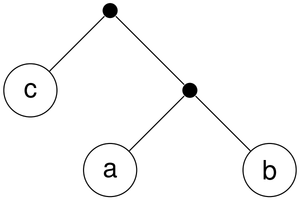
# Applicative Functors in Isabelle/HOL

Joshua Schneider
`joshuas@student.ethz.ch`

December 8, 2015

## Introduction: Tree Labels

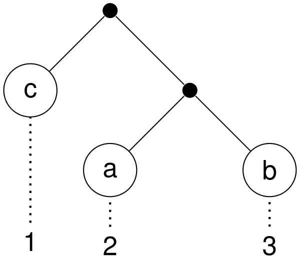
```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```



Inspired by G. Hutton and D. Fulger, “Reasoning About Effects: Seeing the Wood Through the Trees.” in *Proceedings of the Symposium on Trends in Functional Programming*, (Nijmegen, The Netherlands, 2008).

## Introduction: Tree Labels

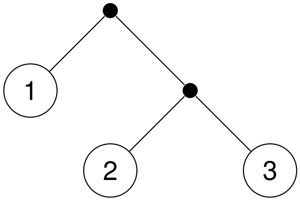
```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```



Inspired by G. Hutton and D. Fulger, “Reasoning About Effects: Seeing the Wood Through the Trees.” in *Proceedings of the Symposium on Trends in Functional Programming*, (Nijmegen, The Netherlands, 2008).

## Introduction: Tree Labels

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```



Inspired by G. Hutton and D. Fulger, “Reasoning About Effects: Seeing the Wood Through the Trees.” in *Proceedings of the Symposium on Trends in Functional Programming*, (Nijmegen, The Netherlands, 2008).

## Composing Stateful Computations

Standard solution:
state monad

```
fresh = do
  x <- get
  put (x + 1)
  return x
```

## Composing Stateful Computations

Standard solution:
state monad

```
fresh = do
  x <- get
  put (x + 1)
  return x

numberTree (Leaf _)   = do
  x <- fresh
  return (Leaf x)
numberTree (Node l r) = do
  l' <- numberTree l
  r' <- numberTree r
  return (Node l' r')
```

## Composing Stateful Computations

Standard solution:
state monad

```
fresh = do
  x <- get
  put (x + 1)
  return x

numberTree (Leaf _)   = do
  x <- fresh
  return (Leaf x)
numberTree (Node l r) = do
  l' <- numberTree l
  r' <- numberTree r
  return (Node l' r')
```

Applicative style

```
class Applicative m => Monad m where
  ...
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

C. McBride and R. Paterson, “Applicative Programming with Effects.” *Journal of Functional Programming, 18* (1). 2008, 1–13.

## Composing Stateful Computations

Standard solution:
state monad

```
fresh = do
  x <- get
  put (x + 1)
  return x

numberTree (Leaf _)   = do
  x <- fresh
  return (Leaf x)
numberTree (Node l r) = do
  l' <- numberTree l
  r' <- numberTree r
  return (Node l' r')
```

Applicative style

```
class Applicative m => Monad m where
  ...
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

numberTree (Leaf _)   =
  pure Leaf <*> fresh

numberTree (Node l r) =
  pure Node <*>
    numberTree l <*> numberTree r
```

C. McBride and R. Paterson, “Applicative Programming with Effects.” *Journal of Functional Programming, 18* (1). 2008, 1–13.

## Renaming Trees and Lists

```
labels (Leaf x)   = [x]
labels (Node l r) = labels l ++ labels r
```

## Renaming Trees and Lists

```
labels (Leaf x)   = [x]
labels (Node l r) = labels l ++ labels r

numberList []     = pure []
numberList (x:xs) = pure (:) <*> fresh <*> numberList xs

numberTree (Leaf _)   = pure Leaf <*> fresh
numberTree (Node l r) = pure Node <*> numberTree l <*> numberTree r
```

## Renaming Trees and Lists

```
labels (Leaf x)   = [x]
labels (Node l r) = labels l ++ labels r

numberList []     = pure []
numberList (x:xs) = pure (:) <*> fresh <*> numberList xs

numberTree (Leaf _)   = pure Leaf <*> fresh
numberTree (Node l r) = pure Node <*> numberTree l <*> numberTree r
```

### Proposition

```
pure labels <*> numberTree t = numberList (labels t)
```

Proof by induction on *t*. Leaf case:

```
pure labels <*> (pure Leaf <*> fresh) = pure (:) <*> fresh <*> pure []
```

Compare with

```
labels (Leaf x) = (:) x []
```

## A Proof Method for Isabelle

### Project Goal

Implement a proof method for Isabelle/HOL which lifts equations to applicative functors.

$$\text{base equation} \Longrightarrow \text{lifted equation}$$

## A Proof Method for Isabelle

### Project Goal

Implement a proof method for Isabelle/HOL which lifts equations to applicative functors.

$$\text{base equation} \Longrightarrow \text{lifted equation}$$

- Proof method: User interface for goal state transformation
- Applicative functor or *idiom* given by
  - type constructor
  - constants pure, $\diamond$ (<*> in Haskell)
  - proofs of applicative functor laws

# A Proof Method for Isabelle

## Project Goal

Implement a proof method for Isabelle/HOL which lifts equations to applicative functors.

base equation $\Longrightarrow$ lifted equation

- Proof method: User interface for goal state transformation
- Applicative functor or *idiom* given by
  - type constructor
  - constants pure, $\diamond$ (<*> in Haskell)
  - proofs of applicative functor laws
- Lifting an equation ($x$ is a variable): $[\,] \mathbin{@} x = x$

$$\begin{array}{lrcrcrcl} \text{base equation:} & \mathit{append} & & [\,] & & x & = & x \\ \Longrightarrow & \text{pure}\ \mathit{append} & \diamond & \text{pure}\,[\,] & \diamond & x & = & x \end{array}$$

## Overview of Operation

Input: Lifted equation

$$e_1 = e_2$$

Transform into *canonical forms*

$$\Longleftarrow \quad \text{pure}\, f \diamond x_1 \diamond \ldots \diamond x_n = \text{pure}\, g \diamond x_1 \diamond \ldots x_n$$

## Overview of Operation

Input: Lifted equation

$$e_1 = e_2$$

Transform into *canonical forms*

$$\Longleftarrow \quad \text{pure}\, f \diamond x_1 \diamond \ldots \diamond x_n = \text{pure}\, g \diamond x_1 \diamond \ldots x_n$$

Reduce by congruence

$$\Longleftarrow \quad f = g$$

Extensionality

$$\Longleftarrow \quad \forall y_1 \ldots y_n. \quad f y_1 \ldots y_n = g y_1 \ldots y_n$$

## Hinze's Lemmas

### Lemma (Normal Form)

*Let $e$ be an idiomatic term with variables $x_1, \ldots, x_n$, from left to right. There exists $f$ such that*

$$e = \text{pure } f \diamond x_1 \diamond \ldots \diamond x_n.$$

Can lift equations

1. where both sides have the same list of variables, and
2. no variable is repeated.

R. Hinze, "Lifting Operators and Laws." 2010. Retrieved June 6, 2015,
`http://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf`

## Hinze's Lemmas

### Lemma (Normal Form)

*Let $e$ be an idiomatic term with variables $x_1, \ldots, x_n$, from left to right. There exists $f$ such that*

$$e = \text{pure } f \diamond x_1 \diamond \ldots \diamond x_n.$$

Can lift equations

1. where both sides have the same list of variables, and
2. no variable is repeated.

### Lemma (Lifting Lemma, modified)

*Let $e'$ be the lifted term of $e$, with variables $x_1, \ldots, x_n$. If the idiom satisfies additional properties, then*

$$e' = \text{pure } e \diamond x_1 \diamond \ldots \diamond x_n.$$

Lifts any equation, but not to all applicative functors.

R. Hinze, "Lifting Operators and Laws." 2010. Retrieved June 6, 2015, `http://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf`

## Combinatory Logic

- Eliminate variables from terms, introduce combinator constants
- BCKW system is equivalent to lambda calculus

$$\begin{aligned}\mathbf{B}gfx &= g(fx)\\ \mathbf{C}fxy &= fyx\\ \mathbf{K}xy &= x\\ \mathbf{W}fx &= fxx\end{aligned}$$

## Combinatory Logic

- Eliminate variables from terms, introduce combinator constants
- BCKW system is equivalent to lambda calculus

$$\begin{aligned}\mathbf{B}gfx &= g(fx)\\ \mathbf{C}fxy &= fyx\\ \mathbf{K}xy &= x\\ \mathbf{W}fx &= fxx\end{aligned}$$

- If not all combinators are available, not all terms can be represented
- Conversion by *bracket abstraction* algorithms

## Fancier Idioms

- Some idioms satisfy additional laws, one or more of

$$\begin{array}{ll}
\text{(c)} & \text{pure}\,\mathbf{C} \diamond f \diamond x \diamond y = f \diamond y \diamond x \\
\text{(k)} & \text{pure}\,\mathbf{K} \diamond x \diamond y = x \\
\text{(w)} & \text{pure}\,\mathbf{W} \diamond f \diamond x = f \diamond x \diamond x
\end{array}$$

- Hinze's Lifting Lemma requires all three
- Examples
  - state monad: none
  - set (application via Cartesian product): (c)
  - sum type, e.g. `Either`: (w)
  - option/`Maybe`: (c), (w)
  - environment functor, streams: (c), (k), (w)

## Lifting Bracket Abstraction

- Assume an applicative functors satisfies (c)

$$\begin{array}{rlcrl} & \lambda xy.\, x(fy) & & & x \diamond (\mathrm{pure}\, f \diamond y) \\ = & \mathbf{CB}f & & = & \mathrm{pure}\,\mathbf{C} \diamond \mathrm{pure}\,\mathbf{B} \diamond \mathrm{pure}\, f \diamond x \diamond y \\ & & & = & \mathrm{pure}\,(\mathbf{CB}f) \diamond y \diamond x \\ & & & = & \mathrm{pure}\,(\lambda xy.\, x(fy)) \diamond y \diamond x \end{array}$$

- We obtain a canonical form if the base term is representable

## Lifting Bracket Abstraction

- Assume an applicative functors satisfies (c)

$$\begin{array}{rlrl} & \lambda xy.\, x(fy) & & x \diamond (\mathrm{pure}\, f \diamond y) \\ = & \mathbf{CB}f & = & \mathrm{pure}\, \mathbf{C} \diamond \mathrm{pure}\, \mathbf{B} \diamond \mathrm{pure}\, f \diamond x \diamond y \\ & & = & \mathrm{pure}\, (\mathbf{CB}f) \diamond y \diamond x \\ & & = & \mathrm{pure}\, (\lambda xy.\, x(fy)) \diamond y \diamond x \end{array}$$

- We obtain a canonical form if the base term is representable
- Variable order for idiomatic terms? Remember

$$\mathrm{pure}\, f \diamond x_1 \diamond \ldots \diamond x_n = \mathrm{pure}\, g \diamond x_1 \diamond \ldots x_n$$

## Usage (1)

```
applicative state
for
  pure: Pair
  ap: "ap_state :: ('s ⇒ ('a ⇒ 'b) × 's) ⇒ ('s ⇒ 'a × 's) ⇒ 's ⇒ 'b × 's"
unfolding ap_state_def
by (auto split: split_split)
```

# Usage (2)

```
lemma "Pair labels ◇ number_tree t = number_list (labels t)"
proof (induction t)
  case (Leaf x)
  have "Pair labels ◇ (Pair Leaf ◇ fresh) = Pair op # ◇ fresh ◇ Pair []"
    by applicative_lifting simp
  thus ?case by simp
next
  case (Node l r)
  let ?ll = "Pair labels ◇ number_tree l"
  let ?lr = "Pair labels ◇ number_tree r"
  have "Pair labels ◇ (Pair Node ◇ number_tree l ◇ number_tree r) = Pair op @ ◇ ?ll ◇ ?lr"
    by applicative_lifting simp
  thus ?case using Node.IH by (simp add: label_append)
qed
```

## Conclusion

- Implemented applicative lifting in Isabelle/HOL
- Extended Hinze's results with bracket abstraction
- Use case: Algebra lifted to streams and infinite trees

Questions?

## Applicative Functor Laws

(identity) $$\mathrm{pure}\,\mathbf{I} \diamond u = u$$

(composition) $$\mathrm{pure}\,\mathbf{B} \diamond u \diamond v \diamond w = u \diamond (v \diamond w)$$

(homomorphism) $$\mathrm{pure}\,f \diamond \mathrm{pure}\,x = \mathrm{pure}\,(fx)$$

(interchange) $$u \diamond \mathrm{pure}\,x = \mathrm{pure}\,(\lambda f.\,fx) \diamond u$$

# What are the Variables?

- Remember that both canonical forms need the same variable lists:

$$\text{pure}\, f \diamond x_1 \diamond \ldots \diamond x_n = \text{pure}\, g \diamond x_1 \diamond \ldots x_n$$

- Must be able to represent terms with available combinators
- Instantiation:

$$\begin{aligned} & \forall xy.\ \text{pure}\, f \diamond x \diamond y = \ldots \\ \Longrightarrow\quad & \forall z.\quad \text{pure}\, f \diamond z \diamond z = \ldots \end{aligned}$$

  What if we want to prove the latter, but can only represent the former?
- Algorithm depends on available combinators, partially a heuristic