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:**

\[\Sigma = 1 + A \times A\]

\[p \lor \neg p \Leftrightarrow T\]

<table>
<thead>
<tr>
<th>Terms</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t ::= x)</td>
<td>(\tau ::= T)</td>
</tr>
<tr>
<td>(t \ t)</td>
<td>(\tau \to \tau)</td>
</tr>
<tr>
<td>(\lambda x : \tau . t)</td>
<td>Function application</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lambda abstraction</th>
<th>Primitive type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda x : \tau . t)</td>
<td>Function</td>
</tr>
</tbody>
</table>

\[F\]

8/24/2017
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

Reasoning, inferring, proving

Type compositions

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:

Introduction

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

Formal System 1.

Formal System 2.

Formal System 3.

Formal System 4.
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 System 1.

Formal System 2.

Formal System 3.

Formal System 4.
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
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**:

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

Let’s factor out recursion:

Consider the standard recursive definition of the factorial function:

\[
\text{factorial: } (n) \rightarrow \begin{cases} 
1, & n = 0 \\
 n \cdot \text{factorial}(n - 1), & n \neq 0
\end{cases}
\]
The recursion can be abstracted out in the form of fixed points.

Given $\text{fix}(f) = f(\text{fix}(f))$, 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:

\[
\text{forall } t \in \text{Types} \quad \text{type } \text{Expression}_\text{proto} t = \text{Constant Integer} \mid \text{Addition } t t
\]

\[
\text{type } \text{Expression} = \text{Fix} \text{Expression}_\text{Proto}
\]

With the following helper functions:

\[
\text{fix} : \text{F}(\text{Fix } F) \to \text{Fix } F \quad \text{Hide one level of the tree}
\]

\[
\text{unfix} : \text{Fix } F \to \text{F}(\text{Fix } F) \quad \text{Reveal one level of the tree}
\]
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:

\[
\text{Expression}_\text{proto Integer} -> \text{Integer}
\]
Expression Trees

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:

\[
cata : (F \ a \to a) \to \text{Fix } F \to a
\]
\[
cata \alpha = \alpha \circ F(\text{cata } \alpha) \circ \text{unfix}
\]
Catamorphism

cata : (F a → a) → Fix F → a
\[ \text{cata } \alpha = \alpha \circ F(\text{cata } \alpha) \circ \text{unfix} \]

The *catamorphism* takes an evaluator (\( \alpha: F a \rightarrow a \)) 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

\[ \text{cata} : (F \alpha \rightarrow \alpha) \rightarrow \text{Fix } F \rightarrow \alpha \]
\[ \text{cata } \alpha = \alpha \circ F(\text{cata } \alpha) \circ \text{unfix} \]

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

\[ \text{Fix}( F ) \rightarrow F( \text{Fix}( F )) \]

Fix (Expression_proto) \rightarrow Addition

Fix (Expression_proto) \rightarrow Fix (Expression_proto)
Catamorphism

cata : (F a → a) → Fix F → a

cata α = α ∘ F(cata α) ∘ unfix

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

\[
F \left( \text{Fix}(F) \right) \rightarrow F \ a
\]

Addition

Fix (Expression_proto) → 4

Fix (Expression_proto) → 5

Addition
Catamorphism

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

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

\[
\begin{align*}
F(a) &\to a \\
\text{Addition} &\to 9
\end{align*}
\]
Catamorphism - example

So in our expression example:

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

\[
\begin{align*}
\text{ana} &: (a \rightarrow F a) \rightarrow a \rightarrow \text{Fix } F \\
\text{ana} \circ \bar{\alpha} &= \text{fix} \circ F(\text{ana} \circ \bar{\alpha}) \circ \bar{\alpha} \\
\text{cata} &: (F a \rightarrow a) \rightarrow \text{Fix } F \rightarrow a \\
\text{cata} \circ \alpha &= \alpha \circ F(\text{cata} \circ \alpha) \circ \text{unfix}
\end{align*}
\]

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

\[
\text{ana} : (a \to F a) \to a \to \text{Fix } F
\]
\[
\text{ana } \bar{\alpha} = \text{fix } \circ F(\text{ana } \bar{\alpha}) \circ \bar{\alpha}
\]

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

\[
\text{coalg } n = \text{if}( n == 1 ) (\text{Constant } 1) \quad \text{else}
\]
\[
\text{if}( \text{is\_even}(n) ) (\text{Addition } n/2 \ n/2) \quad \text{else} \quad (\text{Addition } n\!-\!1 \ 1)
\]

\text{decompose} : \text{Integer } \to \text{Expression}
\[
\text{decompose } n = (\text{ana } \text{coalg}) n
\]
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 structured 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.

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

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.

```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)
```

\(^1\) LambdaGen, see András Leitereg’s [github](https://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](https://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:

```
Net  Net  Net
/\   /\   /\   \\
|  Net Net Net
|  /\   /\   /\   \\
| /\   |  |  |  |
|/\   Input1 Input2 Input3
```

Final output

The same net with the same weights operate at each level at each branch!
One more use case for recursion schemes

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

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.

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😊
Thank you!

See the [online paper](#) for more references.

Erik Meijer, J. Hughes, M.M. Fokkinga, Ross Paterson
[F]unctional 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 Williams’ [talk](#)

Recursive Neural Networks


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