GPU Lab

Selected use cases of structured recursion schemes

Dániel Berényi
Wigner Research Centre for Physics

Budapest Hackathon 2016
2016. 08. 07.

λ

Introduction

Wigner Research Centre for Physics:

- One of the largest research institute of the Hungarian Academy of Sciences
- Member of many important international collaborations:
  CERN LHC (particle physics), LIGO/VIRGO (gravitational waves), ESA (Rosetta mission), ITER, Jet (fusion experiments)…
- Wigner Datacenter: largest off-site compute infrastructure of the CERN

λ

Introduction

Wigner GPU Lab:

- Research and support group at the Wigner Institute providing
    - Computational resources:
      small GPU cluster and development machines from all vendors
    - Developer's assistance:
      Help researchers with programming, dev tools, recommendations
      Dissemination: annual GPU Day, lectures
    - Research/Develop scalable, generic simulations and visualizations
    - Seek and Evaluate new, emerging technologies, participate in the development of existing ones
      we're members of the Khronos OpenCL Advisory Panel, and keep annoying members of the C++ committee

KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON

λ

Introduction

Wigner GPU Lab:

Most importantly:

trying to find the best language combination that could make generic abstract mathematical simulations realized from clusters down to GPUs, while being portable, generic, user friendly, developer friendly . . .

We mostly work in *modern* C++ (14/17)
while keeping an eye on rust and often blinking at Haskell and F#

Introduction

So how Haskell comes into the picture?

λ

Introduction

GPU Lab

So how Haskell comes into the picture?

Best recent literature, community, clean syntax, good representation to think in.
Seems like an ideal entry to functional programming today.

Even more importantly:
very useful abstractions were developed for manipulating trees...

λ

Trees

Trees are everywhere:

- Document Object/Layout Models
  (HTML, XML, XAML, and their friends)
- Constructive Solid Geometry
- Binary Search trees, space partitioning
- Hierarchical data (google maps)

Trees

Trees are everywhere:

- Document Object/Layout Models (HTML, XML, XAML, and their friends)
- Constructive Solid Geometry
- Binary Search trees, space partitioning
- Hierarchical data (google maps)
- Most importantly: *abstract syntax trees*

Wigner GPU Lab

λ

Fix points in a nutshell

```
fix :: (a -> a) -> a
fix f = f (fix f) -- same as: let x = f x in x

factorial_proto :: (Integer -> Integer) -> Integer -> Integer
factorial_proto self n = if n == 0 then 1 else n * self (n-1)

factorial_proto :: Integer -> Integer
factorial = fix factorial_proto

main = print $ factorial 5
```

Fix points in a nutshell

```
fix :: (a -> a) -> a
fix f = f (fix f) -- same as: let x = f x in x

factorial_proto :: (Integer -> Integer) -> Integer -> Integer
factorial_proto self n = if n == 0 then 1 else n * self (n-1)

factorial_proto :: Integer -> Integer
factorial = fix factorial_proto

main = print $ factorial 5
```

GPU Lab

λ

Fix points in a nutshell

```
fix :: (a -> a) -> a
fix f = f (fix f) -- same as: let x = f x in x

factorial_proto :: (Integer -> Integer) -> Integer -> Integer
factorial_proto self n = if n == 0 then 1 else n * self (n-1)

factorial_proto :: Integer -> Integer
factorial = fix factorial_proto

main = print $ factorial 5
```

This is non recursive!

Fix points in a nutshell

```haskell
fix :: (a -> a) -> a
fix f = f (fix f) -- same as: let x = f x in x

factorial_proto :: (Integer -> Integer) -> Integer -> Integer
factorial_proto self n = if n == 0 then 1 else n * self (n-1)

factorial_proto :: Integer -> Integer
factorial = fix factorial_proto

main = print $ factorial 5 -- 120
```

Now this *is* recursive!

Fix points one level higher

λ

Fix points one level higher


```
-- fix point combinator at type level:
newtype Fix f = Fix (f (Fix f))

-- compare with the value level version:
fix :: (a -> a) -> a
fix f = f (fix f)
```

λ

Recursive types

Wigner GPU Lab

Recursive types: trees!

```haskell
-- sumtype node for a simple expression tree:
data ExprF r = Const Integer
             | Add r r
             | Mul r r
```

type Expr = Fix ExprF

Recursive types

Wigner GPU Lab

Recursive types: trees!

```
-- sumtype node for a simple expression tree:
data ExprF r = Const Integer
             | Add r r
             | Mul r r
```

The Const branch is the stopping point of recursion (no *r* in it!)

r will be the recursive position

type Expr = Fix ExprF

λ

Recursive types

Wigner GPU Lab

Recursive types: trees!

```
-- sumtype node for a simple expression tree:
data ExprF r = Const Integer
             | Add r r
             | Mul r r
    deriving Functor
```

type Expr = Fix ExprF

Now this *is* recursive!

λ

Recursive types

How to construct such a tree?

```
-- constant node tree
Fix $ Const 42 :: Fix ExprF

-- tree with a multiplication node and two constants
Fix $ Mul (Fix $ Const 6) (Fix $ Const 7)::Fix ExprF
```

λ

Tree Recursions

What can we do with trees?

Bottom-up consume/traverse,
Top-down create/traverse

One helper needed: unFix, revealing one level inside the tree:

```
unFix :: Fix f -> f (Fix f)
unFix (Fix x) = x
```

Tree Recursions

What can we do with trees?

Bottom-up consume/traverse,
Top-down create/traverse

One helper needed: unFix, revealing one level inside the tree:

```
unFix :: Fix f -> f (Fix f)
unFix (Fix x) = x
```

After unFix
you can use the sumtype

λ

Tree Recursions

Bottom-up: *catamorphism*

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

Top-down: *anamorphism*

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg
```

Tree Recursions


```haskell
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

These functions describe the logic to be done at a single step in the tree

```haskell
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg
```

Tree Recursions

Bottom-up: *catamorphism*

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg tree = (alg . fmap (cata alg) . unFix) tree
```

Reveal one level of the tree

Tree Recursions

Bottom-up traverse: *catamorphism*

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg tree = (alg . fmap (cata alg) . unFix) tree
```

Step into the new level and apply itself to the childs

Tree Recursions

Bottom-up traverse: *catamorphism*

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg tree = (alg . fmap (cata alg) . unFix) tree
```

After the level below are done
process this level using the results
from below

Tree Recursions

Bottom-up traverse: *catamorphism*

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg tree = (alg . fmap (cata alg) . unFix) tree
```

types at the stages:

a

f a

f (Fix f)

Fix f

Tree Recursions

Top-down traverse: *anamorphism*

```haskell
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg seed = (Fix . fmap (ana coalg) . coalg) seed
```

Fix (wrap) the level
in the tree type

step into new level
and repeat

create one level of
the tree from a seed

Tree Recursions

Bottom-up traverse: *catamorphism*

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg seed = (Fix . fmap (ana coalg) . coalg) seed
```

types at the stages:

Fix f

f (Fix f)

f a

a

Catamorphisms

Simple sample: pretty printer for the expression tree:

```
showF :: Fix ExprF -> [Char]
showF = cata alg
    where
        alg (Const x) = show x
        alg (Add x y) = "(" ++ x ++ "+" ++ y ++ ")"
        alg (Mul x y) = "(" ++ x ++ "*" ++ y ++ ")"
```

Catamorphisms

Simple sample: evaluator for the expression tree:

```
evalF :: Fix ExprF -> Integer
evalF = cata alg
    where
        alg (Const x) = x
        alg (Add x y) = x + y
        alg (Mul x y) = x * y
```

Anamorphisms

Simple sample: evaluator for the expression tree:

```haskell
-- Sumtype for the expression node types
data ExprF r = Const Integer
             | Add r r
             | Mul r r
             | Pow r r
     deriving (Show, Functor)

-- Sumtype for restrictig recursion in some cases
data WantRec = Rec | NoRec
```

Anamorphisms


```
toPowExpr :: Integer -> Integer -> Expr
toPowExpr base x = ana coAlg (x, Rec) where
 coAlg (n, NoRec)              = Const n
 coAlg (n, _) | n <= base      = Const n
 coAlg (n, _) | isPow base n =
                  Pow (base, NoRec) (intLog base n, NoRec)
 coAlg (n, _) | otherwise       = let ln = intLog base n in
                   Add (base^ln, Rec) (n - (base^ln), Rec)
```

Anamorphisms

```
toPowExpr :: Integer -> Integer -> Expr
toPowExpr base x = ana coAlg (x, Rec) where
  coAlg (n, NoRec)               = Const n
  coAlg (n, _) | n <= base       = Const n
  coAlg (n, _) | isPow base n =
                   Pow (base, NoRec) (intLog base n, NoRec)
  coAlg (n, _) | otherwise       = let ln = intLog base n in
                    Add (base^ln, Rec) (n - (base^ln), Rec)
```

Keep small constants as is and also stop if NoRec was given

Anamorphisms

```
toPowExpr :: Integer -> Integer -> Expr
toPowExpr base x = ana coAlg (x, Rec) where
 coAlg (n, NoRec)              = Const n
 coAlg (n, _) | n <= base
 coAlg (n, _) | isPow base n =
                 Pow (base, NoRec) (intLog base n, NoRec)
 coAlg (n, _) | otherwise        = let ln = intLog base n in
                  Add (base^ln, Rec) (n - (base^ln), Rec)
```

If we have a perfect power of the base create a Pow, just calculate the exponent

Anamorphisms


```
toPowExpr :: Integer -> Integer -> Expr
toPowExpr base x = ana coAlg (x, Rec) where
 coAlg (n, NoRec)               = Const n
 coAlg (n, _) | n <= base
 coAlg (n, _) | isPow base n
                 Pow (base, NoRec) (intLog base n, NoRec)
 coAlg (n, _) | otherwise     = let ln = intLog base n in
                  Add (base^ln, Rec) (n - (base^ln), Rec)
```

Else, break the number into a sum separating the largest possible whole power

λ

Anamorphisms

Example outputs:

```
putStrLn $ showF $ toPowExpr 2 31
((2^4)+((2^3)+((2^2)+(2+1))))

putStrLn $ showF $ toPowExpr 4 115
((4^3)+((4^2)+((4^2)+((4^2)+3))))
```

Tree transformations

The funny part begins, when both the inputs and the outputs are trees!

λ

Tree transformations

```haskell
-- Sumtype for the expression tree
data ExprF r = Const Integer
             | Add r r
             | Mul r r
             | Pow r r
    deriving (Show, Functor)

data WithCostF f r = WithCost (f r) Integer
    deriving Functor

cost :: WithCostF f r -> Integer
cost (WithCost _ c) = c
```

We store an additional integer in the tree

Tree transformations

```haskell
-- cost estimator for the expression tree
calculateCost :: Fix ExprF -> Fix (WithCostF ExprF)
calculateCost = cata alg
  where costF = cost . unFix
    alg :: ExprF (Fix (WithCostF ExprF)) -> Fix (WithCostF ExprF)
    alg (Const x   ) = Fix $ WithCost (Const x)    1
    alg (Add x0 y0) = Fix $ WithCost (Add x0 y0) (1  + costF x0 + costF y0)
    alg (Mul x0 y0) = Fix $ WithCost (Mul x0 y0) (2  + costF x0 + costF y0)
    alg (Pow x0 y0) = Fix $ WithCost (Pow x0 y0) (10 + costF x0 + costF y0)
```

Tree transformations

Example output with a pretty printer:

```
before: ((2^(4+3))*2)

after: ((2[1]^(4[1]+3[1])[3])[14]*2[1])[17]
```

Tree transformations

A more complicated example: simple lambda calculus evaluator

Tree transformations

```haskell
-- sumtype for the expression tree
data ExprF r = Const Integer
             | Var Char
             | Add r    r
             | Sub r    r
             | Abs Char r
             | App r    r
    deriving Functor
```

Tree transformations

```haskell
-- sumtype for the expression tree
data ExprF r = Const Integer
             | Var Char
             | Add r    r
             | Sub r    r
             | Abs Char r
             | App r    r
    deriving Functor
```

Variable, also a terminal type

Some arithmetic for fun

Lambda abstraction

Lambda application

WIGNER GPU Lab

λ

Tree transformations

```haskell
data EvalF f r = EvalF {
        expr  :: f r,

        stack :: [Fix (EvalF f)],

        env   :: [(Char, Fix (EvalF f))]
} deriving Functor
```

Tree

Unbound expression stack

Environment: bound variables

type EvalExprF = EvalF ExprF

Tree transformations

Idea: as the anamorphism descends,

- it takes the arguments from the App nodes

    and put them onto the stack

- at Abs it takes the trees from the stack and binds them to the variable
- at Var it searches the environment and replace Var with the bound tree

λ

Tree transformations

Main points:

```
coalg :: Fix EvalExprF -> EvalExprF (Fix EvalExprF)
coalg (Fix (EvalF x stack env)) =
                         case x :: ExprF (Fix EvalExprF) of

--take first (earliest) expression from the stack and bind
Abs c ex->let EvalF y _ _ = (unFix ex)
              z = Fix $ (EvalF y (tail stack) (env++[(c, head stack)]))
                  in EvalF (Abs c z) [] []
```

Tree transformations

Main points:

```haskell
coalg :: Fix EvalExprF -> EvalExprF (Fix EvalExprF)
coalg (Fix (EvalF x stack env)) =
                         case x :: ExprF (Fix EvalExprF) of
--add the current expression (right node) to the stack
App ex1 ex2 -> let EvalF x1 _ _ = (unFix ex1)
                   EvalF x2 _ _ = (unFix ex2) in

EvalF (App
       (Fix $ (EvalF x1 (xs++[Fix $ (EvalF x2 stack env)]) env))
       (Fix $ (EvalF (Const 0) [] []))
      ) [] []
```

Tree transformations

WIGNER GPU Lab

The evaluator is simpler:

```
-- algebra for evaluating an expression tree
alg :: Expr Integer -> Integer
alg (EvalF x _ _) = case x of
                Const i   -> i
                Var   c   -> error "undefined variable!"
                Add   x y -> x + y
                Sub   x y -> x - y
                Abs   s b -> b
                App   f v -> f
```

Tree transformations

Finally the whole algorithm is a *composition* of the ana and cata pass:

```
eval :: Fix Expr -> Integer
eval = cata alg

pre :: (Fix Expr) -> (Fix Expr)
pre = ana coalg

lambdaCalcEval tree = (eval . pre) tree
```

Tree transformations

WIGNER GPU Lab

Finally the whole algorithm is a *composition* of the ana and cata pass:

```
eval :: Fix Expr -> Int
eval = cata alg

pre :: (Fix Expr) -> (Fix Expr)
pre = ana coalg

lambdaCalcEval tree = (eval . pre) tree
```

This composition is called a *hylomorphism*

λ

Outlook

- How all this stuff is relevant in physics simulations?

Outlook

GPU Lab

Theorem

- Scientists are not good at programming…

Justification

- It is not their job to be…

Corollary

- The field is ful of inefficient codes…

λ

Outlook

WIGNER GPU Lab

Theorem

- Scientists are not good at programming…

Justification

- It is not their job to be…

Corollary

- The field is ful of inefficient codes…

λ

Outlook

WIGNER GPU Lab

Theorem

- Scientists are not good at programming...

Justification

- It is not their job to be...

Corollary

- The field is full of inefficient codes...

λ

Outlook

WIGNER GPU Lab

Scientists need domain specific languages to express algorithms and equations in their field -> trees...

$$-\frac{\partial B}{\partial t} = \nabla \times E$$

This high level task should be processed down in multiple steps into a *highly efficient* numerical scheme (in C++) running on clusters and GPUs and what not

We have a prototype implementation targeting lin alg implemented fully in C++ using catas

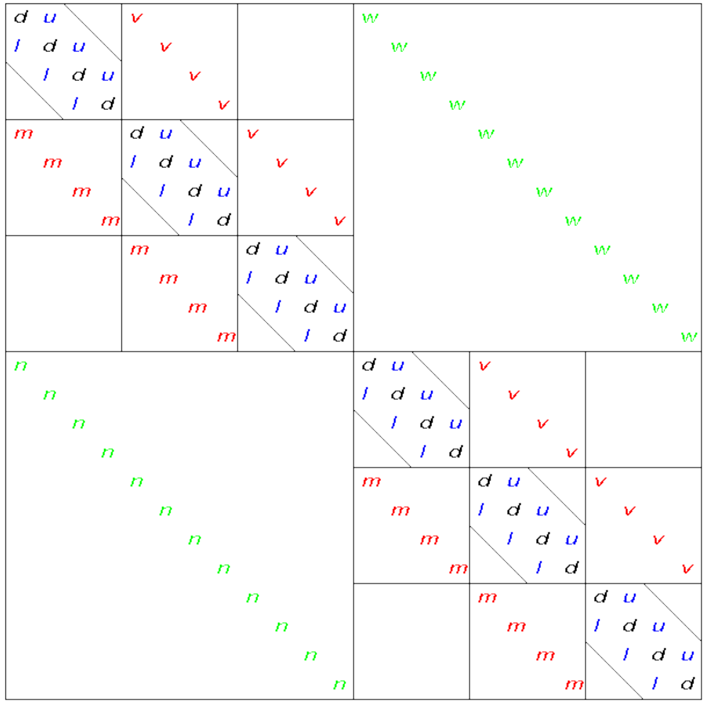
λ

Outlook

High efficiency is obligatory, since the computation demand is enormous
(simulations are usually not memory limited)

High level formulation is useful, since high level optimizations cannot be carried out at low level (information is lost downward)!

Numerical details are usually inferable from the equations and very high level field specific assumptions



Outlook

Trees emerge not just by expressions . . .

Even flat programming tasks develop trees because of the underlying hardware mechanisms:

memory limits,

caching,

hierarchical parallelism

λ

Outlook

End user:
„Give me a tool that can multiply matrices as fast as possible!”

Programmer:
„Ha, no problem, let’s use a GPU library, how big are your matrices?”

End user:

„10^10 by 10^10”

Programmer:

„Well, ... floats?”

End user:

„No way! Complex doubles!”

Outlook

GPU Lab

End user:
„Give me a tool that can multiply matrices as fast as possible!”

Programmer:
„Ha, no problem, let’s use a GPU library, how big are your matrices?”

End user:

„10^8 by 10^8”

Programmer:

„Well, ... floats?”

End user:

„No way! Complex doubles!”

λ

Outlook

End user:
„Give me a tool that can multiply matrices as fast as possible!”

Programmer:
„Ha, no problem, let’s use a GPU library, how big are your matrices?”

End user:

„10^8 by 10^8”

Programmer:

„Well, ... floats?”

End user:

„No way! Complex doubles!

Outlook

Matrix multiplication is a flat thing: $\mathrm{C}_{\mathrm{ik}} = \sum_j A_{ij} B_{jk}$

```
 map the rows of A -> r
      map the cols of B -> c
             the dot product: fold (+) 0 ( zip (*) r c )
```

λ

Outlook

Matrix multiplication is a flat thing ... except that:

- Above N ~ 4 you run out of (vector) registers
- Above N ~ 16-64 you run out of cache or local memory ~ $O(32k)$
  max # work group threads (~256 on GPUs)
- Above N ~ 16k you run out of max single memory allocation on GPU
- Above N ~ 64k you run out of max thread count in 2D
  and total video memory, may try to use multiple GPUs
- Above N ~ 100k you run out of RAM on the host,
  you need to stream to/from disk,
  you need to use distributed parallelism
- Around 10^6 you give up.
- The user comes back that he just discovered, that N ~ 10^12 would be needed due to...

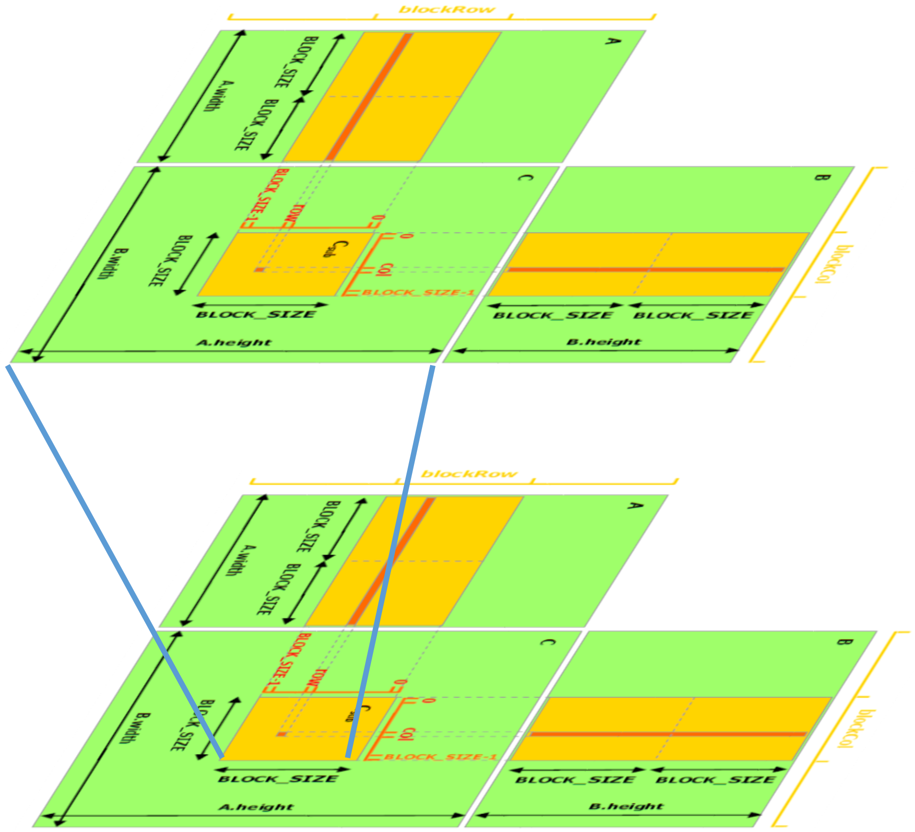
Outlook

Matrix multiplication is a flat thing ... except that:

- Above N ~ 4 — you run out of (vector) registers
- Above N ~ 16-64 — you run out of cache or local memory ~ O(32k)
max # work group threads (~256 on GPUs)
- Above N ~ 16k — you run out of max single memory allocation on GPU
- Above N ~ 64k — you run out of max thread count in 2D
and total video memory, may try to use multiple GPUs
- Above N ~ 100k — you run out of RAM on the host,
you need to stream to/from disk,
you need to use distributed parallelism
- Around 10^6 you give up.
- The user comes back that he just discovered, that N ~ 10^12 would be needed due to...

Outlook

- You need to partition into blocks at multiple levels repeatedly



Outlook

Your „flat” matrix type was:

```
a ^ (N x N)
```

But became:

```
a ^ (n1 x n1) ^ (n2 x n2) ^ ...
          where n1 x n2 x ... = N
```

voila, trees again!

Memory type and other annotations may be placed at the different levels to drive placement of blocks by cost estimation. Ideal job for cata/ana.

λ

Outlook

Usually numerical algorithms are given by imperative / procedural (pseudo)codes.

Reformulating them by recursion schemes could reduce code bloat, improve maintainability, makes optimizations easier.

λ

Outlook

The repeated tree transformations by the large zoo of recursion schemes are currently being investigated to drive annotations, optimizing replacements and code generation for such simulations to aid scientists.

link

Destruction Morphisms

catamorphism

```
cata :: ∀ a. (f a → a) → μf → a
             f-algebra
```

Also known as “fold”. Deconstructs a f-structure level-by-level and applies the algebra [13, 5, 14, 6].

paramorphism

```
para :: ∀ a. (f (μf , a) → a) → μf → a
```

A.k.a. “the Tupling-Trick”. Like cata, but allows access to the full subtree during teardown. Is a special case of zygo, with the helper being the initial-algebra [16].

zygomorphism

```
zygo :: ∀ a b. (f (a , b) → a) →
               (f b → b)        → μf → a
```

Allows depending on a helper algebra for deconstructing a f-structure. A generalisation of para.

histomorphism

```
histo :: ∀ a. (f (Cofree f a) → a) → μf → a
```

Deconstructs the f-structure with the help of all previous computation for the substructures (the trace). Difference to para: The subcomputation is already available and needs not to be recomputed.

prepromorphism

```
prepro :: ∀ a. (f a → a) → (f ↝ f) → μf → a
```

Applies the natural transformation at every level, before destructing with the algebra. Can be seen as a one-level rewrite. This extension can be combined with other destruction morphisms [4].

Construction Morphisms

anamorphism

```
ana :: ∀ a. (a → f a) → a → νf
            f-coalgebra
```

Also known as “unfold”. Constructs a f-structure level-by-level, starting with a seed and repeatedly applying the coalgebra [13, 5].

apomorphism

```
apo :: ∀ a.  (a → f (a + νf)) → a → νf
```

A.k.a. “the Co-Tupling-Trick”™. Like ana, but also allows to return an entire substructure instead of one level only. Is a special case of g-apo, with the helper being the final-coalgebra [17, 16].

g-apomorphism

```
gapo :: ∀ a b. (a → f (a + b)) →
               (b → f b)        → a → νf
```

Allows depending on a helper coalgebra for constructing a f-structure. A generalisation of apo.

futumorphism

```
futu :: ∀ a. (a → f (Free f a)) → a → νf
```

Constructs a f-structure stepwise, but the coalgebra can return multiple layers of a-valued substructures at once. Difference to apo: the subtrees can again contain as [16].

postpromorphism

```
postpro :: ∀ a. (a → f a) → (f ↝ f) → a → νf
```

Applies the natural transformation at every level, after construction with the coalgebra. Can be seen as a one-level rewrite. This extension can be combined with other construction morphisms.

Thank you for your attention!

Feed back and references are very welcome!

Further reading:

Tim Willams' talk

Edward Kmett's blog posts

Patrick Thomson's blog posts

Bartosz Milewski's blog posts

Special thanks to

Gábor Lehel