Applications of structured recursion schemes

Dániel Berényi
Wigner Research Centre for Physics, GPU Lab

In collaboration with

András Leitereg and **Gábor Lehel**

Eötvös University

Logic, Relativity and Beyond '17
August 23-27. Budapest

Introduction

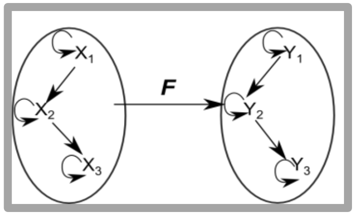
Modern programming tries to tackle more and more complex problems and to succeed it relies on results and tools from

- Functional Programming
- Type Theory
- Algebra
- Logic
- Category Theory

Monoid:
$\Sigma = 1 + A \times A$

$p \lor \neg p \Leftrightarrow T$

		Terms	
t	::=	x	Variable
	|	$t\ t$	Function application
	|	$\lambda x : \tau . t$	Lambda abstraction
		Types	
τ	::=	T	Primitive type
	|	$\tau \rightarrow \tau$	Function



Introduction

What do we use from them?
(Non exhaustive collection!)

- Functional Programming
- Type Theory
- Algebra
- Logic
- Category Theory

Compositional, abstract (rel. to hardware) primitives

Safety, improved reasoning, soundness

Type compositions

Reasoning, inferring, proving

Proving abstract relations in TT and FP

Introduction

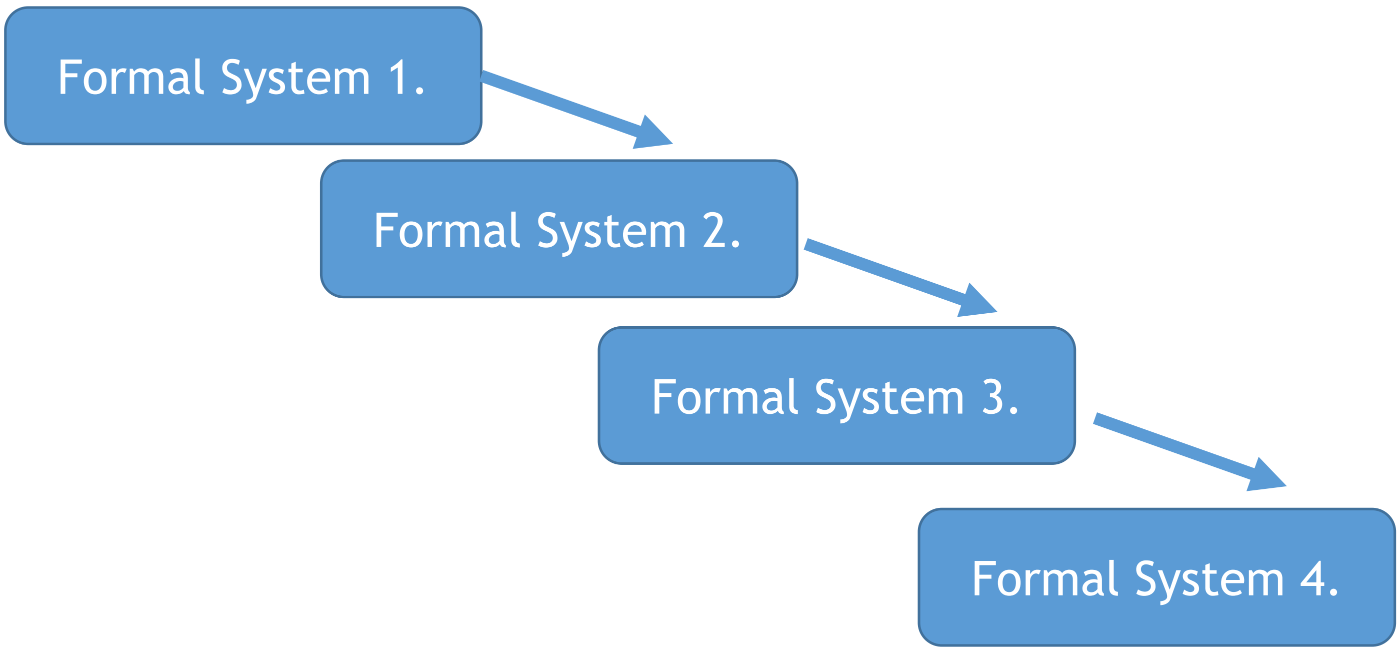
When solving a problem in a certain field with the aid of a computer and programming languages,

we inevitably face a transition from one formal system to another:

Formal System 1. → Formal System 2.

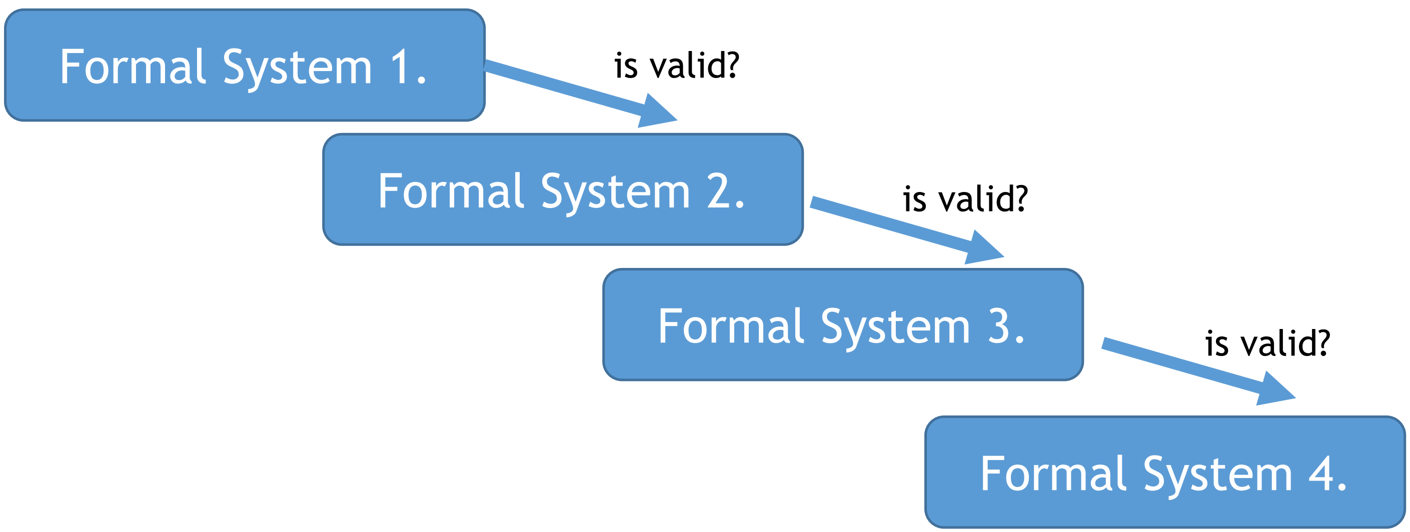
Introduction

Even worse, usually we end up with a series of transitions:



Introduction

During the transition many operations should be carried out on the expressions on the formal systems and we need tools that are easy to reason about...



Formal Systems

Formal systems consist of:

- Symbols
- Grammar
- Axioms
- Inference rules

Formal Systems

Formal systems consist of:

- Symbols
- Grammar
- Axioms
- Inference rules

Tells how to build well-formed expressions from the symbols

Formal Systems

Example system: addition of integers

Valid expressions:

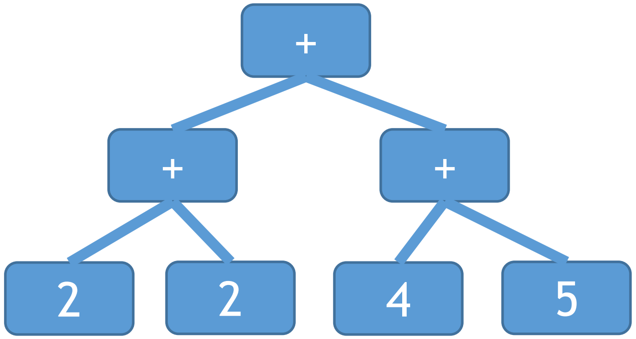
- 1
- 1+1
- (1+2) + 4
- (2+2) + (4+5)

Formal Systems

Example system: addition of integers

Valid expressions:

- 1
- 1+1
- (1+2) + 4
- (2+2) + (4+5)



Expression Trees

When transforming this into a programming language, we need to assign a type to the syntax of the expressions...

- `1` `Integer`
- `1+1` `Addition Integer Integer`
- `(1+2) + 4` `Addition (Addition Integer Integer) Integer`

But this seems unnatural, as we would like to have a common type for all expressions...

Expression Trees

Let’s capture the recursive nature of expression trees into a single type:

„an Expression is EITHER (a Constant)

or (an Addition of two Expressions)”

Expression Trees

„an Expression is EITHER (a Constant)
or (an Addition of two Expressions)"

This is exactly represented by sum types:

```
type Expression = Constant Integer
                | Addition Expression Expression
```

Expression Trees

Let's factor out recursion:

Consider the standard recursive definition of the factorial function:

$$\text{factorial: } (n) \to \begin{cases} 1, & n = 0 \\ n \cdot \text{factorial}(n-1), & n \neq 0 \end{cases}$$

Expression Trees

The recursion can be abstracted out in the form of fixed points.

Given $\mathrm{fix}(f) = f(\,\mathrm{fix}(f))$, where f is a function taking a function (itself under the image of fix) as first argument:

$$\text{factorial\_prototype: } (f, n) \to \begin{cases} 1, & n = 0 \\ n \cdot f(f, n-1), & n \neq 0 \end{cases}$$

and then:

$$\text{factorial}(x) \;=\; \text{fix}(\,\text{factorial\_prototype}\,)(x)$$

Expression Trees

Similarly we can create a parametric type:

```
forall t ∈ Types
type Expression_proto t =
                        Constant Integer | Addition t t

type Expression = Fix Expression_Proto
```

With the following helper functions:

```
  fix : F( Fix F ) -> Fix F        Hide one level of the tree
unfix : Fix F      -> F( Fix F )   Reveal one level of the tree
```

Expression Trees

One of the common operations on expressions is reducing them according to certain rules.

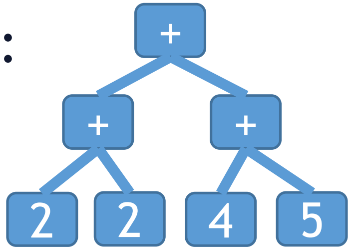
- How does an evaluator look like for our grammar?
- We would like to have something like this if the sub exprs are already evaluated:

If `e` is an `Expression_proto Integer`, then

```
case Constant:  Integer -> Integer
case Addition: (Integer, Integer) -> Integer
```

So together the signature of this evaluator function is:

```
Expression_proto Integer -> Integer
```

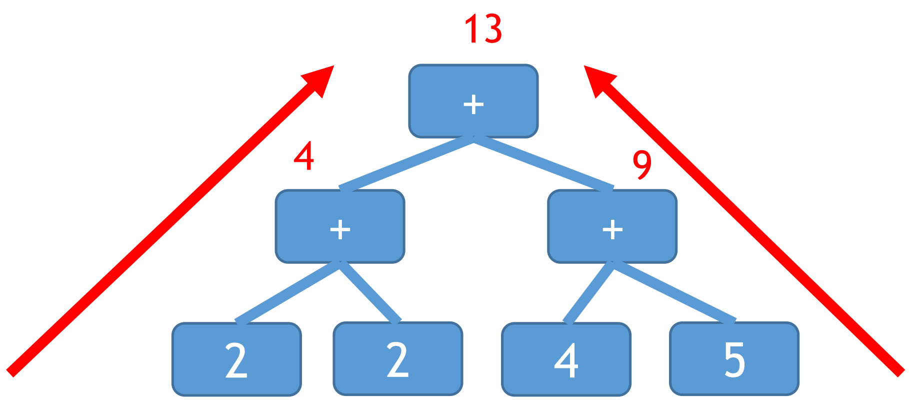


Expression Trees

WIGNER GPU Lab

But...

We have a recursive tree, we need to apply our evaluator bottom-up and be well-typed at every level...



Structured Recursions

The solution is called the *catamorphism*, and was constructed in functional programming and it's properties were proven in category theory:

$$\text{cata} : (F\ a \to a) \to \text{Fix}\ F \to a$$

$$\text{cata}\ \alpha = \alpha \circ F(\text{cata}\ \alpha) \circ \text{unfix}$$

$$\begin{tikzcd}
F(\,\text{Fix}(F)\,) \arrow[r, "F(\Phi)"] \arrow[d, "\text{fix}"'] & F(a) \arrow[d, "\alpha"] \\
\text{Fix}(F) \arrow[r, "\Phi"'] & a
\end{tikzcd}$$

$$\phi = \text{cata}\ \alpha$$

Catamorphism

$$\text{cata} : (\text{F a} \rightarrow \text{a}) \rightarrow \text{Fix F} \rightarrow \text{a}$$
$$\text{cata}\ \alpha = \alpha \circ \text{F}(\text{cata}\ \alpha) \circ \text{unfix}$$

The *catamorphism* takes an evaluator $(\alpha\!: \text{F a} \rightarrow \text{a})$ that produces a type 'a' from an expression type F holding evaluated subresults.

The evaluator is called an *algebra* and the type a is called the *carrier type* of the algebra.

The parametric expression type F should be a Functor in the category of types.

Catamorphism

$$\text{cata} : (F\,a \to a) \to \text{Fix}\,F \to a$$
$$\text{cata}\,\alpha = \alpha \circ F(\text{cata}\,\alpha) \circ \text{unfix}$$

The *catamorphism* first unwraps the fixed point type, revealing one step below:

```
Fix( F )     ->     F ( Fix( F ) )
```

Fix (Expression_proto) -> Addition

Fix (Expression_proto)

Fix (Expression_proto)

Catamorphism

$$\mathrm{cata} : (\mathrm{F\,a} \rightarrow \mathrm{a}) \rightarrow \mathrm{Fix\,F} \rightarrow \mathrm{a}$$
$$\mathrm{cata}\,\alpha = \alpha \circ \mathrm{F}(\mathrm{cata}\,\alpha) \circ \mathrm{unfix}$$

Then it applies itself recursively to evaluate subexpressions down until it reaches a terminal leaf (in our case an `Constant`)

`F ( Fix( F ) )` `->` `F a`

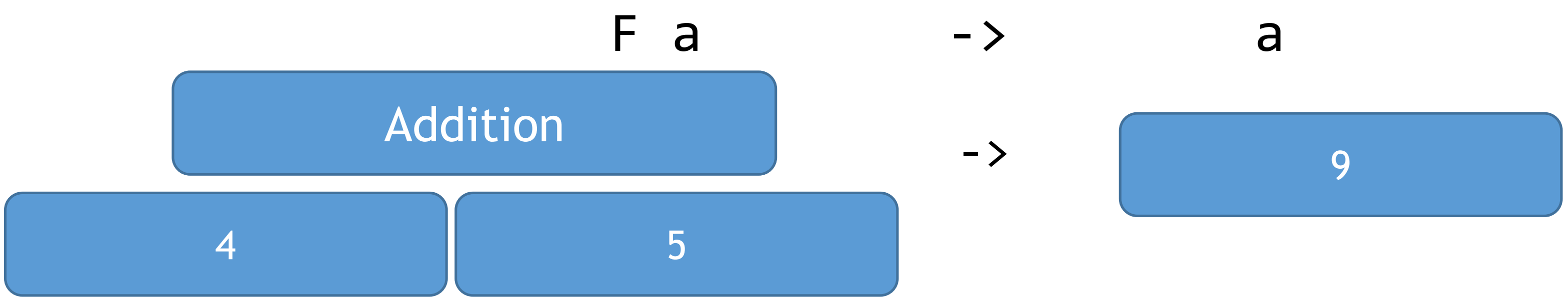
Addition -> Addition

Fix (Expression_proto) | Fix (Expression_proto) | 4 | 5

Catamorphism

$$\mathrm{cata} : (F\,a \to a) \to \mathrm{Fix}\,F \to a$$
$$\mathrm{cata}\,\alpha = \alpha \circ F(\mathrm{cata}\,\alpha) \circ \mathrm{unfix}$$

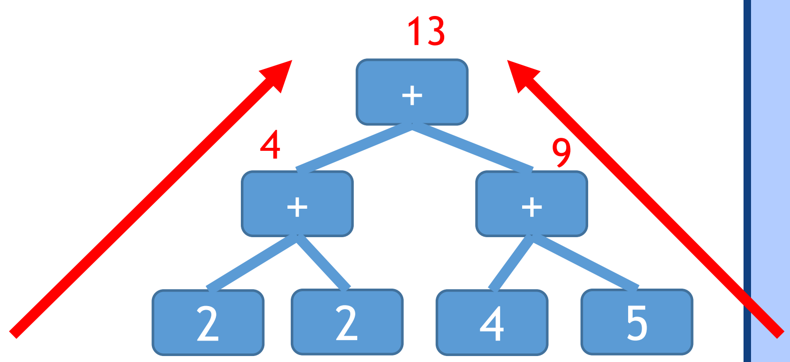
Finally, with the subresults available, it can apply the algebra at the current level:



Catamorphism - example

So in our expression example:

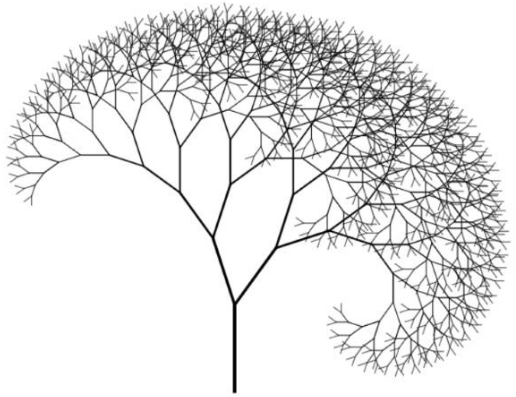
```
type Expression_proto t = Constant Integer
                        | Addition t t
type Expression = Fix Expression_proto

alg : Expression_proto Integer -> Integer
alg x = case (Constant n) => n
        case (Addition left right) => left + right

sum : Expression -> Integer
sum tree = (cata alg) tree
```



Anamorphism

Category theoretical constructs usually come with dual theorems, in this case by reversing the arrows we arrive at the *anamorphism*:

$$\mathrm{ana} : (a \to F\,a) \to a \to \mathrm{Fix}\,F$$
$$\mathrm{ana}\,\overline{\alpha} = \mathrm{fix} \circ F(\mathrm{ana}\,\overline{\alpha}) \circ \overline{\alpha}$$



$$\mathrm{cata} : (F\,a \to a) \to \mathrm{Fix}\,F \to a$$
$$\mathrm{cata}\,\alpha = \alpha \circ F(\mathrm{cata}\,\alpha) \circ \mathrm{unfix}$$

This recursion scheme takes a co-algebra that creates one level of a tree, takes an initial value, and repeats the co-recursion to create a full fixed tree.

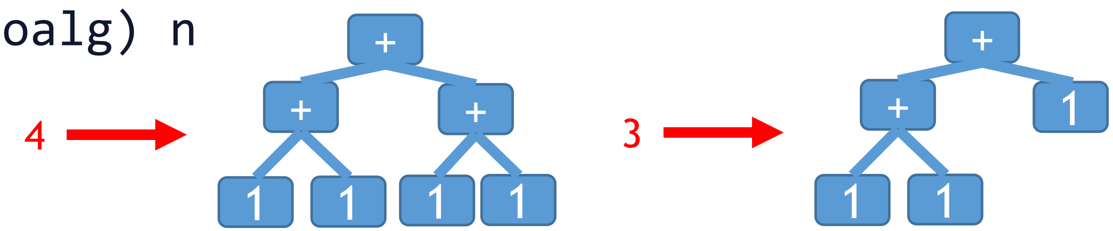
Anamorphism

$$ana : (a \to F\,a) \to a \to Fix\,F$$
$$ana\,\overline{\alpha} = fix \circ F(ana\,\overline{\alpha}) \circ \overline{\alpha}$$

An example of an anamorphism can be generating an expression from a value:

```
coalg n = if( n == 1 ) (Constant 1)
          else
           if( is_even(n) ) (Addition n/2 n/2)
           else             (Addition n-1 1)
```

```
decompose : Integer -> Expression
decompose n = (ana coalg) n
```



Zoo of morphisms

There are many other recursion schemes, in fact there is a hierarchy of more and more general schemes:

Catamorphism – consume tree level by level

Paramorphism – same consumption, but can depend on the structure of the subtrees

Zygomorphism – same consumption with an auxiliary tree traversal

Mutumorphism – consumption with a pair of recursive functions

Why stuctured recursion?

Why are these good for us?

- Factor out recursion from other code
- Makes reasoning simpler
- Makes it possible to algebraically reason about code operating on algebraic structures (products, trees, etc.)
- Expresses intent more clearly
- Combination/fusion identities

What can be done with recursion schemes?

We started with the claim that recursion schemes makes conversion of expressions from one formal system to another simpler.

What can be done with recursion schemes?

One research project at the Wigner GPU Lab is dealing with transforming formulas down to low level GPU code automatically.

Obviously, the two formal systems are quite different, and lots of information need to be analysed and synthetized in the transition.

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

How to get there?

What can be done with recursion schemes?

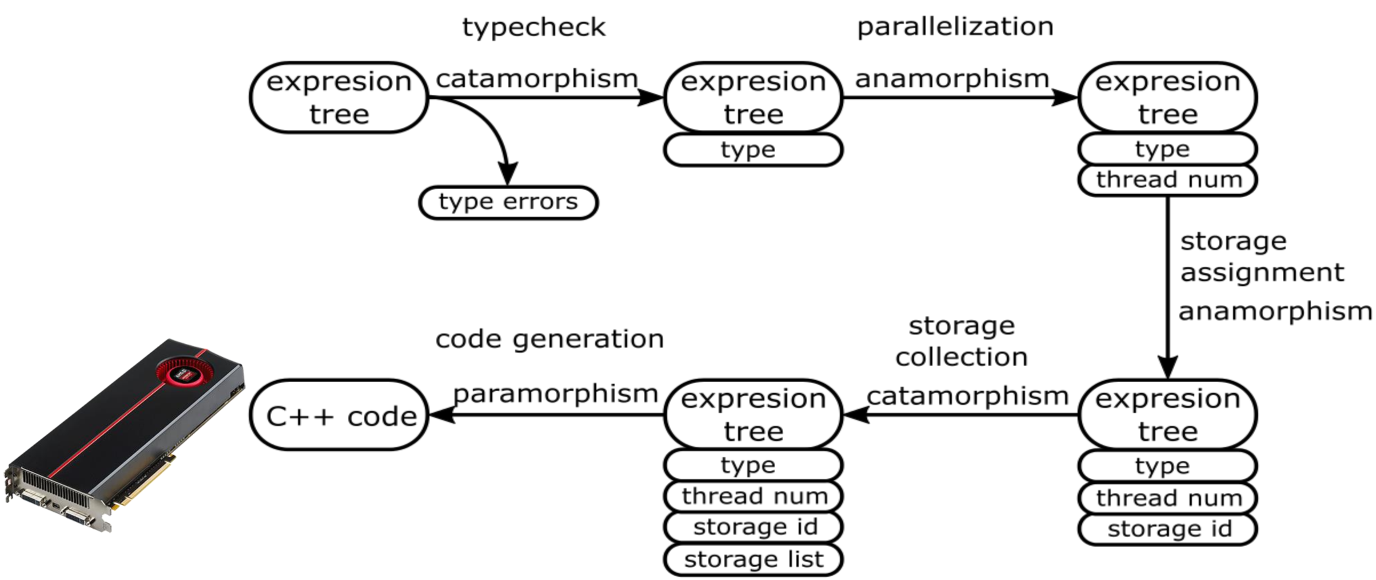
We are developing a library[1] to transform linear algebraic formulas into efficient GPU code

```
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)
```

[1] LambdaGen, see András Leitereg's github page.

What can be done with recursion schemes?

We are developing a library[1] to transform linear algebraic formulas into efficient GPU code



[1] LambdaGen, see András Leitereg's github page.

What can be done with recursion schemes?

By using recursion schemes, it is really hard to create mistakes in the code, as most of them can be caught by the type checker.

Recursion schemes also make the transition modular: we can easily compose yet another traversal onto the pipeline

One more use case for recursion schemes

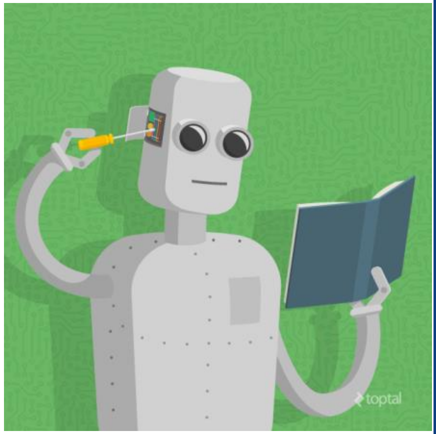
Another seemingly different area where structured recursion started to pop up is...

One more use case for recursion schemes

Another seemingly different area where structured recursion started to pop up is...

... Machine Learning

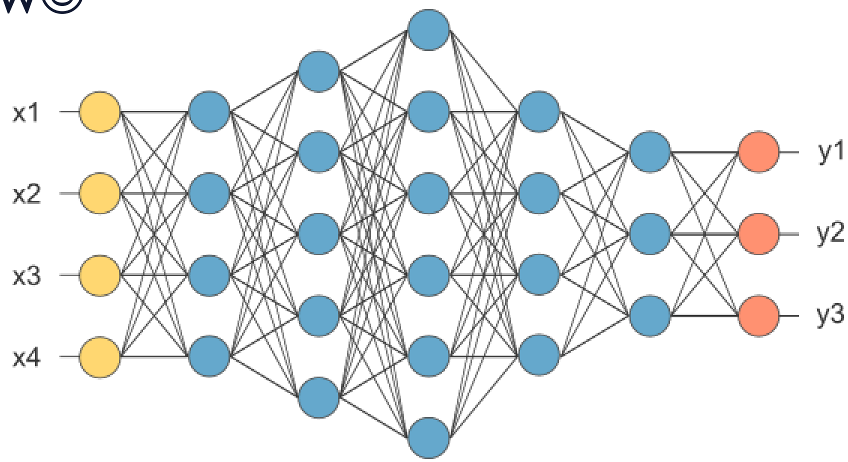
Especially the case of Neural Networks...



One more use case for recursion schemes

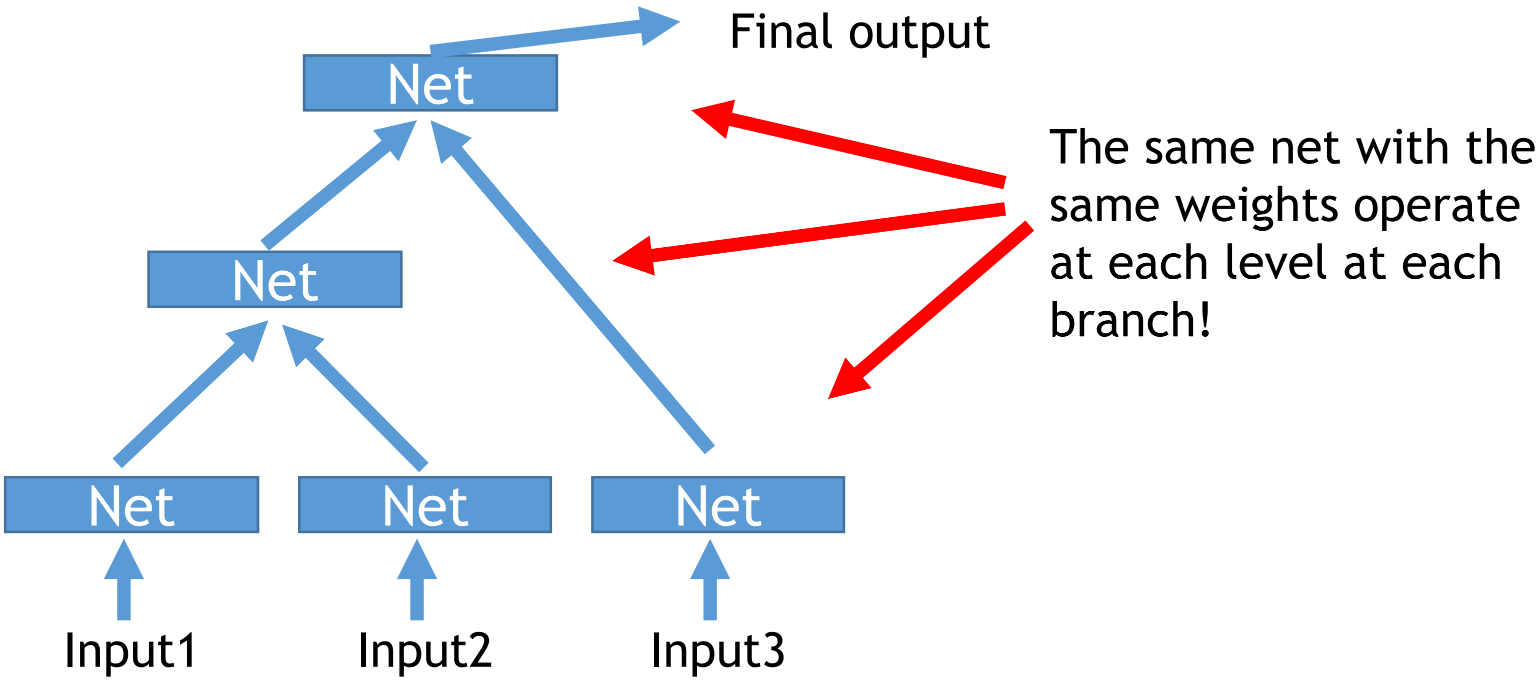
Neural networks are no more than differentiable function compositions optimized with automatic differentiation.

The interesting part is what kind of differentiable functions to compose and how☺



One more use case for recursion schemes

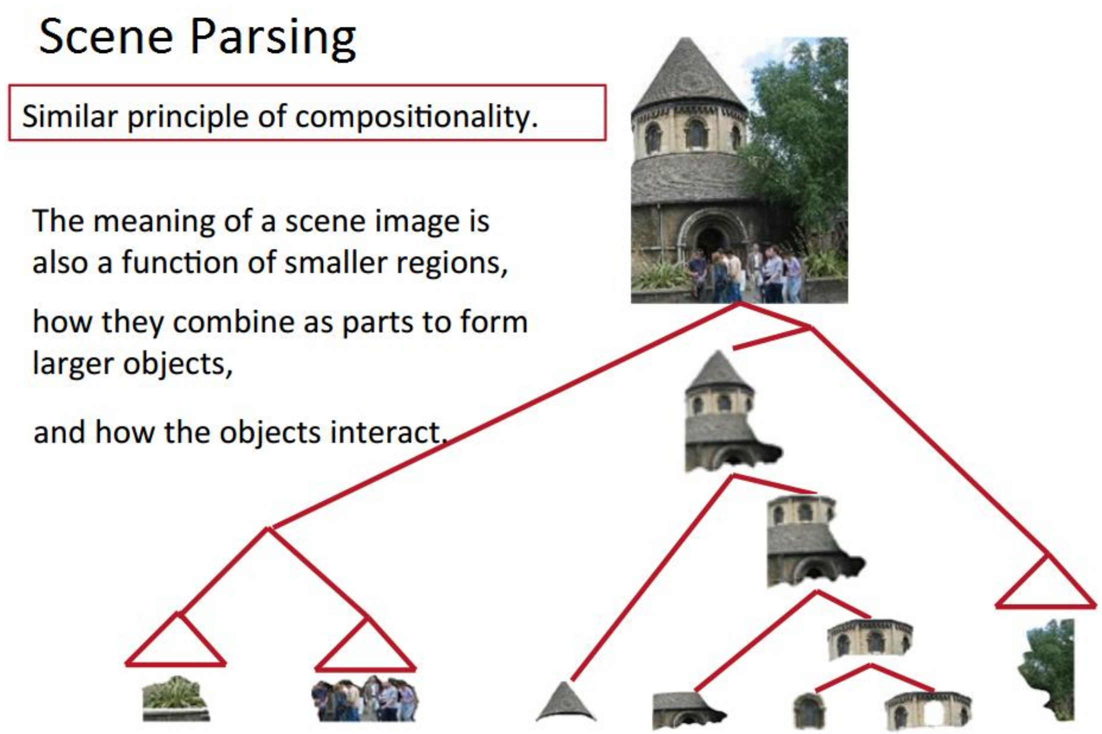
There is a type of Neural Network that looks like the following:



One more use case for recursion schemes

It is called Recursive Neural Network that works just like a catamorphism...

Proved useful in speech-, text processing, scene parsing, etc., where structure is essential



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.

Functional Programming and Neural Networks

In fact, it turns out that neural network layer types correspond to functional programming primitives (see here)

This opens an interesting new field where category theoretical results prove valuable.

Summary

- Structured Recursion Schemes are useful tools for manipulating generic trees and expressing analysis, transformation and evaluation of them.

- They connect algebra, type theory and functional programming, and are backed up by category theoretical identities.

- Hopefully they will soon power the tools of researchers of all kinds☺

References

Thank you!

See the online paper for more references.

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

Edward Kmett's blog posts

Patrick Thomson's blog posts

Bartosz Milewski's blog posts

Tim Willams' talk

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).