \Longrightarrow s \circ t \leftrightarrow s' \circ t'$
⟨*proof*⟩

lemma *equiv-abs*[*intro*]: $t \leftrightarrow t' \Longrightarrow Abs\ t \leftrightarrow Abs\ t'$
⟨*proof*⟩

lemma *equiv-lift*: $s \leftrightarrow t \Longrightarrow lift\ s\ k \leftrightarrow lift\ t\ k$
⟨*proof*⟩

lemma *equiv-liftn*: $s \leftrightarrow t \Longrightarrow liftn\ n\ s\ k \leftrightarrow liftn\ n\ t\ k$
⟨*proof*⟩

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

lemma *equiv-eq-rtscl-reds*: $term\text{-}equiv = (sup\ beta\text{-}eta\ beta\text{-}eta^{-1-1})^{**}$
⟨*proof*⟩

end

5.3 Combinators defined as closed lambda terms

theory *Combinators*
imports *Beta-Eta*
begin

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

lemma *I-eval*: $\mathcal{I} \circ x \rightarrow_\beta x$
⟨*proof*⟩

lemma *I-equiv*[*iff*]: $\mathcal{I} \circ x \leftrightarrow x$
⟨*proof*⟩

lemma *I-closed*[*simp*]: $liftn\ n\ \mathcal{I}\ k = \mathcal{I}$
⟨*proof*⟩

lemma *B-eval1*: $\mathcal{B} \circ g \rightarrow_\beta Abs\ (Abs\ (lift\ (lift\ g\ 0)\ 0 \circ (Var\ 1 \circ Var\ 0)))$
⟨*proof*⟩

lemma *B-eval2*: $\mathcal{B} \circ g \circ f \rightarrow_\beta{}^* Abs\ (lift\ g\ 0 \circ (lift\ f\ 0 \circ Var\ 0))$
⟨*proof*⟩

lemma *B-eval*: $\mathcal{B} \circ g \circ f \circ x \rightarrow_\beta{}^* g \circ (f \circ x)$
⟨*proof*⟩

lemma *B-equiv*[*iff*]: $\mathcal{B} \circ g \circ f \circ x \leftrightarrow g \circ (f \circ x)$
⟨*proof*⟩

lemma *B-closed*[*simp*]: $liftn\ n\ \mathcal{B}\ k = \mathcal{B}$
⟨*proof*⟩

lemma *T-eval1*: $\mathcal{T} \circ x \rightarrow_\beta Abs\ (Var\ 0 \circ lift\ x\ 0)$
⟨*proof*⟩

lemma *T-eval*: $\mathcal{T} \circ x \circ f \rightarrow_\beta{}^* f \circ x$
⟨*proof*⟩

lemma *T-equiv*[*iff*]: $\mathcal{T} \circ x \circ f \leftrightarrow f \circ x$
⟨*proof*⟩

lemma *T-closed*[*simp*]: $liftn\ n\ \mathcal{T}\ k = \mathcal{T}$
⟨*proof*⟩

end

5.4 Idiomatic terms – Properties and operations

theory *Idiomatic-Terms*
imports *Combinators*
begin

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

5.4.1 Basic definitions

datatype *'a itrm* =
  Opaque 'a | *Pure dB*
 | *IAp 'a itrm 'a itrm* (**infixl** ‹◇› *150*)

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

abbreviation *iorder x* ≡ *length* (*opaque x*)

inductive *itrm-cong* :: (*'a itrm* ⇒ *'a itrm* ⇒ *bool*) ⇒ *'a itrm* ⇒ *'a itrm* ⇒ *bool*
for *R*
where
  into-itrm-cong: *R x y* ⟹ *itrm-cong R x y*
 | *pure-cong*[*intro*]: *x* ↔ *y* ⟹ *itrm-cong R* (*Pure x*) (*Pure y*)
 | *ap-cong*: *itrm-cong R f f'* ⟹ *itrm-cong R x x'* ⟹ *itrm-cong R* (*f* ◇ *x*) (*f'* ◇ *x'*)
 | *itrm-refl*[*iff*]: *itrm-cong R x x*

| *itrm-sym*[*sym*]: *itrm-cong R x y* $\Longrightarrow$ *itrm-cong R y x*
| *itrm-trans*[*trans*]: *itrm-cong R x y* $\Longrightarrow$ *itrm-cong R y z* $\Longrightarrow$ *itrm-cong R x z*

lemma *ap-congL*[*intro*]: *itrm-cong R f f'* $\Longrightarrow$ *itrm-cong R* $(f \diamond x)$ $(f' \diamond x)$
⟨*proof*⟩

lemma *ap-congR*[*intro*]: *itrm-cong R x x'* $\Longrightarrow$ *itrm-cong R* $(f \diamond x)$ $(f \diamond x')$
⟨*proof*⟩

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 $x \cong y \equiv$ *itrm-cong* (λ- -. *False*) *x y*

lemma *pure-similarE*:
assumes *Pure* $x' \cong y$
obtains y' **where** $y =$ *Pure* y' **and** $x' \leftrightarrow y'$
⟨*proof*⟩

lemma *opaque-similarE*:
assumes *Opaque* $x' \cong y$
obtains y' **where** $y =$ *Opaque* y' **and** $x' = y'$
⟨*proof*⟩

lemma *ap-similarE*:
assumes $x1 \diamond x2 \cong y$
obtains *y1 y2* **where** $y = y1 \diamond y2$ **and** $x1 \cong y1$ **and** $x2 \cong y2$
⟨*proof*⟩

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* $\mathcal{I} \diamond x$) *x*
| *idiom-comp*: *idiom-rule* (*Pure* $\mathcal{B} \diamond g \diamond f \diamond x$) $(g \diamond (f \diamond x))$
| *idiom-hom*: *idiom-rule* (*Pure* $f \diamond$ *Pure* x) (*Pure* $(f\ ^\circ\ x)$)
| *idiom-xchng*: *idiom-rule* $(f \diamond$ *Pure* $x)$ (*Pure* $(\mathcal{T}\ ^\circ\ x) \diamond f$)

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* $\Longrightarrow$ $x \simeq y$ ⟨*proof*⟩

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*: $x \cong y \Longrightarrow x \simeq y$
⟨*proof*⟩

lemma *opaque-equiv*: $x \simeq y \Longrightarrow$ *opaque* x = *opaque* y
⟨*proof*⟩

lemma *iorder-equiv*: $x \simeq y \Longrightarrow$ *iorder* x = *iorder* y
⟨*proof*⟩

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

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

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

lemma *equiv-into-ext-equiv*: $x \simeq y \Longrightarrow x \simeq^+ y$
⟨*proof*⟩

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

end

5.4.2 Syntactic unlifting

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

lemma *absn-cong*[*intro*]: $s \leftrightarrow t \Longrightarrow$ (*Abs*^^*n*) $s \leftrightarrow$ (*Abs*^^*n*) t
⟨*proof*⟩

lemma *free-unlift*: *free* (*unlift' n x i*) $j \Longrightarrow j \geq n \vee (j \geq i \wedge j < i +$ *iorder x*)
⟨*proof*⟩

lemma *unlift-subst*: $j \leq i \wedge j \leq n \Longrightarrow$ (*unlift'* (*Suc n*) *t* (*Suc i*))[*s*/*j*] = *unlift' n*

$t\ i$
⟨*proof*⟩

lemma *unlift′-equiv*: $x \simeq y \Longrightarrow \mathit{unlift}'\ n\ x\ i \leftrightarrow \mathit{unlift}'\ n\ y\ i$
⟨*proof*⟩

lemma *unlift-equiv*: $x \simeq y \Longrightarrow \mathit{unlift}\ x \leftrightarrow \mathit{unlift}\ y$
⟨*proof*⟩

Preserving variables **primrec** *unlift-vars* :: $\mathit{nat} \Rightarrow \mathit{nat}\ \mathit{itrm} \Rightarrow \mathit{dB}$
where
  $\mathit{unlift\text{-}vars}\ n\ (\mathit{Opaque}\ i) = \mathit{Var}\ i$
| $\mathit{unlift\text{-}vars}\ n\ (\mathit{Pure}\ x) = \mathit{liftn}\ n\ x\ 0$
| $\mathit{unlift\text{-}vars}\ n\ (x \diamond y) = \mathit{unlift\text{-}vars}\ n\ x \circ \mathit{unlift\text{-}vars}\ n\ y$

lemma *all-pure-unlift-vars*: $\mathit{opaque}\ x = [] \Longrightarrow x \simeq \mathit{Pure}\ (\mathit{unlift\text{-}vars}\ 0\ x)$
⟨*proof*⟩

5.4.3 Canonical forms

inductive-set *CF* :: $'a\ \mathit{itrm}\ \mathit{set}$
where
  pure-cf[*iff*]: $\mathit{Pure}\ x \in \mathit{CF}$
| *ap-cf*[*intro*]: $f \in \mathit{CF} \Longrightarrow f \diamond \mathit{Opaque}\ x \in \mathit{CF}$

primrec *CF-pure* :: $'a\ \mathit{itrm} \Rightarrow \mathit{dB}$
where
  $\mathit{CF\text{-}pure}\ (\mathit{Opaque}\ \text{-}) = \mathit{undefined}$
| $\mathit{CF\text{-}pure}\ (\mathit{Pure}\ x) = x$
| $\mathit{CF\text{-}pure}\ (x \diamond \text{-}) = \mathit{CF\text{-}pure}\ x$

lemma *ap-cfD1*[*dest*]: $f \diamond x \in \mathit{CF} \Longrightarrow f \in \mathit{CF}$
⟨*proof*⟩

lemma *ap-cfD2*[*dest*]: $f \diamond x \in \mathit{CF} \Longrightarrow \exists x'.\ x = \mathit{Opaque}\ x'$
⟨*proof*⟩

lemma *opaque-not-cf*[*simp*]: $\mathit{Opaque}\ x \in \mathit{CF} \Longrightarrow \mathit{False}$
⟨*proof*⟩

lemma *cf-unlift*:
 assumes $x \in \mathit{CF}$
  shows $\mathit{CF\text{-}pure}\ x \leftrightarrow \mathit{unlift}\ x$
⟨*proof*⟩

lemma *cf-similarI*:
 assumes $x \in \mathit{CF}\ y \in \mathit{CF}$
  and $\mathit{opaque}\ x = \mathit{opaque}\ y$
  and $\mathit{CF\text{-}pure}\ x \leftrightarrow \mathit{CF\text{-}pure}\ y$

shows $x \cong y$
$\langle proof \rangle$

lemma *cf-similarD*:
assumes *in-cf*: $x \in CF$ $y \in CF$
and *similar*: $x \cong y$
shows $CF\text{-}pure\ x \leftrightarrow CF\text{-}pure\ y \wedge opaque\ x = opaque\ y$
$\langle proof \rangle$

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 $x \cong y$
$\langle proof \rangle$

5.4.4 Normalisation of idiomatic terms

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

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

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

lemma *norm-pn-in-cf*:
assumes $x \in CF$
shows $norm\text{-}pn\ f\ x \in CF$
$\langle proof \rangle$

lemma *norm-nn-in-cf*:
assumes $n \in CF$ $n' \in CF$
shows $norm\text{-}nn\ n\ n' \in CF$
$\langle proof \rangle$

lemma *norm-in-cf*: $norm\ x \in CF$

⟨*proof*⟩

lemma *norm-pn-equiv*:
assumes $x \in CF$
shows *norm-pn* $f\ x \simeq$ *Pure* $f \diamond x$
⟨*proof*⟩

lemma *norm-nn-equiv*:
assumes $n \in CF$ $n' \in CF$
shows *norm-nn* $n\ n' \simeq n \diamond n'$
⟨*proof*⟩

lemma *norm-equiv*: *norm* $x \simeq x$
⟨*proof*⟩

lemma *normal-form*: **obtains** n **where** $n \simeq x$ **and** $n \in CF$
⟨*proof*⟩

5.4.5 Lifting with normal forms

lemma *nf-unlift*:
assumes *equiv*: $n \simeq x$ **and** *cf*: $n \in CF$
shows *CF-pure* $n \leftrightarrow$ *unlift* x
⟨*proof*⟩

theorem *nf-lifting*:
assumes *opaque*: *opaque* $x =$ *opaque* y
and *base-eq*: *unlift* $x \leftrightarrow$ *unlift* y
shows $x \simeq y$
⟨*proof*⟩

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-dist* $=$ *fold* $(\lambda i\ t.\ t \circ$ *Var* $i)$

lemma *var-dist-Nil*[*simp*]: *var-dist* $[]\ t = t$
⟨*proof*⟩

lemma *var-dist-Cons*[*simp*]: *var-dist* $(v\ \#\ vs)\ t =$ *var-dist* $vs\ (t \circ$ *Var* $v)$
⟨*proof*⟩

lemma *var-dist-append1*: *var-dist* $(vs\ @\ [v])\ t =$ *var-dist* $vs\ t \circ$ *Var* v
⟨*proof*⟩

lemma *var-dist-frees*: $\mathit{frees}\ (\mathit{var\text{-}dist}\ \mathit{vs}\ t) = \mathit{frees}\ t \cup \mathit{set}\ \mathit{vs}$
⟨*proof*⟩

lemma *var-dist-subst-lt*:
$\forall v \in \mathit{set}\ \mathit{vs}.\ i < v \Longrightarrow (\mathit{var\text{-}dist}\ \mathit{vs}\ s)[t/i] = \mathit{var\text{-}dist}\ (\mathit{map}\ (\lambda v.\ v - 1)\ \mathit{vs})\ (s[t/i])$
⟨*proof*⟩

lemma *var-dist-subst-gt*:
$\forall v \in \mathit{set}\ \mathit{vs}.\ v < i \Longrightarrow (\mathit{var\text{-}dist}\ \mathit{vs}\ s)[t/i] = \mathit{var\text{-}dist}\ \mathit{vs}\ (s[t/i])$
⟨*proof*⟩

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

lemma *vsubst-subst*[*simp*]: $(\mathit{Var}\ u)[\mathit{Var}\ v/w] = \mathit{Var}\ (\mathit{vsubst}\ u\ v\ w)$
⟨*proof*⟩

lemma *vsubst-subst-lt*[*simp*]: $u < w \Longrightarrow \mathit{vsubst}\ u\ v\ w = u$
⟨*proof*⟩

lemma *var-dist-subst-Var*:
$(\mathit{var\text{-}dist}\ \mathit{vs}\ s)[\mathit{Var}\ i/j] = \mathit{var\text{-}dist}\ (\mathit{map}\ (\lambda v.\ \mathit{vsubst}\ v\ i\ j)\ \mathit{vs})\ (s[\mathit{Var}\ i/j])$
⟨*proof*⟩

lemma *var-dist-cong*: $s \leftrightarrow t \Longrightarrow \mathit{var\text{-}dist}\ \mathit{vs}\ s \leftrightarrow \mathit{var\text{-}dist}\ \mathit{vs}\ t$
⟨*proof*⟩

Preliminaries: Eta reductions with permuted variables **lemma** *absn-subst*:
$((\mathit{Abs}^{\frown\frown}n)\ s)[t/k] = (\mathit{Abs}^{\frown\frown}n)\ (s[\mathit{liftn}\ n\ t\ 0/k{+}n])$
⟨*proof*⟩

lemma *absn-beta-equiv*: $(\mathit{Abs}^{\frown\frown}\mathit{Suc}\ n)\ s \circ t \leftrightarrow (\mathit{Abs}^{\frown\frown}n)\ (s[\mathit{liftn}\ n\ t\ 0/n])$
⟨*proof*⟩

lemma *absn-dist-eta*: $(\mathit{Abs}^{\frown\frown}n)\ (\mathit{var\text{-}dist}\ (\mathit{rev}\ [0..{<}n])\ (\mathit{liftn}\ n\ t\ 0)) \leftrightarrow t$
⟨*proof*⟩

primrec *strip-context* :: $\mathit{nat} \Rightarrow \mathit{dB} \Rightarrow \mathit{nat} \Rightarrow \mathit{dB}$
where
$\quad\mathit{strip\text{-}context}\ n\ (\mathit{Var}\ i)\ k = (\mathit{if}\ i < k\ \mathit{then}\ \mathit{Var}\ i\ \mathit{else}\ \mathit{Var}\ (i - n))$
$\mid \mathit{strip\text{-}context}\ n\ (\mathit{Abs}\ t)\ k = \mathit{Abs}\ (\mathit{strip\text{-}context}\ n\ t\ (\mathit{Suc}\ k))$
$\mid \mathit{strip\text{-}context}\ n\ (s \circ t)\ k = \mathit{strip\text{-}context}\ n\ s\ k \circ \mathit{strip\text{-}context}\ n\ t\ k$

lemma *strip-context-liftn*: $\mathit{strip\text{-}context}\ n\ (\mathit{liftn}\ (m + n)\ t\ k)\ k = \mathit{liftn}\ m\ t\ k$
⟨*proof*⟩

lemma *liftn-strip-context*:
assumes $\forall i \in \mathit{frees}\ t.\ i < k \lor k + n \le i$
shows $\mathit{liftn}\ n\ (\mathit{strip\text{-}context}\ n\ t\ k)\ k = t$

$\langle proof\rangle$

lemma *absn-dist-eta-free*:
  assumes $\forall\, i{\in}frees\ t.\ n \le i$
  shows $(Abs\hat{}\hat{}n)\ (var\text{-}dist\ (rev\ [0..{<}n])\ t) \leftrightarrow strip\text{-}context\ n\ t\ 0$ (**is** $?lhs\ t \leftrightarrow ?rhs$)
$\langle proof\rangle$

definition *perm-vars* :: $nat \Rightarrow nat\ list \Rightarrow bool$
where $perm\text{-}vars\ n\ vs \longleftrightarrow distinct\ vs \wedge set\ vs = \{0..{<}n\}$

lemma *perm-vars-distinct*: $perm\text{-}vars\ n\ vs \Longrightarrow distinct\ vs$
$\langle proof\rangle$

lemma *perm-vars-length*: $perm\text{-}vars\ n\ vs \Longrightarrow length\ vs = n$
$\langle proof\rangle$

lemma *perm-vars-lt*: $perm\text{-}vars\ n\ vs \Longrightarrow \forall\, i{\in}set\ vs.\ i < n$
$\langle proof\rangle$

lemma *perm-vars-nth-lt*: $perm\text{-}vars\ n\ vs \Longrightarrow i < n \Longrightarrow vs\ !\ i < n$
$\langle proof\rangle$

lemma *perm-vars-inj-on-nth*:
  assumes $perm\text{-}vars\ n\ vs$
    shows $inj\text{-}on\ (nth\ vs)\ \{0..{<}n\}$
$\langle proof\rangle$

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

lemma *perm-vars-inv-nth*:
  assumes $perm\text{-}vars\ n\ vs$
      and $i < n$
    shows $perm\text{-}vars\text{-}inv\ n\ vs\ (vs\ !\ i) = i$
$\langle proof\rangle$

lemma *dist-perm-eta*:
  assumes *perm-vars*: $perm\text{-}vars\ n\ vs$
  obtains vs' **where** $\bigwedge t.\ \forall\, i{\in}frees\ t.\ n \le i \Longrightarrow$
    $(Abs\hat{}\hat{}n)\ (var\text{-}dist\ vs'\ ((Abs\hat{}\hat{}n)\ (var\text{-}dist\ vs\ (liftn\ n\ t\ 0)))) \leftrightarrow strip\text{-}context\ n\ t\ 0$
$\langle proof\rangle$

lemma *liftn-absn*: $liftn\ n\ ((Abs\hat{}\hat{}m)\ t)\ k = (Abs\hat{}\hat{}m)\ (liftn\ n\ t\ (k + m))$
$\langle proof\rangle$

lemma *liftn-var-dist-lt*:
  $\forall\, i{\in}set\ vs.\ i < k \Longrightarrow liftn\ n\ (var\text{-}dist\ vs\ t)\ k = var\text{-}dist\ vs\ (liftn\ n\ t\ k)$

⟨*proof*⟩

lemma *liftn-context-conv*: $k \le k' \Longrightarrow \forall\, i{\in}frees\ t.\ i < k \lor k' \le i \Longrightarrow liftn\ n\ t\ k = liftn\ n\ t\ k'$
⟨*proof*⟩

lemma *liftn-liftn0*: $\forall\, i{\in}frees\ t.\ k \le i \Longrightarrow liftn\ n\ t\ k = liftn\ n\ t\ 0$
⟨*proof*⟩

lemma *dist-perm-eta-equiv*:
  assumes *perm-vars*: *perm-vars n vs*
    and *not-free*: $\forall\, i{\in}frees\ s.\ n \le i\ \forall\, i{\in}frees\ t.\ n \le i$
    and *perm-equiv*: $(Abs\widehat{\ }\widehat{\ }n)\ (var\text{-}dist\ vs\ s) \leftrightarrow (Abs\widehat{\ }\widehat{\ }n)\ (var\text{-}dist\ vs\ t)$
  shows $strip\text{-}context\ n\ s\ 0 \leftrightarrow strip\text{-}context\ n\ t\ 0$
⟨*proof*⟩

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 (λa b. Option.bind b (f a)) xs (Some e)*

lemma *bind-eq-SomeE*:
  assumes *Option.bind x f = Some y*
  obtains x' **where** $x = Some\ x'$ **and** $f\ x' = Some\ y$
⟨*proof*⟩

lemma *foldr-option-Nil*[*simp*]: *foldr-option f [] e = Some e*
⟨*proof*⟩

lemma *foldr-option-Cons-SomeE*:
  assumes *foldr-option f (x#xs) e = Some y*
  obtains y' **where** $foldr\text{-}option\ f\ xs\ e = Some\ y'$ **and** $f\ x\ y' = Some\ y$
⟨*proof*⟩

locale *bracket-abstraction* =
  fixes *term-bracket* :: $nat \Rightarrow dB \Rightarrow dB\ option$
  assumes *bracket-app*: $term\text{-}bracket\ i\ s = Some\ s' \Longrightarrow s' \circ Var\ i \leftrightarrow s$
  assumes *bracket-frees*: $term\text{-}bracket\ i\ s = Some\ s' \Longrightarrow frees\ s' = frees\ s - \{i\}$
begin

definition *term-brackets* :: $nat\ list \Rightarrow dB \Rightarrow dB\ option$
where *term-brackets = foldr-option term-bracket*

lemma *term-brackets-Nil*[*simp*]: *term-brackets [] t = Some t*
⟨*proof*⟩

lemma *term-brackets-Cons-SomeE*:
  assumes $term\text{-}brackets\ (v\#vs)\ t = Some\ t'$
  obtains s' **where** $term\text{-}brackets\ vs\ t = Some\ s'$ **and** $term\text{-}bracket\ v\ s' = Some\ t'$

⟨*proof*⟩

lemma *term-brackets-ConsI*:
  assumes $term\text{-}brackets\ vs\ t = Some\ t'$
    and $term\text{-}bracket\ v\ t' = Some\ t''$
   shows $term\text{-}brackets\ (v\#vs)\ t = Some\ t''$
⟨*proof*⟩

lemma *term-brackets-dist*:
  assumes $term\text{-}brackets\ vs\ t = Some\ t'$
   shows $var\text{-}dist\ vs\ t' \leftrightarrow t$
⟨*proof*⟩

end

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

lemma *strip-unlift-vars*:
  assumes $opaque\ x = []$
  shows $strip\text{-}context\ n\ (unlift\text{-}vars\ n\ x)\ 0 = unlift\text{-}vars\ 0\ x$
⟨*proof*⟩

lemma *unlift-vars-frees*: $\forall\, i{\in}frees\ (unlift\text{-}vars\ n\ x).\ i \in set\ (opaque\ x) \lor n \le i$
⟨*proof*⟩

locale $itrm\text{-}abstraction = special\text{-}idiom\ extra\text{-}rule$ **for** $extra\text{-}rule :: nat\ itrm \Rightarrow \text{-} +$
  fixes $itrm\text{-}bracket :: nat \Rightarrow nat\ itrm \Rightarrow nat\ itrm\ option$
  assumes *itrm-bracket-ap*: $itrm\text{-}bracket\ i\ x = Some\ x' \Longrightarrow x' \diamond Opaque\ i \simeq^{+} x$
  assumes *itrm-bracket-opaque*:
   $itrm\text{-}bracket\ i\ x = Some\ x' \Longrightarrow set\ (opaque\ x') = set\ (opaque\ x) - \{i\}$
begin

definition $itrm\text{-}brackets = foldr\text{-}option\ itrm\text{-}bracket$

lemma *itrm-brackets-Nil*[*simp*]: $itrm\text{-}brackets\ []\ x = Some\ x$
⟨*proof*⟩

lemma *itrm-brackets-Cons-SomeE*:
  assumes $itrm\text{-}brackets\ (v\#vs)\ x = Some\ x'$
  obtains y' **where** $itrm\text{-}brackets\ vs\ x = Some\ y'$ **and** $itrm\text{-}bracket\ v\ y' = Some\ x'$
⟨*proof*⟩

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

lemma *opaque-dist-cong*: $x \simeq^{+} y \Longrightarrow opaque\text{-}dist\ vs\ x \simeq^{+} opaque\text{-}dist\ vs\ y$
⟨*proof*⟩

lemma *itrm-brackets-dist*:
  assumes *defined*: $itrm\text{-}brackets\ vs\ x = Some\ x'$
    shows $opaque\text{-}dist\ vs\ x' \simeq^{+} x$
⟨*proof*⟩

lemma *itrm-brackets-opaque*:
  assumes $itrm\text{-}brackets\ vs\ x = Some\ x'$
    shows $set\ (opaque\ x') = set\ (opaque\ x) - set\ vs$
⟨*proof*⟩

lemma *itrm-brackets-all*:
  assumes *all-opaque*: $set\ (opaque\ x) \subseteq set\ vs$
      and *defined*: $itrm\text{-}brackets\ vs\ x = Some\ x'$
    shows $opaque\ x' = []$
⟨*proof*⟩

lemma *itrm-brackets-all-unlift-vars*:
  assumes *all-opaque*: $set\ (opaque\ x) \subseteq set\ vs$
      and *defined*: $itrm\text{-}brackets\ vs\ x = Some\ x'$
    shows $x' \simeq^{+} Pure\ (unlift\text{-}vars\ 0\ x')$
⟨*proof*⟩

end

5.4.7 Lifting with bracket abstraction

locale *lifted-bracket* = *bracket-abstraction* + *itrm-abstraction* +
  assumes *bracket-compat*:
    $set\ (opaque\ x) \subseteq \{0..<n\} \Longrightarrow i < n \Longrightarrow$
      $term\text{-}bracket\ i\ (unlift\text{-}vars\ n\ x) = map\text{-}option\ (unlift\text{-}vars\ n)\ (itrm\text{-}bracket\ i$
$x)$
begin

lemma *brackets-unlift-vars-swap*:
  assumes *all-opaque*: $set\ (opaque\ x) \subseteq \{0..<n\}$
      and *vs-bound*: $set\ vs \subseteq \{0..<n\}$
      and *defined*: $itrm\text{-}brackets\ vs\ x = Some\ x'$
    shows $term\text{-}brackets\ vs\ (unlift\text{-}vars\ n\ x) = Some\ (unlift\text{-}vars\ n\ x')$
⟨*proof*⟩

theorem *bracket-lifting*:
  assumes *all-vars*: $set\ (opaque\ x) \cup set\ (opaque\ y) \subseteq \{0..<n\}$
      and *perm-vars*: $perm\text{-}vars\ n\ vs$
      and *defined*: $itrm\text{-}brackets\ vs\ x = Some\ x'\ itrm\text{-}brackets\ vs\ y = Some\ y'$
      and *base-eq*: $(Abs\widehat{\ }\widehat{\ }n)\ (unlift\text{-}vars\ n\ x) \leftrightarrow (Abs\widehat{\ }\widehat{\ }n)\ (unlift\text{-}vars\ n\ y)$
    shows $x \simeq^{+} y$
⟨*proof*⟩

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.