LambdaGen
A GPU Code Generator Powered by Recursion Schemes

Dániel Berényi
Wigner Research Centre for Physics
GPU Lab

András Leitereg, Gábor Lehel
Eötvös University

Budapest Haskell Hackathon 2017

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

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 on Modern Scientific Programming
  - 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

Introduction

We are always looking for tools that help to map abstract mathematical constructs to hardware

Functional programming provides a vast pool of such tools

Structures

Structures
Trees

Structures

Trees

Hierarchies

Structures

Trees

Hierarchies

Recursion

Tree structures

Physicists have lots of nice and compact equations e.g. Maxwell's eqs:

$$\partial_{[\alpha}F_{\beta\gamma]} = 0 \qquad \partial_{\alpha}F^{\alpha\beta} = \mu_0 J^{\beta}$$

that hide a tremendous amount of assumptions, constructs and abbreviations...

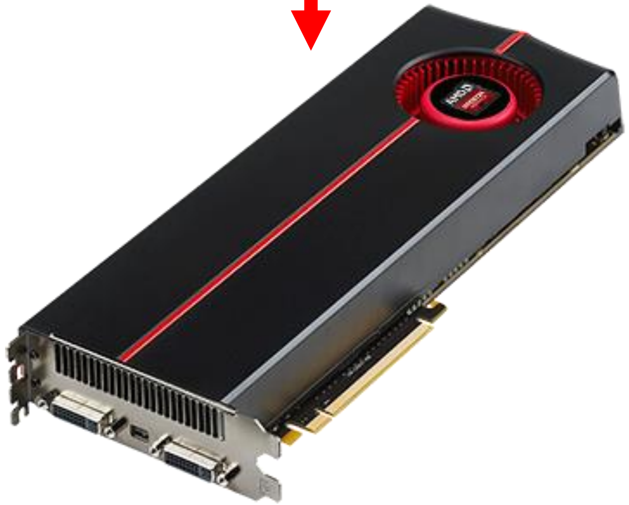
Tree structures

Many times the problems are formulated at the symbolic level

$$\partial_{[\alpha}F_{\beta\gamma]} = 0 \qquad \partial_\alpha F^{\alpha\beta} = \mu_0 J^\beta$$

How to get there?

But require low-level optimal codes on the hardware



Tree structures

Usually the path involves many steps and tools:

$$\partial_{[\alpha}F_{\beta\gamma]} = 0 \qquad \partial_\alpha F^{\alpha\beta} = \mu_0 J^\beta$$

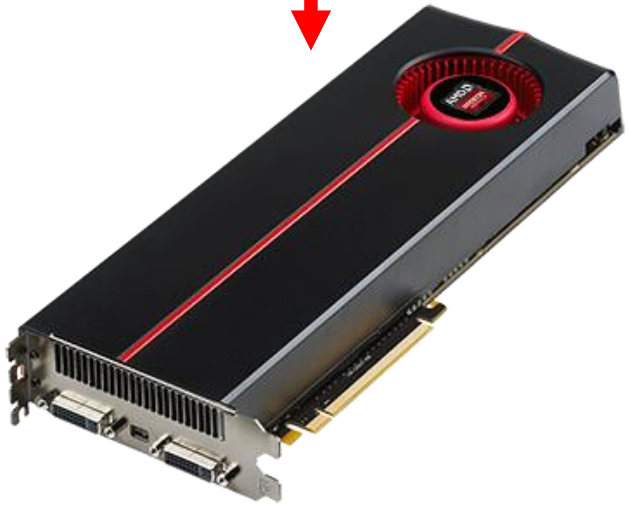
How to get there?

Symbolic math expressions

Dense or sparse linear algebraic expressions

High-level programming terms

Low-level programming terms



Tree structures

Usually the path involves many steps and tools:

Symbolic math expressions

Dense or sparse linear algebraic expressions

High-level programming terms

Low-level programming terms

Symbolic transformations, differentiation, integration, simplification, substitution

Tree structures

Usually the path involves many steps and tools:

Symbolic math expressions

Dense or sparse linear algebraic expressions

High-level programming terms

Low-level programming terms

Numerical solver choices based on the structures of equations, matrix-tensor manipulations, decomposition into well-known primitive operations

Tree structures

Usually the path involves many steps and tools:

Symbolic math expressions

Dense or sparse linear algebraic expressions

High-level programming terms

Low-level programming terms

Types
Higher-order functions
Fusion-Fission
Specialization of generic algorithms

Tree structures

Usually the path involves many steps and tools:

Symbolic math expressions

Dense or sparse linear algebraic expressions

High-level programming terms

Low-level programming terms

memory layout & usage, temporaries, scheduling...

Tree structures

So there are trees everywhere:

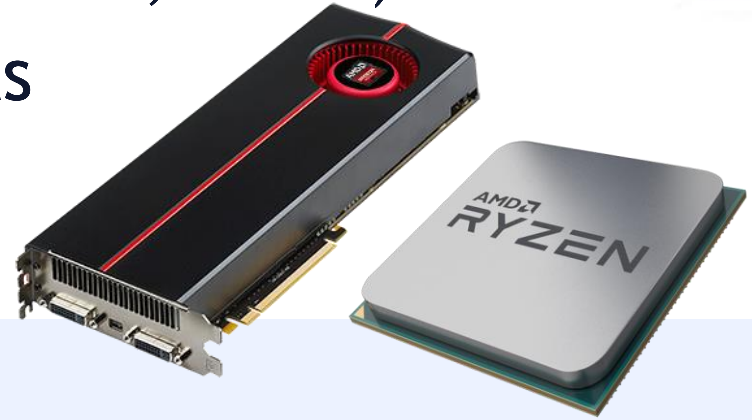
- Expression trees (symbolic or program)
- Abstract Syntax Trees
  (often code generation and code transformations are needed)
- Nested generic algorithms
- Nested data structures
- Hardware hierarchies

Tree structures

Hardware hierarchies:

Hierarchical parallelism:

- Computing center
- Clusters of computers
- Multiple devices (CPU, GPU, FPGA)
- Multiple execution units
- Groups of threads



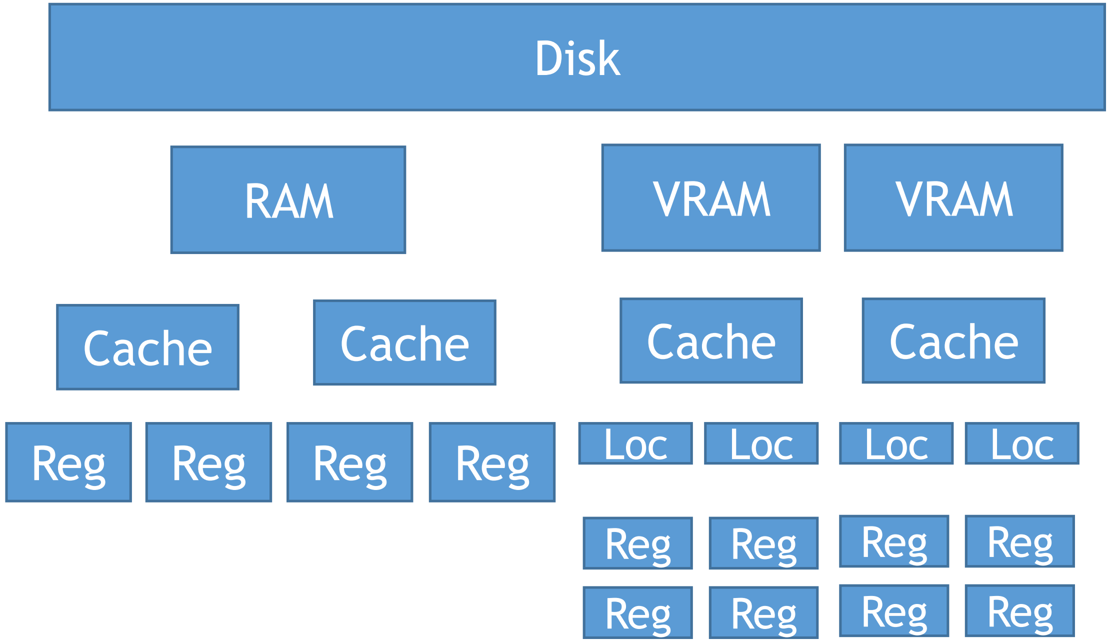
Tree structures

Hardware hierarchies:

Memory hierarchy:

Speed ↓ / Size ↑

- Storage (disk, tape)
- Device memory (RAM, VRAM)
- Caches
- Memory shared by threads
- Registers



So how to deal with all these trees?

Recursion

All these trees are made from the same building blocks (branches and leaves) repeatedly under each other

→ trees are recursive structures

Primitive Recursion

```haskell
data Tree = Empty | Branch Tree Tree

depth :: Tree -> Int
depth (Empty) = 0
depth (Branch l r) = 1 + max (depth l) (depth r)

d0 = Empty
d1 = Branch Empty Empty
d2 = Branch (Branch Empty Empty) Empty

main = print $ depth d2
```

Primitive Recursion

```haskell
data Tree = Empty | Branch Tree Tree

depth :: Tree -> Int
depth (Empty) = 0
depth (Branch l r) = 1 + max (depth l) (depth r)

d0 = Empty
d1 = Branch Empty Empty
d2 = Branch (Branch Empty Empty) Empty

main = print $ depth d2
```

Primitive Recursion
The defined name also appears on the rhs

Primitive Recursion

Problems of this approach:

- Each recursive function had to be written with the particular datatype in mind (pattern match)
- Possible packing and unpacking of data transferred between levels is on the writer of the recursive function
- Same recursive logic might get implemented multiple times

Structured Recursion

```haskell
{-# LANGUAGE DeriveFunctor #-}
newtype Fix f = Fix (f (Fix f)) -- Fix point combinator

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

data TreeF r = Empty | Branch r r
deriving (Show, Functor)

type Tree = Fix TreeF

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

Structured Recursion

```haskell
{-# LANGUAGE DeriveFunctor #-}
newtype Fix f = Fix (f (Fix f)) -- Fix point combinator

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

data TreeF r = Empty | Branch r r
deriving (Show, Functor)

type Tree = Fix TreeF

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

We write the recursion logic once,
this is the only place where the same name appears on the rhs too.

Structured Recursion

```haskell
depth2 :: Tree -> Int
depth2 = cata alg
    where
        alg (Empty) = 0
        alg (Branch l r) = 1 + max l r

d0 = Fix $ Empty
d1 = Fix $ (Branch d0 d0)
d2 = Fix $ (Branch (Fix $ Branch d0 d0) d0)

main = print $ depth2 d2
```

Structured Recursion

```haskell
depth2 :: Tree -> Int
depth2 = cata alg
    where
        alg (Empty) = 0
        alg (Branch l r) = 1 + max l r

d0 = Fix $ Empty
d1 = Fix $ (Branch d0 d0)
d2 = Fix $ (Branch (Fix $ Branch d0 d0) d0)

main = print $ depth2 d2
```

Structured Recursion

This kind of traversal only depends on the evaluated results of the branches immediately below!

depth2 will not call itself directly!

Structured Recursion

Why is this better?

- Separation of the recursion logic from the data type
- Parametrization over the recursive datatype and application logic
- More clearly expresses intent

Structured Recursion

Catamorphism:

Deconstruct a fixedpoint structure (Functor f)

into a summary value of type a:

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

Structured Recursion

Catamorphism:

Deconstruct a fixedpoint structure (Functor f) into a summary value of type a:

The deconstruction logic (algebra) receives an instance of the same functor with the subresults

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

Structured Recursion

Anamorphism:

Dual of catamorphism: constructs a structure from a seed

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

Structured Recursion

Anamorphism:

Dual of catamorphism: constructs a structure from a seed

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

The construction logic (coalgebra) receives a seed value of type a and creates an instance of the same functor with the new seeds

Structured Recursion

Anamorphism:

Task: create a tree with given depth

```
plant :: Int -> Tree
plant = ana coalg
      where
          coalg 0 = Empty
          coalg n = Branch (n-1) (n-1)
```

Structured Recursion

Paramorphism (Generalization of the catamorphism):

Makes the original structure (before evaluation) available for the operation *beside* the evaluated sub-results:

```
para :: (Functor f) => (f (Fix f, a) -> a) -> Fix f -> a
para ralg = ralg . fmap fanout . unFix
          where  --fanout :: Fix f -> (Fix f, b)
                 fanout t = (t, para ralg t)
```

Structured Recursion

Paramorphism (Generalization of the catamorphism):

Same deconstruction like cata

```
para :: (Functor f) => (f (Fix f, a) -> a) -> Fix f -> a
para ralg = ralg . fmap fanout . unFix
          where  --fanout :: Fix f -> (Fix f, b)
                 fanout t = (t, para ralg t)
```

Structured Recursion

Wigner GPU Lab

Paramorphism (Generalization of the catamorphism):

But the operation receives a structure of pairs:
where the left is the original subtree,
and the right is the subresult

```
para :: (Functor f) => (f (Fix f, a) -> a) -> Fix f -> a
para ralg = ralg . fmap fanout . unFix
          where  --fanout :: Fix f -> (Fix f, b)
                 fanout t = (t, para ralg t)
```

Structured Recursion

Paramorphism:

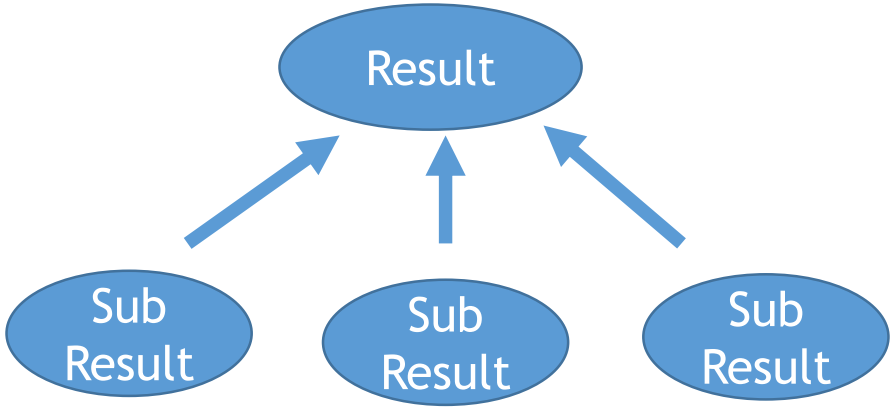
Task: count all half branches in the tree:


```
count :: Tree -> Int
count = para alg
   where
      alg Empty = 0
      alg (Branch (Fix Empty, _)        (Fix Empty, _)       ) = 0
      alg (Branch (Fix (Branch _ _), n) (Fix Empty, _)       ) = 1+n
      alg (Branch (Fix Empty, _)        (Fix (Branch _ _), n)) = n+1
      alg (Branch (Fix (Branch _ _), n) (Fix (Branch _ _), m)) = n+m
```

Structured Recursion – Use Cases

Use cases:

- Catamorphism
  - Decomposing structures level by level
  - Bottom-up traversal of a tree
  - Changing the expression structure type

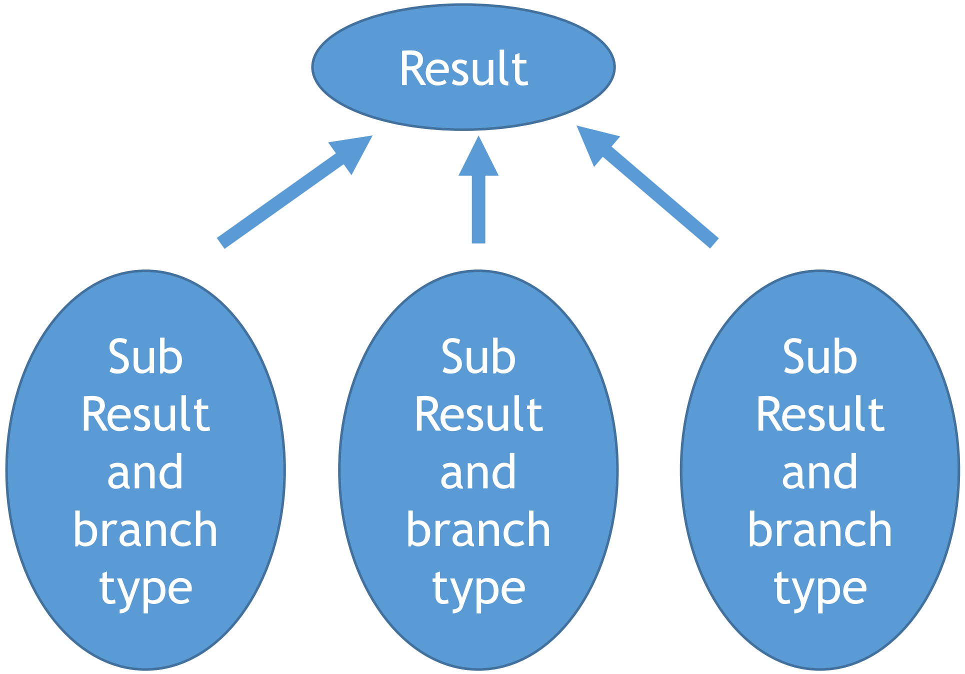


Evaluating, serializing expression trees

Calculating quantities bottom-up

Componation and combination identities

Structured Recursion – Use Cases



Use cases:

- Paramorphism
  - Decomposing structures level by level
  - Bottom-up traversal of a tree
  - Changing the expression structure type

Same things as cata, but ! we can depend on the structure of the subresults too!

Structured Recursion – Use Cases

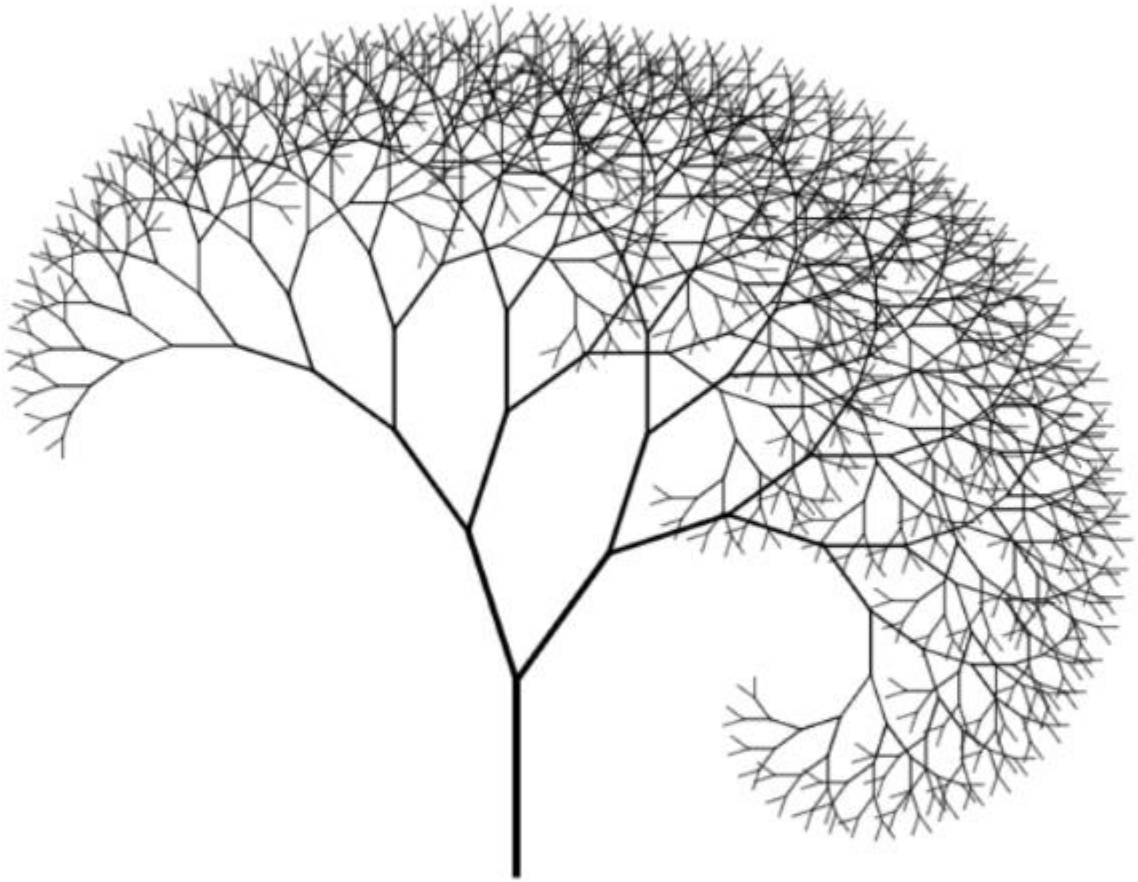
Use cases:

- Anamorphism
  - Building structures level by level
  - Top-down traversal of a tree

De-serializing expression trees

Calculating quantities top-down

Create structure from seed



Structured Recursion

More examples for the case of expression tree manipulations: see last year

Structured Recursion

OK, so what is the hype for?

We just gave a fancy name for some frequent algorithm patterns...

Structured Recursion

OK, so what is the hype for?

We just gave a fancy name for some frequent algorithm patterns...

Not exactly...

Structured Recursion

Recursion schemes root in category theory, and form a nice hierarchy (khm☺) of more and more generic traversal patterns.

The theory gives also combination and fusion theorems that can be used to optimize composite expressions of recursion schemes.

Structured Recursion

A collection of recursion schemes is assembled from the literature by Edward Kmett and implemented in Haskell.

Fold Schemes	Description
Catamorphism	Consume structure level by level
Paramorphism	Consume with primitive recursion
Zygomorphism	Consume with the aid of a helper function
Histomorphism	Consume, possibly multiple levels at once
Prepromorphism	Consume, by repeatedly applying a natural transformation

Unfold Schemes	Description
Anamorphism	Create structure level by level
Apomorphism	Create structure, may stop and return with a branch or level
Futumorphism	Create structure, possibly multiple levels at once
Postpro-morphism	Create, by repeatedly applying a natural transformation

Structured Recursion

Combinations of generalized folds and unfolds are also available:

e.g.:

Hylomorphism = ana + cata

It can be used to build numerical quadrature algorithms

Mutumorphism: mutual recursive pair of functions

Can be used to create adaptive ordinary differential equation solvers

Structured Recursion

Switching from primitive recursion to structured recursion is just like switching from `goto` and explicit loops to packaged generic algorithms

They help seeing the pattern and solve complex problems.

Structured Recursion

An example outside of functional programming...

Neural Networks

Neural Networks become new hype train in computing

Cheap, high-performance computing and large annotated datasets contributed to their success

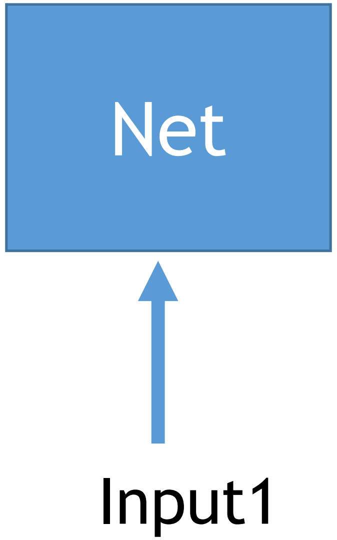
One particularly important case of them are Recurrent Neural Networks, that are very good at natural language processing, handwriting, speech processing, recognition and generation.

Recurrent Neural Networks

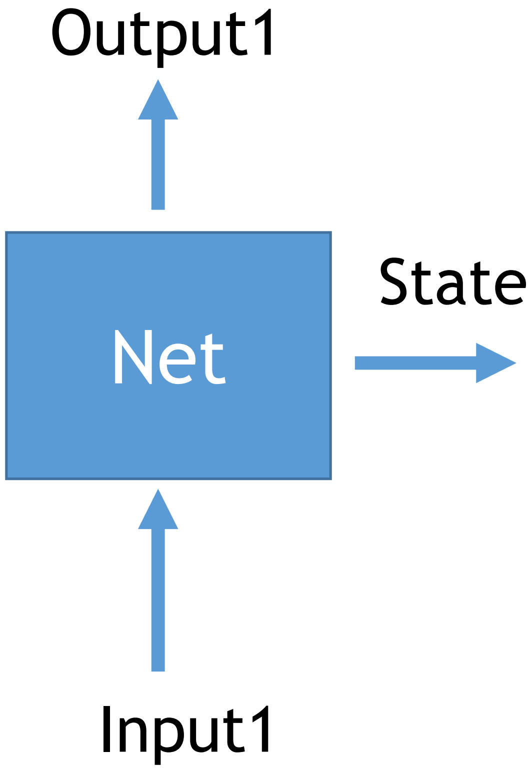
- has a set of weights that describe a nonlinear transformation on an input
- and transform a sequential dataset by repeatedly applying itself on the inputs and the previous self state and/or output

Recurrent Neural Networks

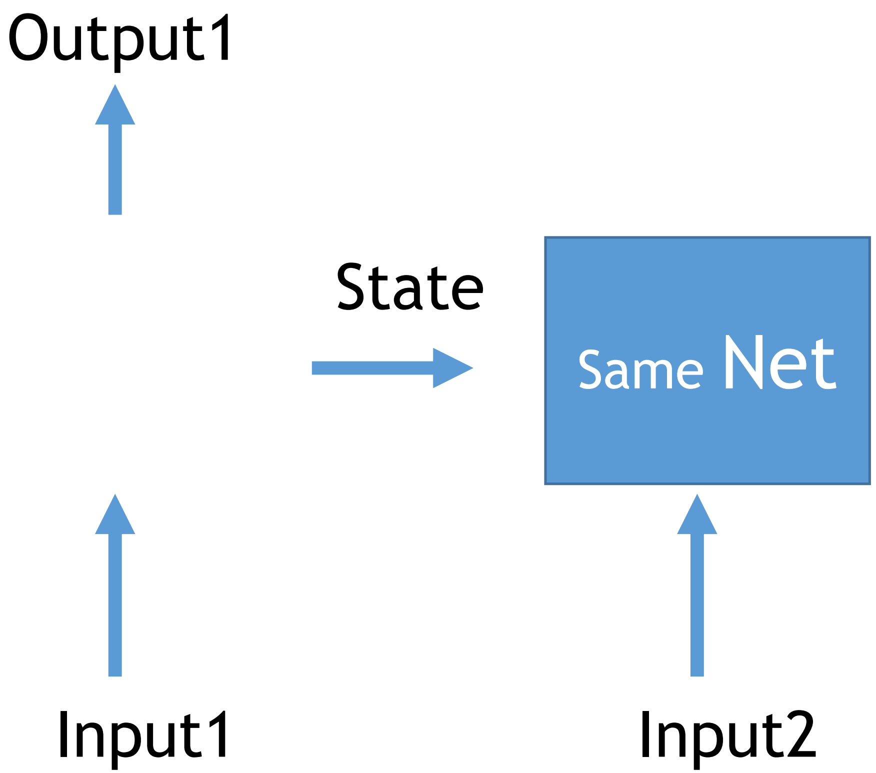
Recurrent Neural Networks



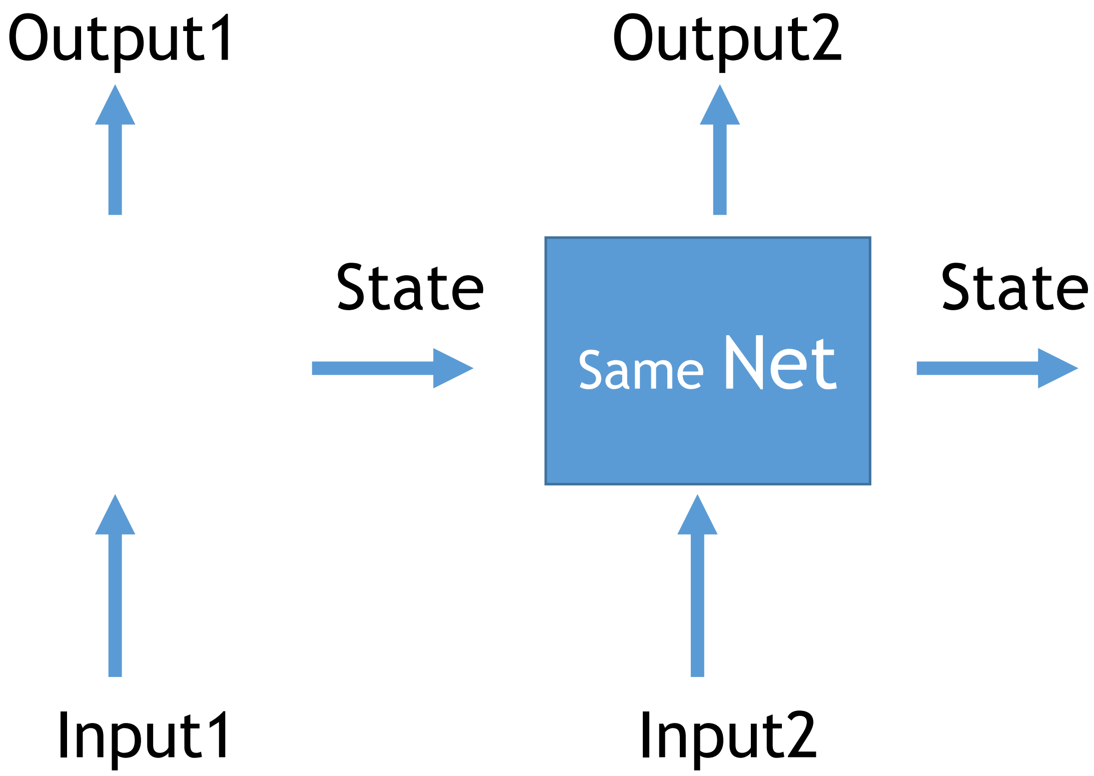
Recurrent Neural Networks



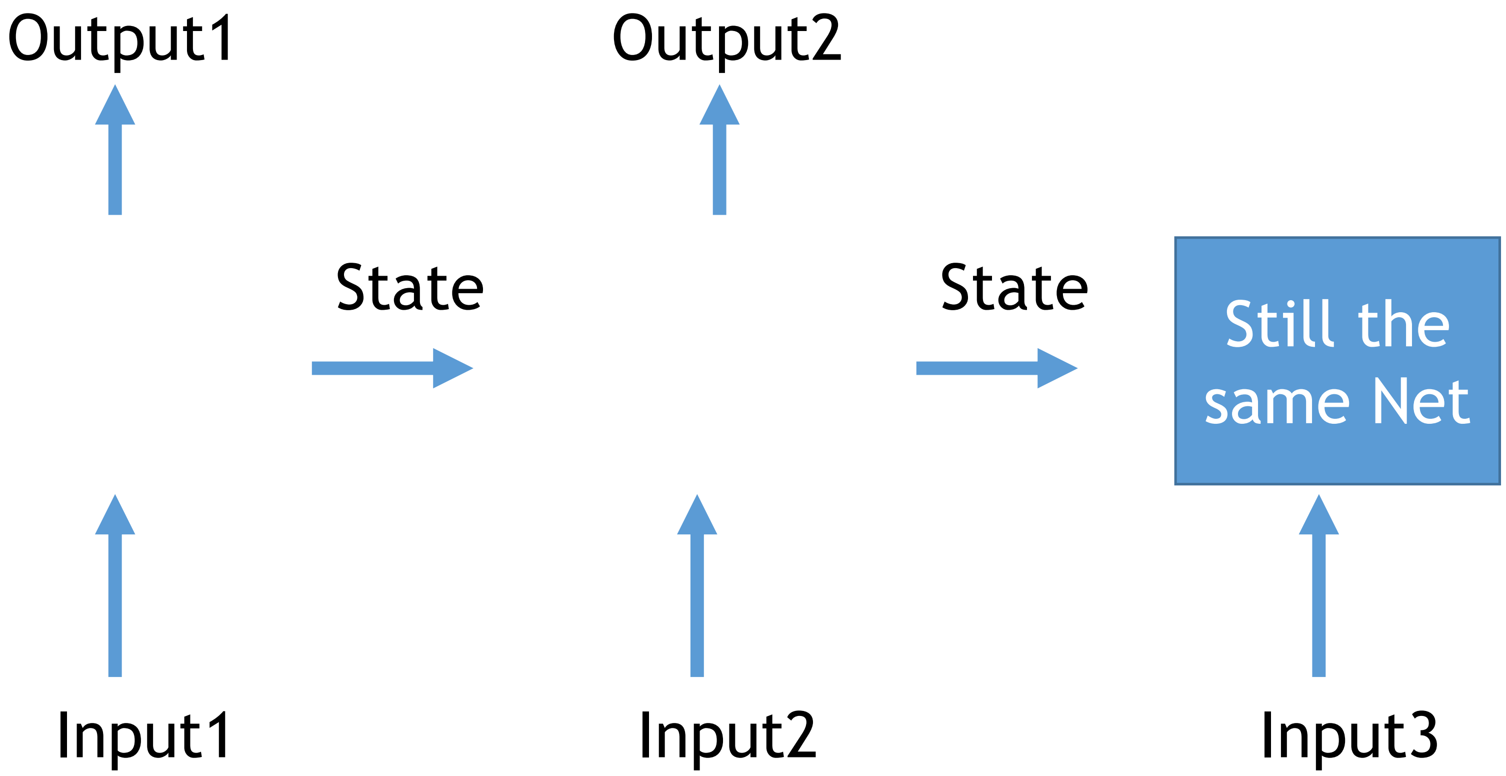
Recurrent Neural Networks



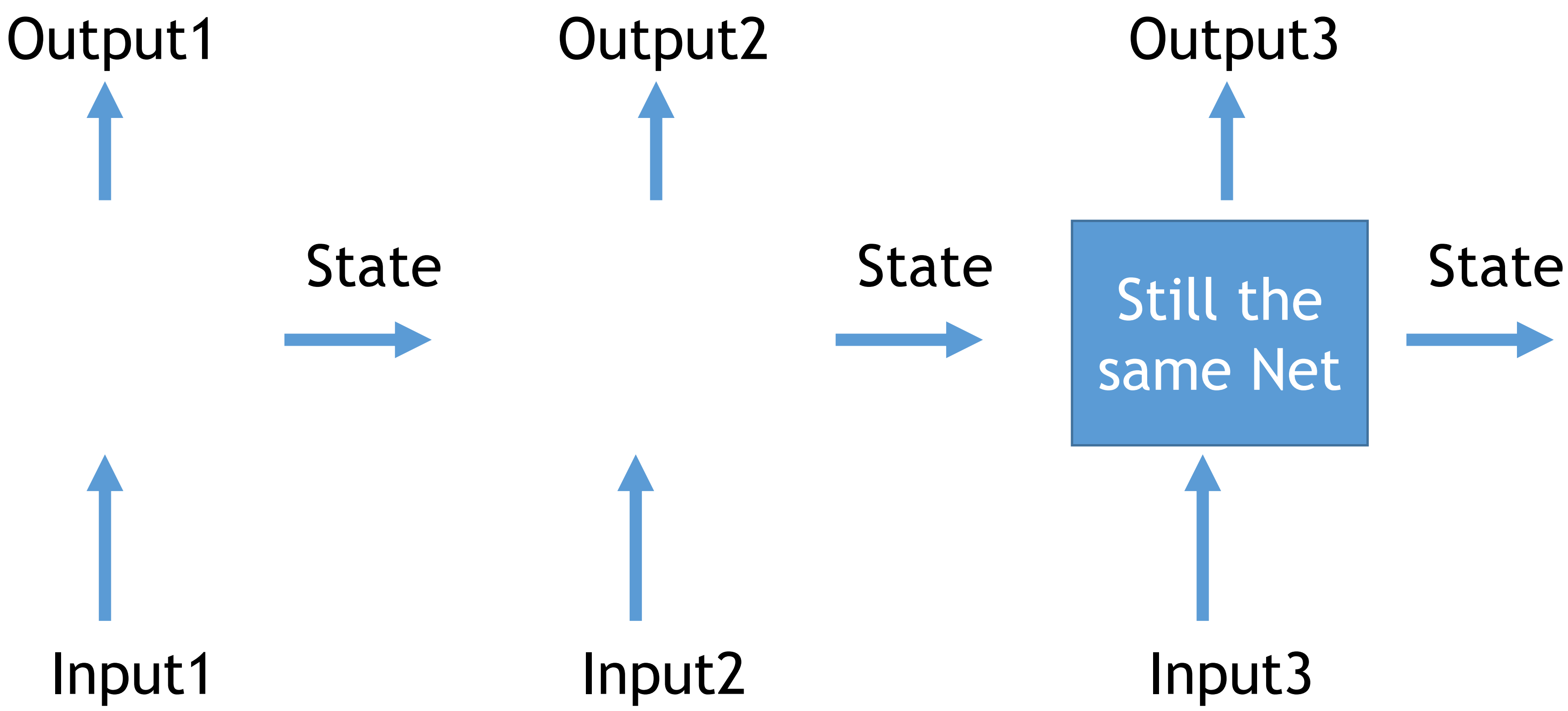
Recurrent Neural Networks



Recurrent Neural Networks



Recurrent Neural Networks

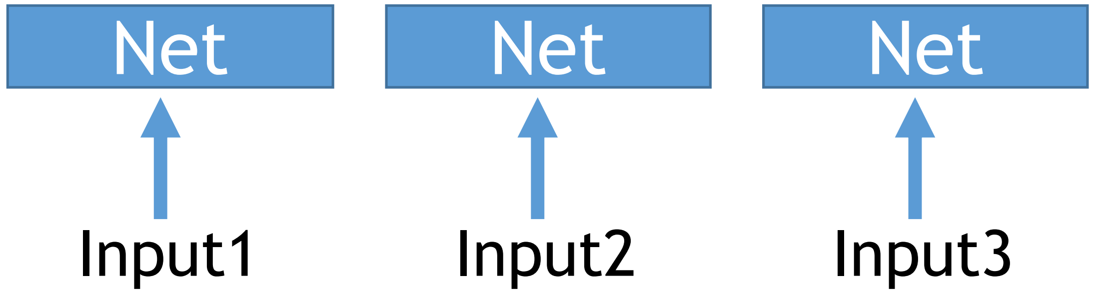


Recursive Neural Networks

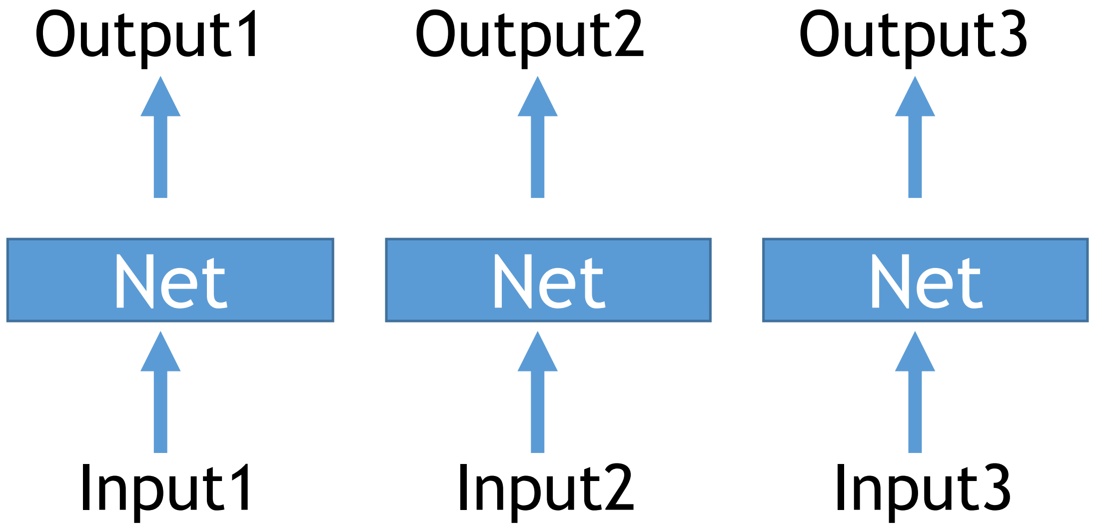
Then it turned out that problems with more hierarchical structure need a different, more generic method:

Recursive neural networks, where the sequential evaluations are not a simple linear series, but form some sort of tree!

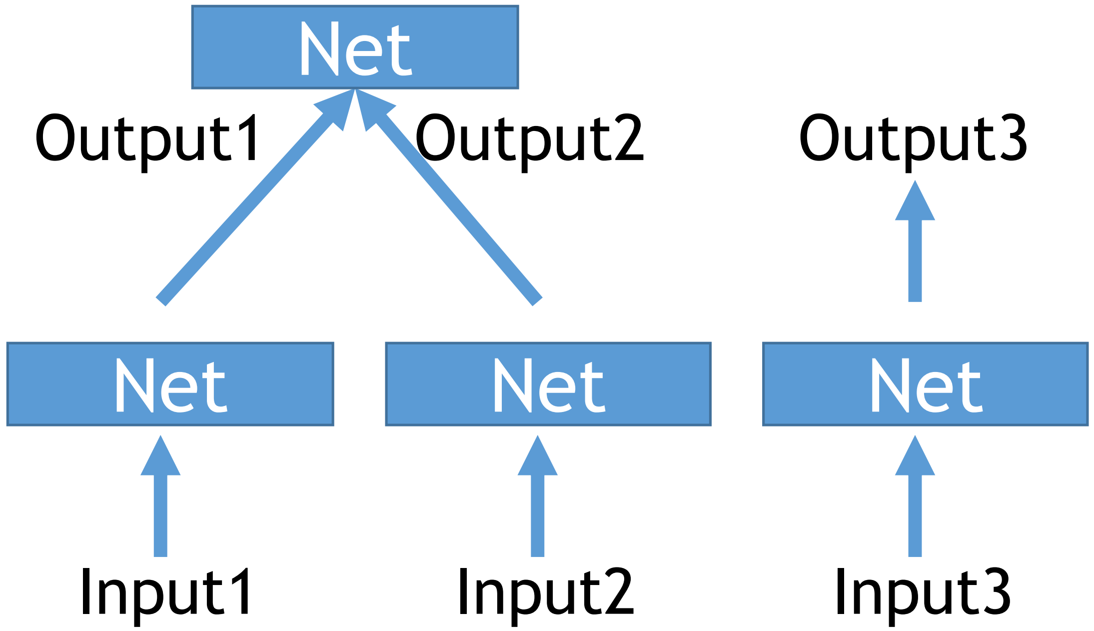
Recursive Neural Networks



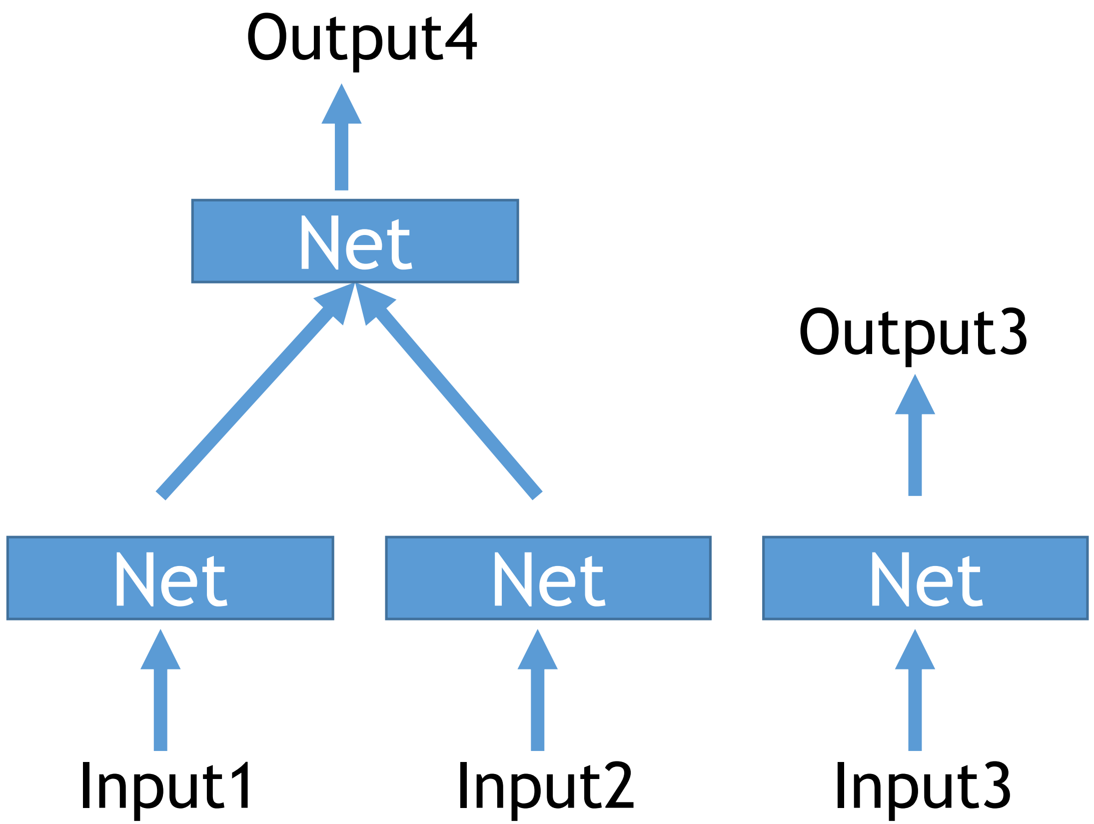
Recursive Neural Networks



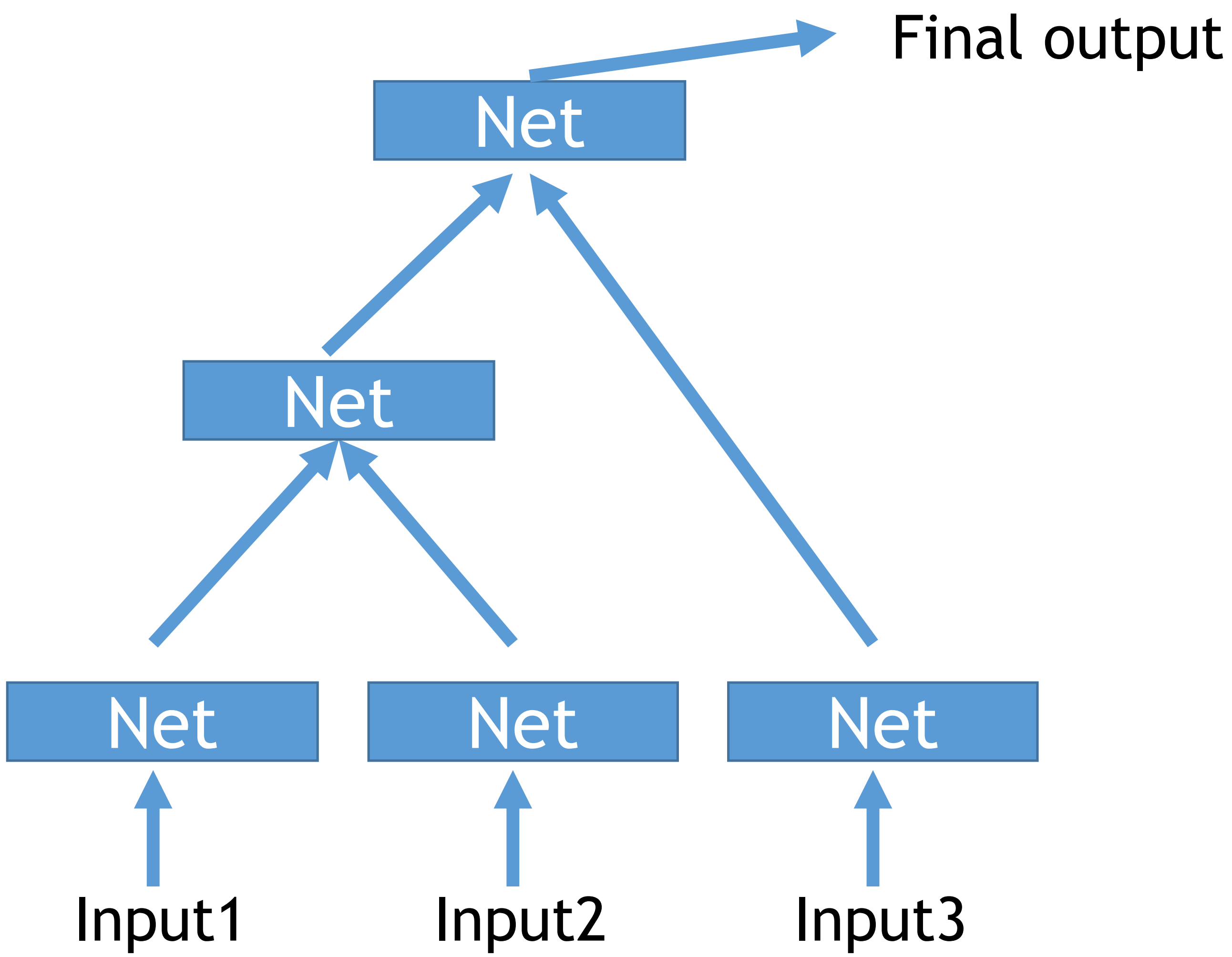
Recursive Neural Networks



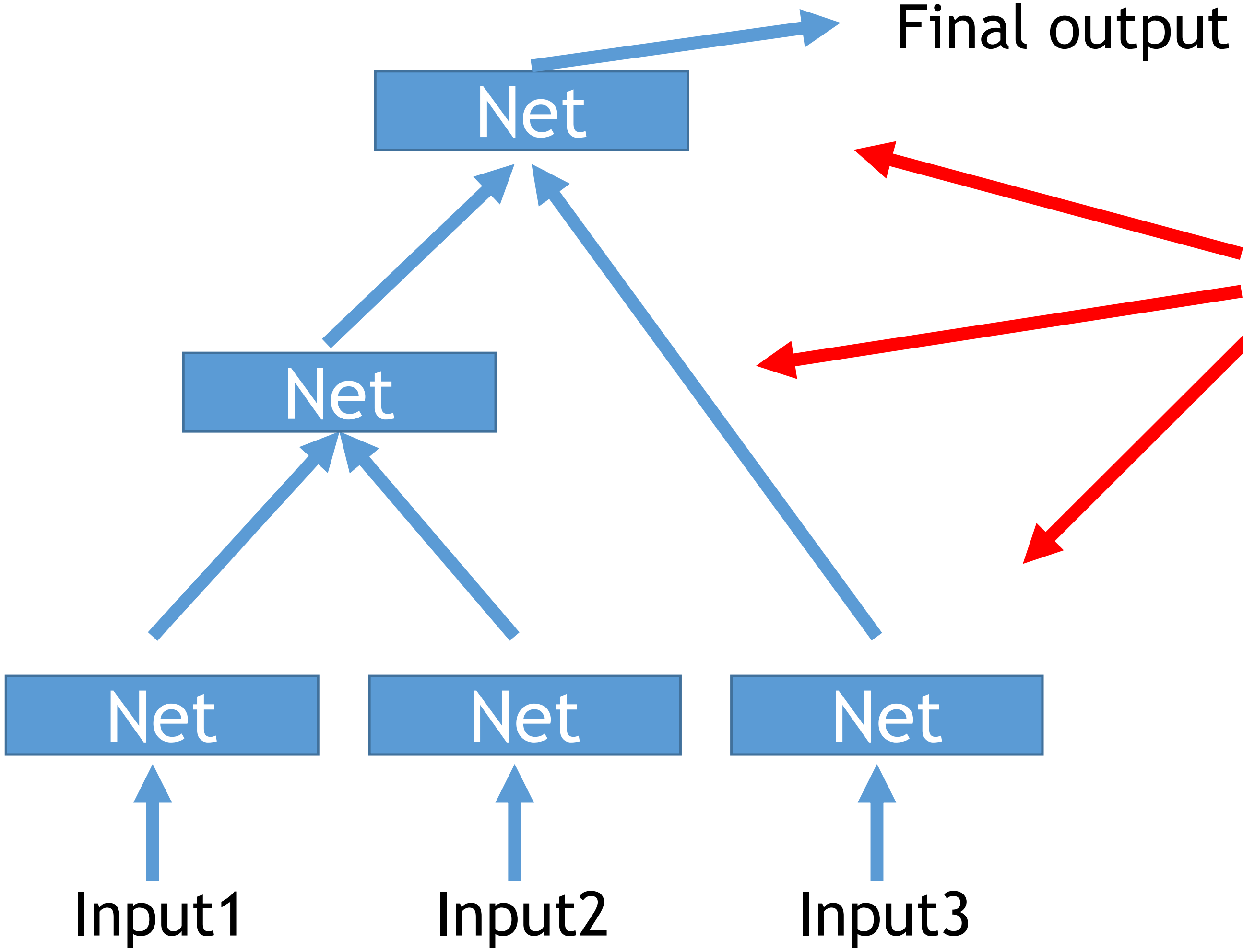
Recursive Neural Networks



Recursive Neural Networks



Recursive Neural Networks



The same net with the same weights operate at each level at each branch!

Recursive Neural Networks

Example:

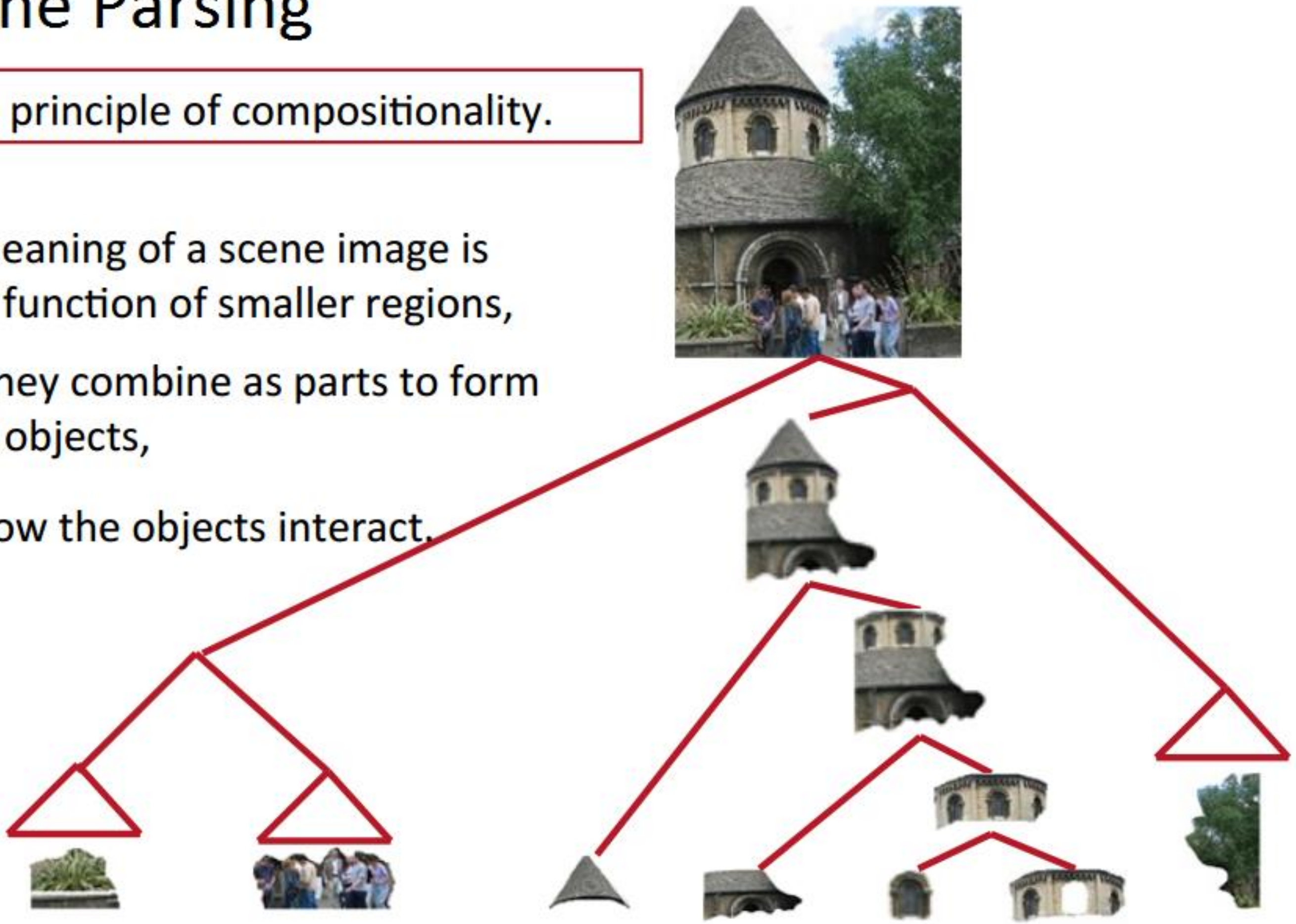
Scene Parsing

Similar principle of compositionality.

The meaning of a scene image is also a function of smaller regions,

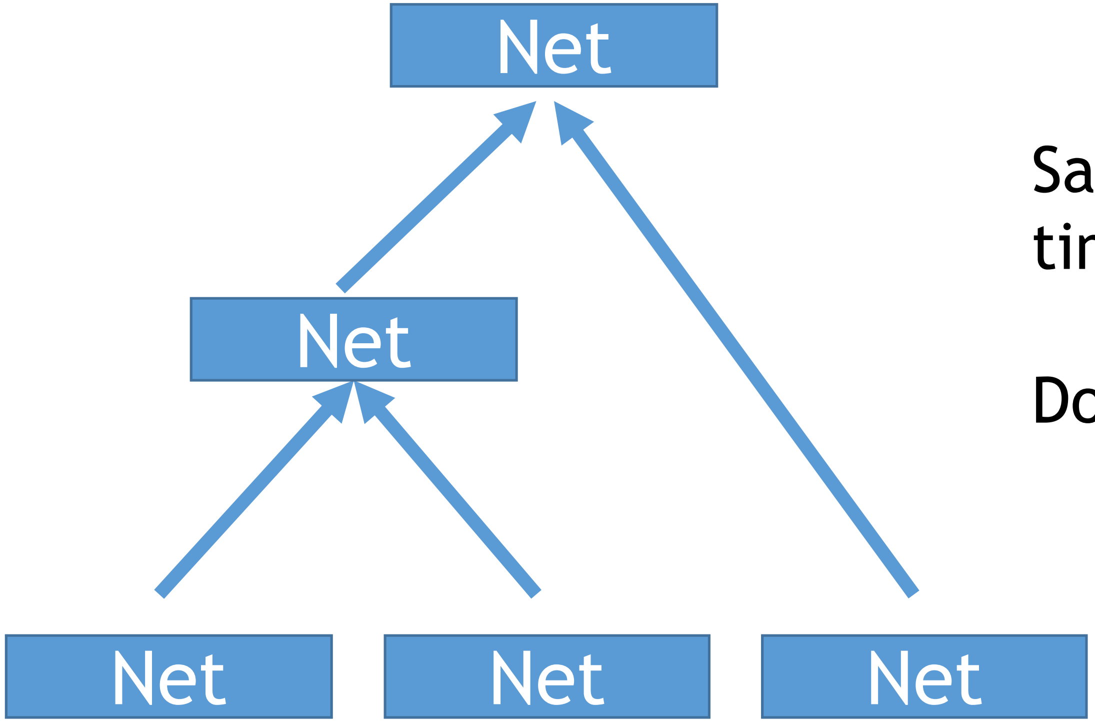
how they combine as parts to form larger objects,

and how the objects interact.



Socher, Richard & Chiung-Yu Lin, Cliff & Y. Ng, Andrew & Manning, Christoper. (2011). Parsing Natural Scenes and Natural Language with Recursive Neural Networks. Proceedings of the 28th International Conference on Machine Learning, ICML 2011. 129-136.

Recursive Neural Networks



Same operation applied every time in a tree?

Does this sound familiar?

Neural Networks and Functional Programming

Christopher Olah pointed out on his blog that different neural network types closely correspond to functional programming primitives.

In particular:

- Recurrent Neural Networks are folds / unfolds
- Recursive Neural Nets are cata / anamorphisms

Recursion Schemes

- Quite a lot of recursion schemes were described by the theoretical literature.
- Many of them are known to be useful and good to know about
- Recognizing them in different settings can make it easier to reason about and solve complex problems

Literature

Wigner GPU Lab

Sample codes: [github.com/u235axe/HaskellHacks]

Erik Meijer, J. Hughes, M.M. Fokkinga, Ross Paterson
Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

Ralf Hinze, Nicolas Wu, Jeremy Gibbons
Unifying Structured Recursion Schemes

Tim Willams' talk

Edward Kmett's blog posts

Patrick Thomson's blog posts

Bartosz Milewski's blog posts

Recursive Neural Networks

Pollack, J. B. Recursive distributed representations. Artificial Intelligence Vol 46 (1990)

Bottou, L. arXiv.1102.1808. 401 (2011)

Frasconi, P., et. al. A general framework for adaptive processing of data structures. IEEE Transactions on Neural Networks , (1998).

Recursion Schemes – in real life

Wigner GPU Lab

Next:

- András will show how to make a parallelizing compiler and GPU code generator by composing cata-, ana- and paramorphism

LambdaGen

LambdaGen automates parts of the
common scientific development pipeline:

- Symbolic math expressions
- Linear algebraic expressions
- High-level programming terms
- Low-level programming terms

$$\partial_{[\alpha}F_{\beta\gamma]} = 0 \quad \partial_{\alpha}F^{\alpha\beta} = \mu_0 J^{\beta}$$

How to get
there?

Goal: minimize dev time + run time

LambdaGen

LambaGen consists of

- A Haskell EDSL to define the computation
  - Functional primitives (map, reduce, zip, lambda calculus)
- A set of analysis and transformation steps
  - The primitives form an expression *tree*
    → all the previous theory applies
- GPU code generation (SYCL / ComputeCpp)
- Automatic memory management (C++)

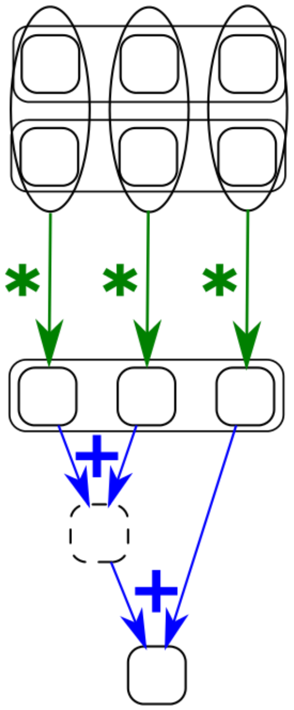
LambdaGen — Primitives

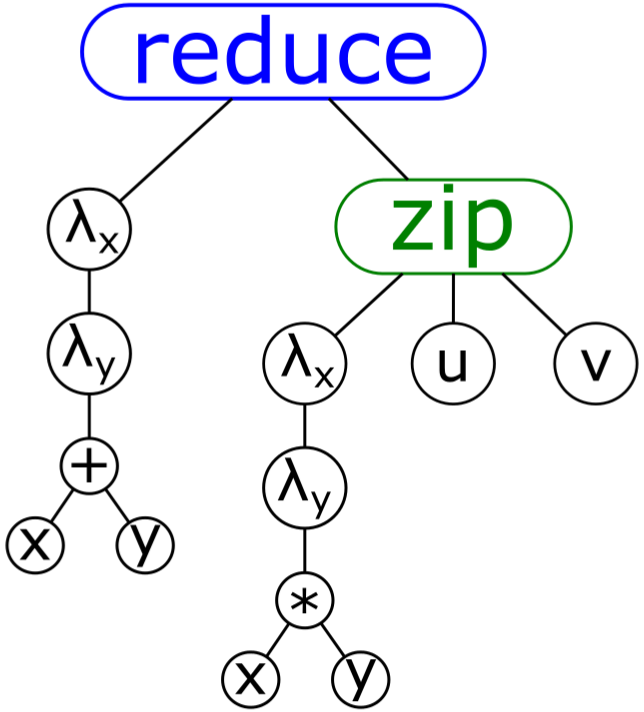
```haskell
data ExprF a =
  Scalar          { getValue :: Double }
| Addition        { left :: a, right :: a }
| Multiplication{ left :: a, right :: a }
| VectorView { id :: String, dms :: [Int], strd :: [Int] }
| Apply       { lambda :: a, value :: a}
| Lambda      { varID :: String, varType :: Type, body :: a }
| Variable    { id :: String, tp :: Type }
| Map         { lambda :: a, vector :: a }
| Reduce      { lambda :: a, vector :: a }
| ZipWith     { lambda :: a, vector1 :: a, vector2 :: a }
deriving (Functor, Show)
```

LambdaGen — Example

$$\sum_i u_i\, v_i$$

```
reduce
    (lam x (lam y
        (add x y)))
    (zip
        (lam x (lam y
            (mul x y)))
        u
        v)
```





Control.Comonad.Cofree

We can put annotations on the expression tree nodes by using

```
data Cofree f a = a :< f (Cofree f a)
             -- = a :< f (a :< f (Cofree f a))
```

Instead of

```
newtype Fix f = Fix (f (Fix f))
           -- = Fix (f (Fix (f (Fix f))))
```

Data.Functor.Foldable

- Fix, Cofree and the others are generalized into:

```
type family Base t :: * -> *
```

- Which for Fix means:

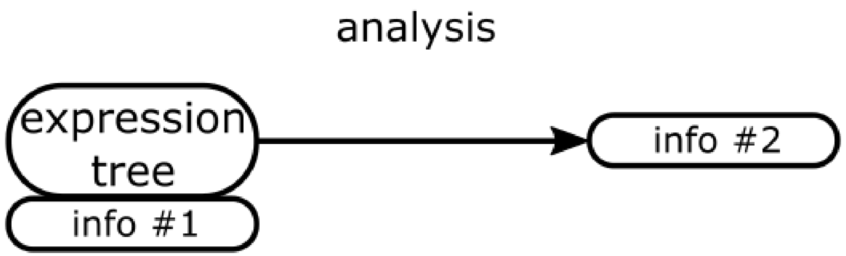
```
type instance Base (Fix f) = f
```

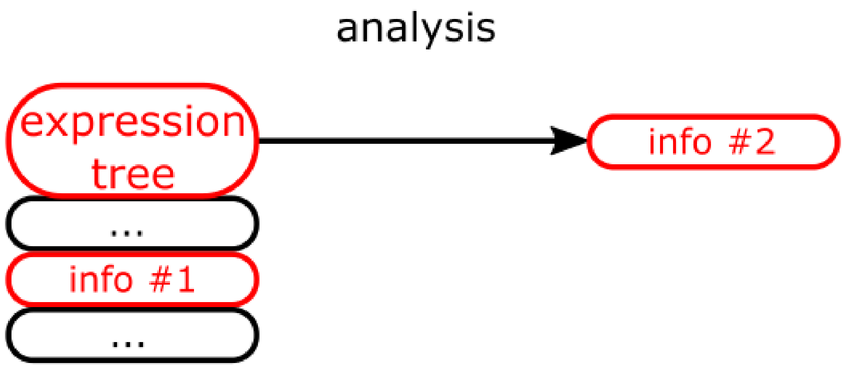
- And so the class of recursive types become:

```
class Functor (Base t) => Recursive t where
    project :: t -> Base t t -- unFix
    cata :: (Base t a -> a) -> t -> a
    cata alg = alg . fmap (cata alg) . project
```

… and countless other schemes …

LambdaGen – Annotations





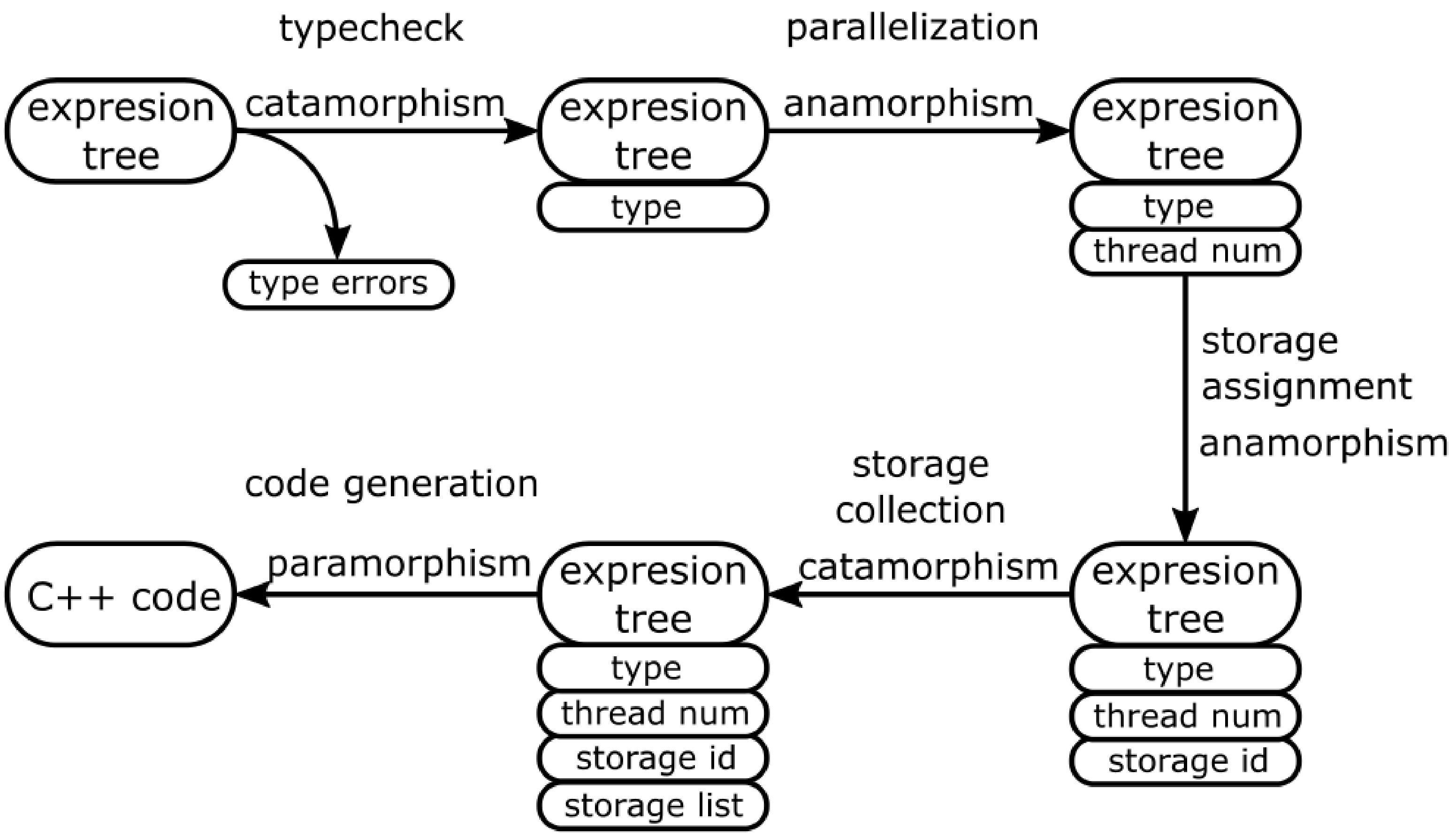
- The set of annotations changes while processing the expression tree
- A transformation must not depend on the exact set, but may require some annotations to be present

Data.Vinyl

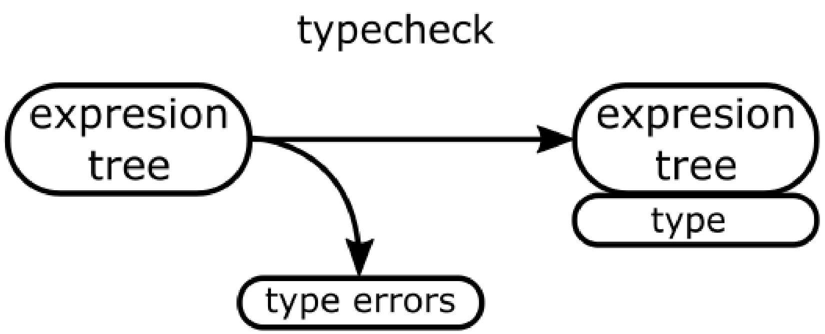
All this and much more is supported by the Vinyl extensible record package:

- The field set is a type level list → extension is `':`
- Presence of fields can be constrained
  - `info1 ∈ fields => …`
- Or even subset relationships
  - `subs ⊆ fields => …`

LambdaGen – Overview



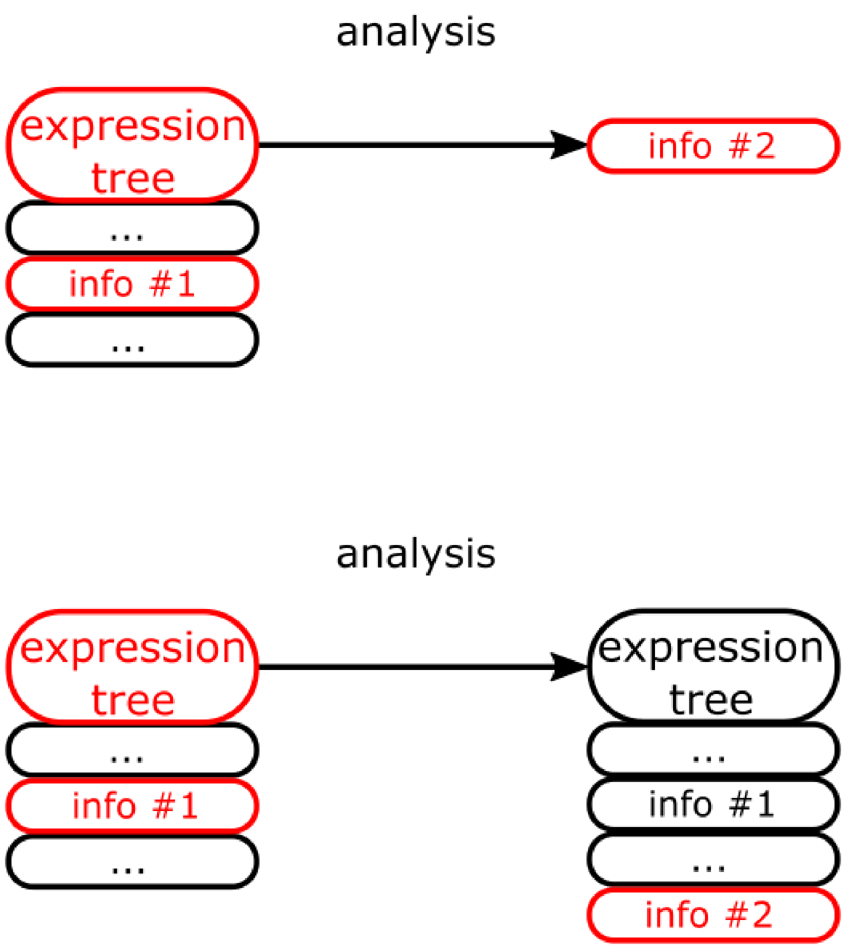
LambdaGen – Typecheck




```haskell
type Algebra t a = Base t a -> a
type Error = String
type TypecheckT = Either Type [Error]

typecheckAlg :: Algebra (Cofree ExprF (R fields)) TypecheckT
```

LambdaGen – Annotations



The previous algebra would reduce the whole tree into a single type

But we'd like to make a chain of transformations

We should annotate the nodes with the partial results

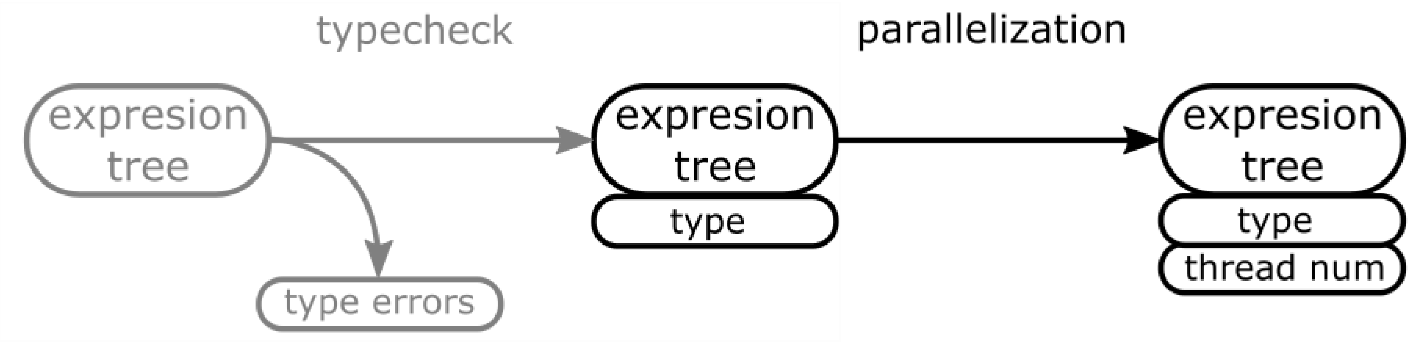
LambdaGen – Algebra adapter

Wigner GPU Lab

The `annotate` function takes a „reducing” algebra and creates a transforming or „annotating” algebra, that keeps the tree and adds some more info:

```
annotate :: Functor f =>
            Algebra (Cofree f (R old)) new ->
            Algebra (Cofree f (R old))
                    (Cofree f (R (new ': old)))
```

LambdaGen – Parallelization




```
type CoAlgebra t a = a -> Base t a
data ParState = None | Started Int | Full Int
type ParData = (Int, Maybe Int)

parallelizerAlg :: TypecheckT ∈ fields => Int ->
             CoAlgebra (Cofree ExprF ParData)
                       (Cofree ExprF (R fields), ParState)
```

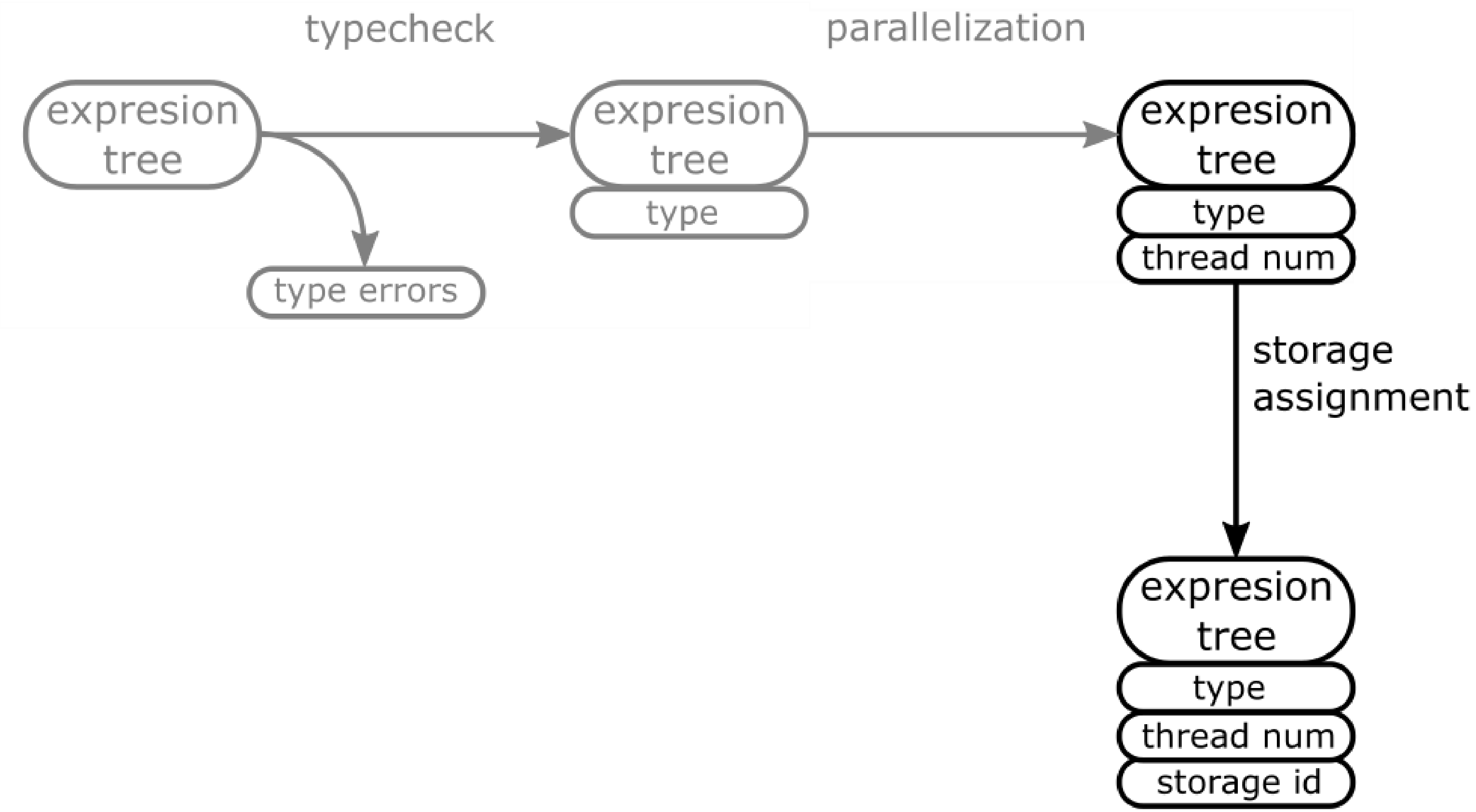
LambdaGen – CoAlgebra adapter

Just like before

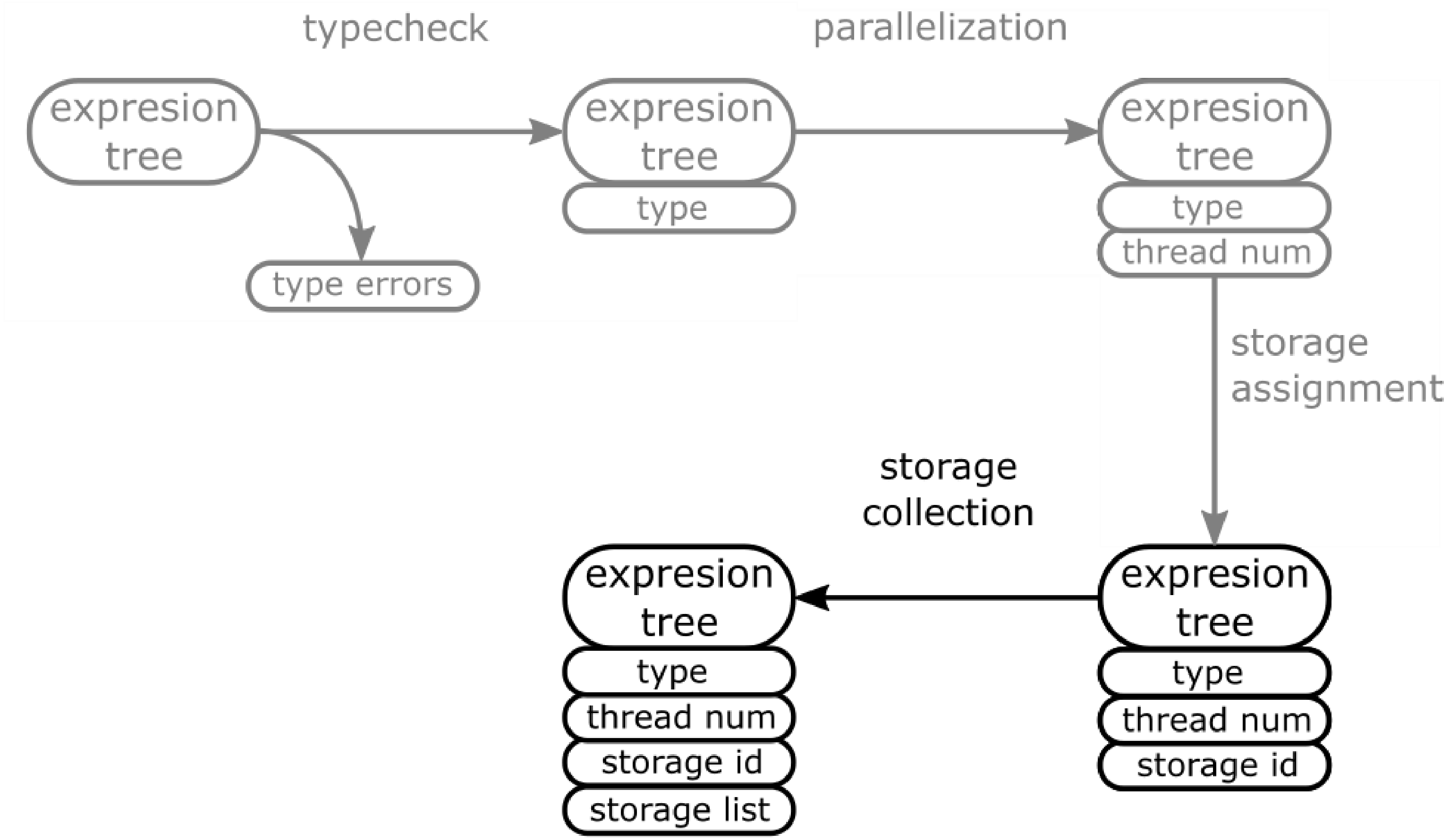
- we produce the new annotation
- the adapter appends it to the existing ones

```
annotateAna :: Functor f =>
               CoAlgebra (Cofree f new)
                         (Cofree f (R old), seed) ->
               CoAlgebra (Cofree f (R (new ': old)))
                         (Cofree f (R old), seed)
```

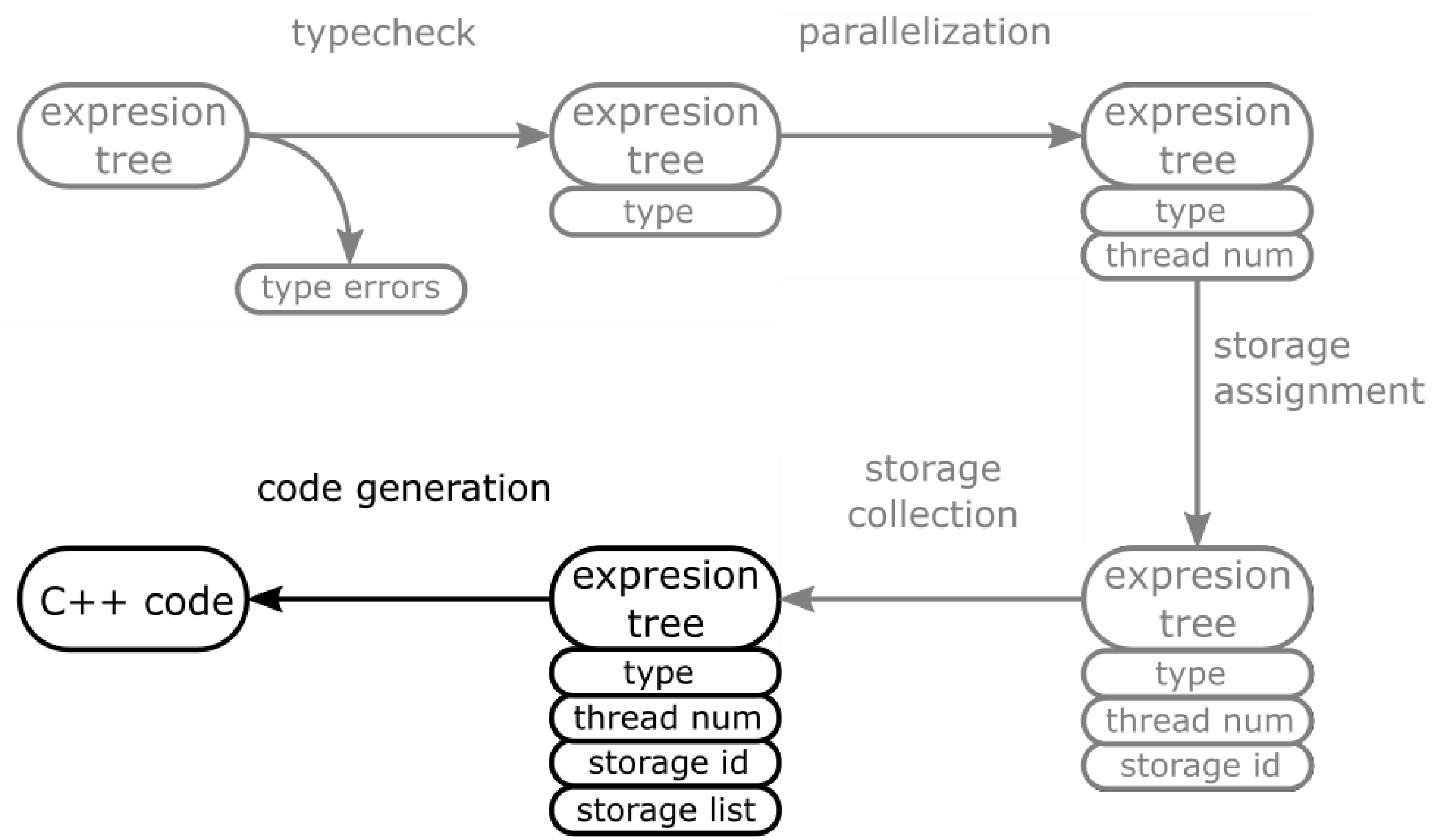
LambdaGen — Storage



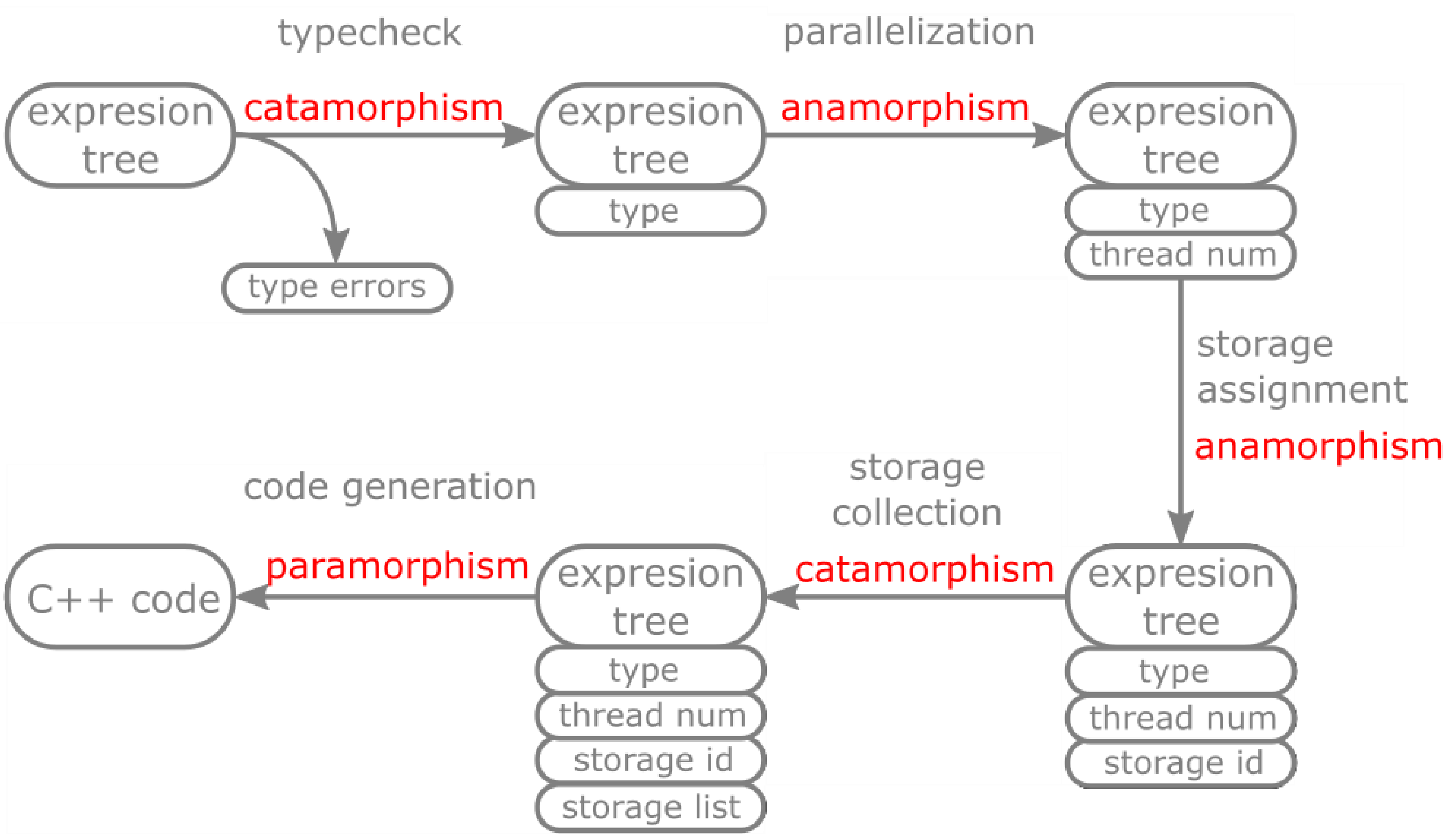
LambdaGen — Storage



LambdaGen – Code generation



LambdaGen – Recursion schemes



LambdaGen — Status

Tested on simple, but multidimensional linear algebra problems

(vector, matrix, tensor) (sums, products)

Works on CPU with `std::thread` based parallelism

Works on GPU, thanks to ComputeCpp’s C++11 support

Code available on github:

github.com/leanil/LambdaGen

LambdaGen — Future

Further work:

- High-level optimization framework
  - Based on fusion/fission rules from theory
  - We'll need to pattern match and replace the expression tree
  - Can this be done with recursion schemes?
- Hierarchical parallelism
  - Local memory usage and vectorisation with HOF splitting and exchanging rules

Summary

Research at the Wigner GPU Lab aims to reduce the burden on scientists via bridging high level mathematical models down to hardware optimized codes with modern programming approaches

LambdaGen with its solely recursion schemes based extensible compiler logic is an important step in this direction

Follow or join the developments:

gpu.wigner.mta.hu

github.com/Wigner-GPU-Lab

github.com/leanil/LambdaGen