From Trees to Graphs and Back Again

Stefan Holdermans

Dutch FP Day 2011
January 7, 2011

Paradijslaan 28
5611 KN Eindhoven
The Netherlands
E-mail: stefan@vectorfabrics.com

Trees and graphs

- Tree structures are ubiquitous in functional programming.
- Some applications are, however, best served by manipulating proper graphs rather than trees.
- Programming with graphs is far more involved than programming with trees.
- Can we *write* programs over trees and *apply* them to graphs?

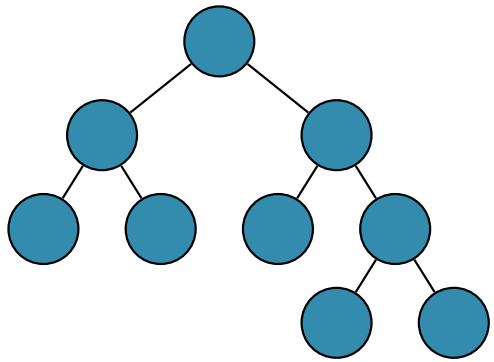
Example
An EDSL for simple expressions

```
data Expr = Const Int          -- integer constant
          | Add Expr Expr      -- addition
```

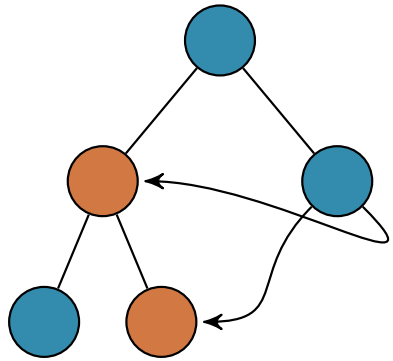
```
eval :: Expr → Int

eval (Const n)   = n
eval (Add e1 e2) = eval e1 + eval e2
```

Example
The fourth Fibonacci number, twice

```
e :: Expr
e = Add e1 e2
  where
    e1 = Add (Const 0) (Const 1)
    e2 = Add (Const 1) e3
    e3 = Add (Const 0) (Const 1)
```



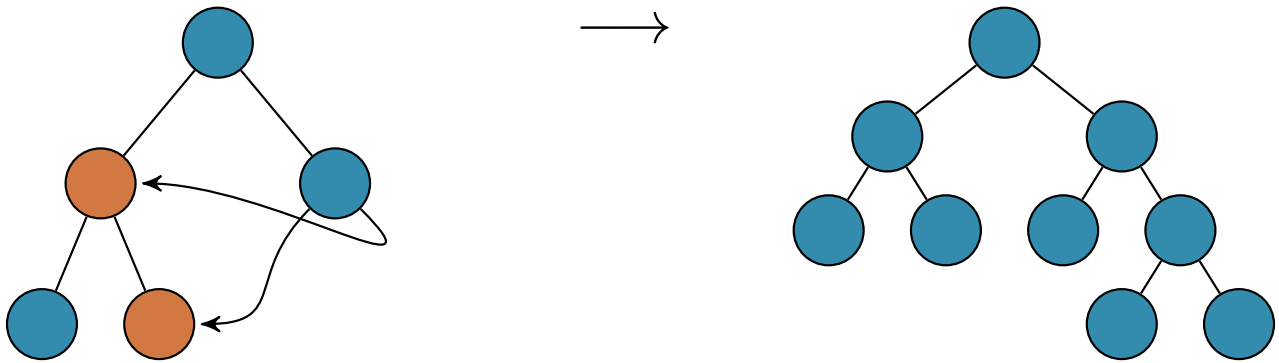
```
e' :: Expr
e' = Add e4 e5
  where
    e4 = Add (Const 0) e6
    e5 = Add e6 e4
    e6 = Const 1
```



Example
Sharing is nonobservable

```
mapConst :: (Int → Int) → Expr → Expr
mapConst f (Const n)     = Const (f n)
mapConst f (Add e1 e2) = Add (mapConst f e1) (mapConst f e2)
```

```
e'' :: Expr
e'' = mapConst id e'
```



Observable sharing
Explicit graph structures

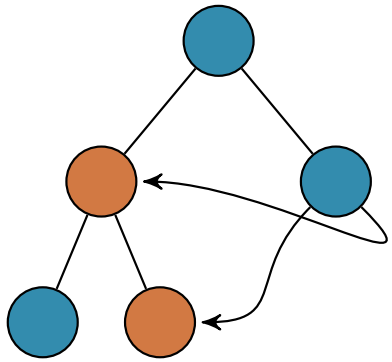
```
newtype Ref = Ref {unRef :: Int} deriving (Eq, Ord)

zero :: Ref
zero = Ref 0

succ :: Ref → Ref
succ (Ref n) = Ref (n + 1)
```

```
data ExprNode  = Const' Int | Add' Ref Ref
data ExprGraph = ExprGraph {root :: Ref, env :: Map Ref ExprNode}
```

Observable sharing

The fourth Fibonacci number, again

```
g :: ExprGraph
g = ExprGraph
        {root = Ref 4, env = fromList nodes}
  where
    nodes =
      [ (Ref 0, Const' 0)
      , (Ref 1, Const' 1)
      , (Ref 2, Add' (Ref 0) (Ref 1))
      , (Ref 3, Add' (Ref 1) (Ref 2))
      , (Ref 4, Add' (Ref 2) (Ref 3))
      ]
```



Observable sharing
Programming with graphs

```
mapConst′ :: (Int → Int) → ExprGraph → ExprGraph
mapConst′ f (ExprGraph root env) = ExprGraph root env′
  where
    env′ = fmap mp env
    mp (Const′ n)   = Const′ (f n)
    mp (Add′ r1 r2) = Add′ r1 r2
```

```
eval′ :: ExprGraph → Int
eval′ (ExprGraph root env) = go root
  where
    go r = fromJust (lookup r env′)
    env′ = fmap evl env
    evl (Const′ n)   = n
    evl (Add′ r1 r2) = go r1 + go r2
```

A universe of polynomial types

A *universe* for datatype-generic programming consists of:

1. A family of *codes* for datatypes.
2. An *interpretation* of codes in the host language.

A universe of polynomial types
Codes

```
infixr 6 :+:
infixr 7 :*:

data Id          -- recursive positions
data Unit        -- nullary constructors
data K a         -- base type
data φ :+: ψ     -- choice between constructors
data φ :*: ψ     -- multiple constructor arguments
```

A universe of polynomial types
Interpreter

```
data El φ a where
  Id    :: a → El Id a
  Unit  :: El Unit a
  Int   :: Int → El (K Int) a
  Char  :: Char → El (K Char) a
  Inl   :: El φ a → El (φ :+: ψ) a
  Inr   :: El ψ a → El (φ :+: ψ) a
  (:*:) :: El φ a → El ψ a → El (φ :*: ψ) a
```

```
instance Functor (El φ) where
  fmap f (Id x)       = Id (f x)
  fmap f Unit         = Unit
  fmap f (Int n)      = Int n
  fmap f (Char c)     = Char c
  fmap f (Inl xs)     = Inl (fmap f xs)
  fmap f (Inr ys)     = Inr (fmap f ys)
  fmap f (xs :*: ys) = fmap f xs :*: fmap f ys
```

A universe of polynomial types
Encoding datatypes

A type a is polynomial if it isomorphic to the fixed point of a polynomial functor El φ.

```
newtype Fix f = In {out :: f (Fix f)}
```

$$
\begin{aligned}
a &\cong \mathsf{Fix}\ (\mathsf{El}\ \varphi) \\
&\cong \mathsf{El}\ \varphi\ (\mathsf{Fix}\ (\mathsf{El}\ \varphi)) \\
&\cong \mathsf{El}\ \varphi\ a
\end{aligned}
$$

```
class Polynomial a where
  type U a
  out :: a → F a a
  inn :: F a a → a
type F a = El (U a)
```

A universe of polynomial types
Example

```
instance Polynomial Expr where
  type U Expr = K Int :+: Id :*: Id

  out (Const n)   = const n
  out (Add e1 e2) = add e1 e2

  inn = expr Const Add
```

```
{- smart constructors -}
const :: Int → F Expr a
const n = Inl (Int n)

add :: a → a → F Expr a
add x y = Inr (Id x :*: Id y)
```

```
{- smart destructor -}
expr :: (Int → b) → (a → a → b) → F Expr a → b
expr f op (Inl (Int n))          = f n
expr f op (Inr (Id x :*: Id y)) = x `op` y
```

A universe of polynomial types
Generic equality

```
instance Eq a ⇒ Eq (El φ a) where
  Id x₁          ≡ Id x₂          = x₁ ≡ x₂
  Unit           ≡ Unit           = True
  Int n₁         ≡ Int n₂         = n₁ ≡ n₂
  Char c₁        ≡ Char c₂        = c₁ ≡ c₂
  Inl xs₁        ≡ Inl xs₂        = xs₁ ≡ xs₂
  Inr ys₁        ≡ Inr ys₂        = ys₁ ≡ ys₂
  xs₁ :*: ys₁ ≡ xs₂ :*: ys₂ = xs₁ ≡ xs₂ ∧ ys₁ ≡ ys₂
  _              ≡ _              = False
```

```
instance Eq Expr where
  e₁ ≡ e₂ = out e₁ ≡ out e₂
```

A universe of polynomial types

Generic ordering

```
instance Ord a ⇒ Ord (El φ a) where
  compare (Id x1) (Id x2)                          = compare x1 x2
  compare Unit Unit                                = EQ
  compare (Int n1) (Int n2)                        = compare n1 n2
  compare (Char c1) (Char c2)                      = compare c1 c2
  compare (Inl xs1) (Inl xs2)                      = compare xs1 xs2
  compare (Inl xs1) (Inr ys2)                      = LT
  compare (Inr ys1) (Inl xs2)                      = GT
  compare (Inr ys1) (Inr ys2)                      = compare ys1 ys2
  compare (xs1 :*: ys1) (xs2 :*: ys2) | xs1 ≡ xs2 = compare ys1 ys2
                                      | otherwise = compare xs1 xs2
```

```
instance Ord Expr where
  compare e1 e2 = compare (out e1) (out e2)
```

Generic recursion patterns
Catamorphisms

```
type Algebra φ b = El φ b → b

cata :: Polynomial a ⇒ Algebra (U a) b → a → b
cata φ = h
  where
    h = φ ∘ fmap h ∘ out
```

```
eval :: Expr → Int
eval = cata φ
  where
    φ = expr id (+)
```

Generic recursion patterns
Anamorphisms

```
type Coalgebra φ b = b → El φ b

ana :: Polynomial a ⇒ Coalgebra (U a) b → b → a
ana ψ = h
  where
    h = inn ∘ fmap h ∘ ψ
```

```
fibs :: Int → Expr
fibs = ana ψ
  where
    ψ 0 = const 0
    ψ 1 = const 1
    ψ n = add (n − 2) (n − 1)
```

Generic recursion patterns
Monadic cata- and anamorphisms

```
type AlgebraM φ m b     = El φ b → m b
type CoalgebraM φ m b = b → m (El φ b)
```

```
cataM :: (Monad m, Polynomial a) ⇒ AlgebraM (U a) m b → a → m b
cataM φM = h
  where
    h = (≫=φM) ∘ mapM h ∘ out
```

```
anaM :: (Monad m, Polynomial a) ⇒ CoalgebraM (U a) m b → b → m a
anaM ψM = h
  where
    h = liftM inn ∘ (≫=mapM h) ∘ ψM
```

Observable sharing, generically

```
data ExprNode  = Const' Int | Add' Ref Ref
data ExprGraph = ExprGraph {root :: Ref, env :: Map Ref ExprNode}
```

```
type Node a  = F a Ref
data Graph a = Graph {root :: Ref, env :: Map Ref (Node a)}
```

ExprNode ≅ Node Expr

ExprGraph ≅ Graph Expr

Graph construction

```
data S a =
  S {refs :: Map (Node a) Ref, nodes :: Map Ref (Node a), next :: Ref}
newtype G a b = G {runG :: S a → (b, S a)}
```

```
ref :: Node a → G a Ref
ref node = G $ λs → case lookup node (refs s) of
  Nothing →
    let r      = next s
        refs′  = insert node r (refs s)
        nodes′ = insert r node (nodes s)
        next′  = succ r
    in  (r, S refs′ nodes′ next′)
  Just r → (r, s)
```

```
reify :: G a Ref → Graph a
reify = uncurry Graph ∘ (id × nodes) ∘ flip runG (S empty empty zero)
```

Graph construction
From trees to graphs

```
instance Monad (G a) where
  return x   = G ((, ) x)
  G h >>= k = G (uncurry (runG ∘ k) ∘ h)
```

$$
\begin{aligned}
\text{ref} \quad &:: \quad \text{Node a} \to \text{G a Ref} \\
&\cong \quad \text{F a Ref} \to \text{G a Ref} \\
&\cong \quad \text{El (U a) Ref} \to \text{G a Ref} \\
&\cong \quad \text{AlgebraM (U a) (G a) Ref}
\end{aligned}
$$

```
toGraph :: Polynomial a ⇒ a → Graph a
toGraph = reify ∘ cataM ref
```

Generic recursion patterns (revisited)
Graph catamorphisms

```
cataG :: Algebra (U a) b → Graph a → b
cataG φ graph = h (root graph)
  where
    h = φ ∘ fmap h ∘ outG graph
outG :: Graph a → Ref → Node a
outG graph r = fromJust (lookup r (env graph))
```

```
eval′ :: Graph Expr → Int
eval′ = cataG φ
  where
    φ = expr id (+)
```

```
fromGraph :: Polynomial a ⇒ Graph a → a
fromGraph = cataG inn
```

Generic recursion patterns (revisited)
Graph anamorphims

```
hyloCM :: (Monad m, Ord a) ⇒ Coalgebra φ a → AlgebraM φ m b → a → m b
hyloCM ψ φM = memo h
  where
    h = fmap ((≫=φM) ∘ sequence) ∘ traverse apply ∘ ψ
```

```
anaG :: Ord b ⇒ Coalgebra (U a) b → b → Graph a
anaG ψ = reify ∘ hyloCM ψ ref
```

```
fibs′ :: Int → Graph Expr
fibs′ = anaG ψ
  where
    ψ 0 = const 0
    ψ 1 = const 1
    ψ n = add (n − 2) (n − 1)
```

Summary

- Graph representations for observable sharing can be defined generically for all polynomial types.
- Maximal sharing and dropping observability can be implemented as datatype-generic functions.
- Lifting a cata- or anamorphism over a tree structure to a computation over a graph structure amounts to replacing the corresponding recursion combinator.

Further work

- Using generic *generalised tries* in both graph construction and memoisation.
- Extending the universe to larger classes of types (regular types, systems of mutually recursive types, . . .).
- Porting the library to Caml.
- Using the library in a real-world application.

Appendix
Effectful traversals

```
instance Foldable (El φ) where
  fold (Id m)      = m
  fold Unit        = mempty
  fold (Int n)     = mempty
  fold (Char c)    = mempty
  fold (Inl ms)    = fold ms
  fold (Inr ks)    = fold ks
  fold (ms :*: ks) = fold ms `mappend` fold ks
```

```
instance Traversable (El φ) where
  traverse f (Id x)      = pure Id ⊛ f x
  traverse f Unit        = pure Unit
  traverse f (Int n)     = pure (Int n)
  traverse f (Char c)    = pure (Char c)
  traverse f (Inl xs)    = pure Inl ⊛ traverse f xs
  traverse f (Inr ys)    = pure Inr ⊛ traverse f ys
  traverse f (xs :*: ys) = pure (:*:) ⊛ traverse f xs ⊛ traverse f ys
```

Appendix
Hylomorphisms

```
hylo :: Coalgebra φ a → Algebra φ b → a → b
hylo ψ φ = h
  where
    h = φ ∘ fmap h ∘ ψ
```

```
hyloM :: Monad m ⇒ Coalgebra φ a → AlgebraM φ m b → a → m b
hyloM ψ φM = h
  where
    h = (=≪φM) ∘ mapM h ∘ ψ
```

Appendix
Memoisation (1)

```
newtype Memo k v a =
  Memo {runMemo :: (k → Memo k v v) → Map k v → (a, Map k v)}
```

```
apply :: Ord k ⇒ k → Memo k v v
apply k = Memo $ λap vs → case lookup k vs of
  Nothing →
    let (v, vs′) = runMemo (ap k) ap vs
    in  (v, insert k v vs′)
  Just v → (v, vs)
```

```
memo :: Ord k ⇒ (k → Memo k v v) → k → v
memo ap k = fst (runMemo (apply k) ap empty)
```

Appendix
Memoisation (2)

```
instance Functor (Memo k v) where
  fmap f (Memo h) = Memo $ λap → (f × id) ∘ h ap
```

```
instance Applicative (Memo k v) where
  pure x = Memo $ λap vs → (x, vs)
  Memo h_f ⊛ Memo h_x = Memo $ λap vs →
    let (f, vs′)  = h_f ap vs
        (x, vs″) = h_x ap vs′
    in  (f x, vs″)
```

Appendix
Memoisation (3)

```
cataC :: (Polynomial a, Ord a) ⇒ Algebra (U a) b → a → b
cataC φ = memo h
  where
    h = fmap φ ∘ traverse apply ∘ out
```

```
anaC :: (Polynomial a, Ord b) ⇒ Coalgebra (U a) b → b → a
anaC ψ = memo h
  where
    h = fmap inn ∘ traverse apply ∘ ψ
```

```
hyloC :: Ord a ⇒ Coalgebra φ a → Algebra φ b → a → b
hyloC ψ φ = memo h
  where
    h = fmap φ ∘ traverse apply ∘ ψ
```