

Compiling to Categories

Mathematically-principled program transformation

T. Mark Ellison and Siva Kalyan

November 15, 2017

Australian National University

Table of contents

1. Categories
2. Cartesian-Closed Categories
3. Conal Elliott: Compiling to Categories
4. Declaring Our CCCs
5. Haskell to Lambda Calculus
6. From λ -Expressions to CCC Constructions
7. Example: Syntactic Analysis
8. Example: Interval Analysis

Haskell and Category Theory

Haskell	Category Theory
Category	Category
Type	Object
Function	Morphism
<u>Hask</u>	<u>Set</u>
...	Terminal Objects
Tuple	Product
Currying, Function Application	Cartesian Closure

Categories

Categories

A category $\underline{\mathbf{C}}$ consists of

1. a class $\text{Obj}(\underline{\mathbf{C}})$ of *objects*, and
2. for each pair of objects $A, B \in \text{Obj}(\underline{\mathbf{C}})$, a set $\text{Hom}_{\underline{\mathbf{C}}}(A, B)$ of *arrows* (or *morphisms*) from A to B , known as a *hom-set*.

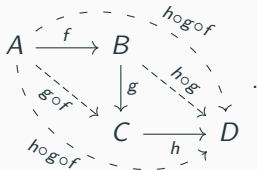
$$\begin{array}{ccc} & \text{Hom}_{\underline{\mathbf{C}}}(A, B) & \\ A & \begin{array}{c} \rightrightarrows \\ \longrightarrow \end{array} & B \end{array}$$

Many familiar parts of Haskell form a category **Hask**: objects are *types* (**Int**, **Char**, etc.), and arrows are *functions* between types (e.g. **ord** :: **Int** → **Char**).

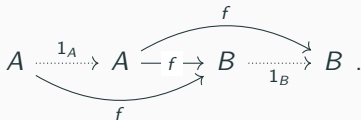
Category Laws

In a category $\underline{\mathbf{C}}$:

1. Given arrows $f: A \rightarrow B$ and $g: B \rightarrow C$ in $\underline{\mathbf{C}}$, the *composition* $g \circ f: A \rightarrow C$ ($= g.f$) is also in $\underline{\mathbf{C}}$.
2. Given arrows $f: A \rightarrow B$, $g: B \rightarrow C$ and $h: C \rightarrow D$,
 $(h \circ g) \circ f = h \circ (g \circ f) = h \circ g \circ f$:



3. Every object $A \in \text{Obj}(\underline{\mathbf{C}})$ is associated with an *identity arrow* $1_A: A \rightarrow A$ ($= \text{id}$). Given any arrow $f: A \rightarrow B$, we have



Examples

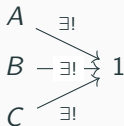
	<u>Set</u>	<u>Hask</u>	<u>POrd</u>	<u>Cat</u>
Objects	sets	types	items	small cats
Morphisms	functions	functions	$a \leq b$	functors
Composition	$f \circ g$	$f.g$	transitivity	$F \circ G$
Identity	1_A	id	$a = a$	$1_{\mathcal{C}}$

Not everything in Haskell can be in Hask if we want it to be a category. Every type in the language contains a **Bottom** (\perp) or **undefined** value, but these 'values' cause mayhem with the category laws (in particular the **Identity** constraint). So when we talk about Hask we'll be talking about vanilla Hask without these abnormal values. (Haskell wiki page on Hask.)

Category Theory: Terminal Objects

A *terminal object* is a type 1 (a.k.a. T) in $\text{Obj}(\underline{\mathbf{C}})$, such that there is only a single mapping from any other type A onto that type:

$$\forall A \in \text{Obj}(\underline{\mathbf{C}}), |\text{Hom}_{\underline{\mathbf{C}}}(A, 1)| = 1.$$



In **Hask**:

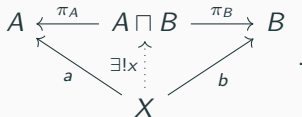
```
1  () -- the type corresponding to 1, containing only itself
2  terminalMap :: t -> ()
3  terminalMap _ = ()
```


Examples

	<u>Set</u>	<u>Hask</u>	<u>POrd</u>	<u>Cat</u>
Objects	sets	types	items	small cats
Morphisms	functions	functions	$a \leq b$	functors
Composition	$f \circ g$	$f.g$	transitivity	$F \circ G$
Identity	1_A	id	$a = a$	$1_{\underline{c}}$
Terminal obj.	$\{*\}$	$()$	upper bound	$\underline{1}$

Products

Given objects A, B in $\underline{\mathbf{C}}$ there may be a (*pairwise*) *product* $A \sqcap B \in \text{Obj}(\underline{\mathbf{C}})$ and *projection arrows* $\pi_A: A \sqcap B \rightarrow A$ and $\pi_B: A \sqcap B \rightarrow B$ such that for any object X in the same category and arrows $a: X \rightarrow A$ and $b: X \rightarrow B$ there is a *unique* arrow $x: X \rightarrow A \sqcap B$ such that $a = \pi_A \circ x$ and $b = \pi_B \circ x$:



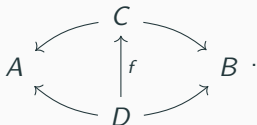
In other words: Given a particular way of mapping X to A and to B , there's only *one* way of mapping X to $A \sqcap B$ such that everything's consistent.

Products

Alternatively, the triplet $\langle A \sqcap B, \pi_A, \pi_B \rangle$ is a *terminal object* in the category whose objects are diagrams of the form

$$A \longleftarrow C \longrightarrow B ,$$

and whose arrows are (commutative) diagrams of the form



Products in Haskell

```
1  (a,b) -- the type containing pairs from types a and b ( $A \times B$ )
2  fst :: (a,b) -> a -- the projection function  $\pi_A$ 
3  fst (x,y) = x
4  snd :: (a,b) -> b -- the projection function  $\pi_B$ 
5  snd (x,y) = y
6  factorThroughProd :: (c -> a) -> (c -> b) -> (c -> (a,b))
7  factorThroughProd f g = \ x -> (f x,g x)
```

It should be obvious that

fst.(factorThroughProd f g) = f, and

snd.(factorThroughProd f g) = g.

Examples

	<u>Set</u>	<u>Hask</u>	<u>POrd</u>	<u>Cat</u>
Objects	sets	types	items	small cats
Morphisms	functions	functions	$a \leq b$	functors
Composition	$f \circ g$	$f.g$	transitivity	$F \circ G$
Identity	1_A	id	$a = a$	$1_{\underline{C}}$
Terminal obj.	$\{*\}$	$()$	upper bound	$\underline{1}$
Product	$A \times B$	(a,b)	$\min(a,b)$	$\underline{C} \times \underline{D}$

Exponential Objects

Given objects A and B in $\underline{\mathbf{C}}$, an *exponential object* B^A (also written $[A \rightarrow B]$) is an object with an arrow eval_B^A such that for any C and any arrow $f: C \sqcap A \rightarrow B$,

$$\begin{array}{ccc} C \sqcap A & & \\ \downarrow \exists! & \searrow f & \\ B^A \sqcap A & \xrightarrow{\text{eval}_B^A} & B \end{array} .$$

Alternatively, the pair $\langle B^A, \text{eval}_B^A \rangle$ constitutes a terminal object in the category whose objects are diagrams of the form

$$C \sqcap A \longrightarrow B ,$$

and whose arrows are commutative diagrams of the form

$$\begin{array}{ccc} D \sqcap A & & \\ \downarrow & \searrow & \\ C \sqcap A & \searrow & B \end{array} .$$

Exponential Objects in Haskell

In Hask, the exponential object of two types `a` and `b` is the *function type* `(a -> b)` (it's akin to the *hom-set* of `a` and `b`). Let's see how this satisfies the above definition.

```
1  eval :: ((a -> b),a) -> b
2  eval (f,x) = f x
3  factoredArrow :: ((c,a) -> b) -> ((c,a) -> ((a -> b),a))
4  factoredArrow f = \ (y,x) -> ((\ x' -> f(y,x')),x)
```

(Spot the currying!)

It can be proven that `eval . (factoredArrow f) = f` — and that `factoredArrow` is the *only* arrow for which this is true.

Cartesian-Closed Categories

Cartesian-Closed Categories (CCC)

There is a terminal object 1 .

There are binary products \square (and hence all finite products).

For any two objects A and B , there is an exponential object B^A .

Examples:

Set the singleton set, pairs, sets of functions

Hask $()$, (a,b) , $a \rightarrow b$

There are more examples, but they're pretty complicated.

Conal Elliott: Compiling to Categories

Compiling to Categories

So far, we have introduced concepts from standard category theory, with a bit of Haskell flavour.

It is well-known that Haskell (or a near-complete subset of it) has category-theoretic semantics (e.g. our last talk), given in terms of a single category **Hask**.

Conal Elliott shows that Category Theory can not only provide semantics, but a range of compile-to domains to which *the same code* can be compiled.

iiiiiii HEAD Single most exciting paper in the interpretation of programming languages. ===== We can map any λ -calculus expression onto a construction in a CCC. The computation resulting from that construction just depends on what that CCC happens to be.

iiiiiii 1fa7e3b0075c2ad1a0fb8384e27a0f67349821de

Compiling to Categories

CONAL ELLIOTT, Target, USA

It is well-known that the simply typed lambda-calculus is modeled by any cartesian closed category (CCC). This correspondence suggests giving typed functional programs a variety of interpretations, each corresponding to a different category. A convenient way to realize this idea is as a collection of meaning-preserving transformations added to an existing compiler, such as GHC for Haskell. This paper describes such an implementation and demonstrates its use for a variety of interpretations including hardware circuits, automatic differentiation, incremental computation, and interval analysis. Each such interpretation is a category easily defined in Haskell (outside of the compiler). The general technique appears to provide a compelling alternative to deeply embedded domain-specific languages.

CCS Concepts: • **Theory of computation** \rightarrow *Lambda calculus*; • **Software and its engineering** \rightarrow *Functional languages*; *Compilers*;

Additional Key Words and Phrases: category theory, compile-time optimization, domain-specific languages

ACM Reference Format:

Conal Elliott. 2017. Compiling to Categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 27 (September 2017), 27 pages.
<https://doi.org/10.1145/3110271>

Compiling to Categories

- you specify (using Haskell classes) the application category
- then Haskell code is compiled to constructions in that category
- compile your Haskell code to: *category theory* (for e.g. theorem proving), *interval analysis* (for correctness proof), *hardware* (!!!), incrementalisation, *automatic differentiation*, and more.

Compiling to Categories

In various levels of prototype, you can compile to these categories

free CCC pretty-printing, syntax-highlighting, or proving correctness

intervals verification

delta partial memoisation

hardware translate software into hardware

linear spaces linear approximations to complex numeric functions

differentials differentiate any haskell numeric function - automatically

Compiling to Categories: Overview

- compile Haskell \rightarrow λ -expressions (grab intermediate output from GHC)
- λ -expressions \rightarrow CCC-constructions
- apply CCC-constructions
- write result

Declaring Our CCCs

Example: we want to compile numeric expressions/functions into something that tells us about the bounds on outputs (minimum possible output and maximum possible output).

This cannot be achieved with a black-box 2nd-order function, except by enumerating possible inputs.

But can be achieved by compilation to categories.

First we define the **type family** of intervals. Here **:*** is a pairing operator.

```
1  type family lv a
2  type instance lv ()    = ()
3  type instance lv Float = Float :* Float
4  type instance lv Double = Double :* Double
5  type instance lv Int   = Int    :* Int
```

Now we define our category. First the data type **IF** which contains our morphisms.

```
1 data IF a b = IF { unIF :: lv a -> lv b }
```

I'm using `pack0`, `pack1`, `pack2` to map functions of 0-, 1- and 2-arguments in Hask into the new category. CE uses `pack`, `inNew` and `inNew2`.

```
1 instance Category IF where
2   id = pack0 id
3   (.) = pack2 (.)
```

```
1 instance ProductCat IF where
2   exl = pack0 exl
3   exr = pack0 exr
4   (&&&) = pack2 (&&&)
```

```
1 instance ClosedCat IF where
2   apply = pack0 apply
3   curry = pack1 curry
4   uncurry = pack1 uncurry
```

```
1 instance lv b ~ (b :* b) => ConstCat IF b where
2   const b = pack0 (const (b,b))
3   unitArrow b = pack0 (unitArrow (b,b))
```

Now define how some atomic Haskell functions map into the CCC.

```
1 instance (lv a ~ (a :* a), Num a, Ord a) => NumCat IF a where  
2   negateC = pack0 (\ (al,ah) -> (-ah, -al))  
3   addC = pack0 (\ ((al,ah),(bl,bh)) -> (al+bl,ah+bh))  
4   subC = addC . second negateC  
5   mulC = pack0 (\ ((al,ah),(bl,bh)) ->  
6       let cs = ((al*bl, al*bh),(ah*bl,ah*bh)) in  
7       (min4 cs, max4 cs))
```


Haskell to Lambda Calculus

Compiling to λ -expressions

Credit: <https://stackoverflow.com/questions/27635111>.

- use the `GHC` module
- functions `compileToCoreModule` or `compileToCoreSimplified` to compile a file
- the code has been reproduced as `processor.hs` in the repository with today's talk. You need to compile it with

```
1      $ ghc --package ghc --package ghc-paths processor.hs
```

Haskell to λ -Calculus

```
1 example :: Int -> Int -> Int
2 example x y = x + y
```

```
1 example = \ (x :: Int) (y :: Int) -> + @ Int $fNumInt x y
```

Haskell to λ -Calculus

```
1 example :: Int -> Int -> Int
2 example x y = x + y
```

```
1 example = \ (x :: Int) (y :: Int) -> + @ Int $fNumInt x y
```

From λ -Expressions to CCC Constructions

The mapping operation is implemented as a pseudo-function `ccc`.

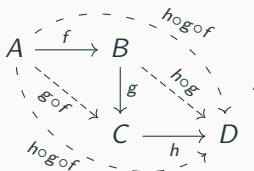
Each transformation either reduces the size of the body of the λ -expression, or eliminates a λ .

Consequently, the transformation process must terminate.

Category Definition

$$\text{Hom}_{\underline{\mathbf{C}}}(A, B)$$

$$A \begin{array}{c} \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{array} B$$



$$A \xrightarrow{\quad 1_A \quad} A \xrightarrow{\quad f \quad} B \xrightarrow{\quad 1_B \quad} B .$$

Curved arrows above and below the straight arrows $A \xrightarrow{f} B$ also represent the morphism f .

Category Definition

- composition $g \circ f = \lambda x \mapsto g(f(x))$.
- identity $\text{id} = \lambda x \mapsto x$,

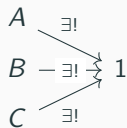
Laws:

- $\text{id} \circ f \equiv f \circ \text{id} \equiv f$
- $h \circ (g \circ f) \equiv (h \circ g) \circ f$

Expression Body is a Single Variable

$$\text{ccc}(\lambda x \mapsto x) = \text{id}$$

A Terminal Object



A Terminal Object

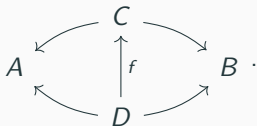
- terminal 1 is the terminal object in the category,
- terminal arrow $\text{it} = \lambda a \mapsto ()$.
- $\text{unitarrow unitarrow } b = \lambda () \mapsto b$.
- constants $\text{const } b = (\text{unitarrow } b) \circ \text{it}$

Laws:

- $\text{it} \circ f \equiv \text{it}$

$$\text{ccc}(\lambda x \mapsto c) = \text{const } c$$

The Product



The Product

- $\text{fork } f \Delta g = \lambda x \mapsto (f\ x, g\ x),$
- $\text{extract-left } \text{exl} = \lambda (a, b) \mapsto a,$
- $\text{extract-right } \text{exr} = \lambda (a, b) \mapsto b.$

Laws:

- $\text{exl} \circ (f \Delta g) \equiv f$
- $\text{exr} \circ (f \Delta g) \equiv g$
- $\text{exl} \circ h \Delta \text{exr} \circ h \equiv h$

Exponential Objects

$$\begin{array}{ccc} C \sqcap A & & \\ \downarrow \exists! & \searrow f & \\ B^A \sqcap A & \xrightarrow{\text{eval}_B^A} & B \end{array} .$$

Exponential Objects

- apply or eval $\text{apply}(f, x) = f\ x$
- curry $f = \lambda a\ b \mapsto f(a, b)$
- uncurry $f = \lambda (a, b) \mapsto f\ a\ b$
- constant functions $\text{constFun}\ f = \text{curry}(f \circ \text{exr}) = \lambda x \mapsto f\ \text{ignores}\ x$, returns a function

Laws:

- $\text{uncurry}(\text{curry}\ f) \equiv f$
- $\text{curry}(\text{uncurry}\ f) \equiv f$
- $\text{apply} \circ (\text{curry}\ f \circ \text{exl} \Delta \text{exr}) \equiv f$

Expression Body is an Application

Expression body is an application

$$\text{ccc}(\lambda x \mapsto U\ V) = \text{apply}\circ(\text{ccc}(\lambda x \mapsto U) \Delta \text{ccc}(\lambda x \mapsto V))$$

Lambda abstraction

$$\text{ccc}(\lambda x \mapsto \lambda y \mapsto U) = \text{curry } \text{ccc}(\lambda (x, y) \mapsto U)$$

Constant functions

$$\text{ccc}(\lambda x \mapsto f) = \text{constFun } \text{ccc}(f)$$

f may need to be *Curried* to reduce its argument dimensionality.

Example: Syntactic Analysis

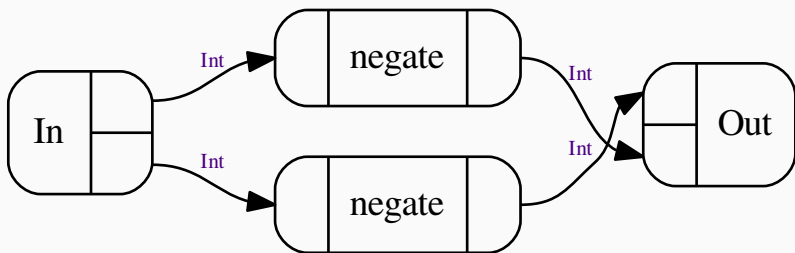
The simplest application is just to build a tree structure of the functions applying in the CCC.

Each function just records a label (the same as its name) on a tree node, and then builds subtrees from any arguments.

Example: Interval Analysis

Negation

```
1  instance (lv a ~ (a :* a), Num a, Ord a) => NumCat IF a where
2      negateC = pack (\ (al,ah) -> (-ah, -al))
3      ..
4      {-# INLINE negateC #-}
5      ..
```



Addition

```
1  instance (lv a ~ (a :* a), Num a, Ord a) => NumCat IF a where
2      ..
3      addC = pack \ ((al,ah),( bl,bh)) -> (al+bl,ah+bh))
4      ..
5      {-# INLINE addC #-}
6      ..
```

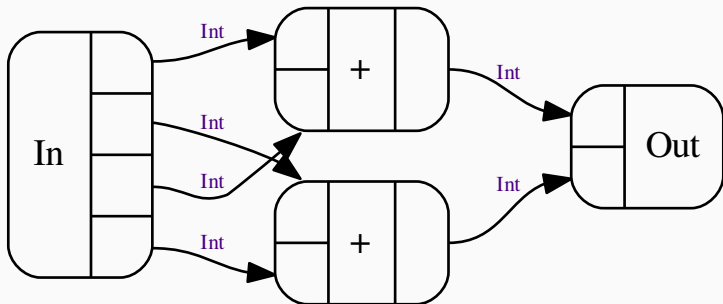
Addition

```
1 runSynME "add" $ toCcc $ ivFun $ uncurry ((+) @Int)
```

```
1 uncurry (curry (apply . (exl &&& exr))) .  
2 (curry  
3 (  
4   (add . (exl . exl &&& exl . exr)  
5     &&&  
6     add . (exr . exl &&& exr . exr)  
7   ) . exr  
8 ) &&& id  
9 )
```

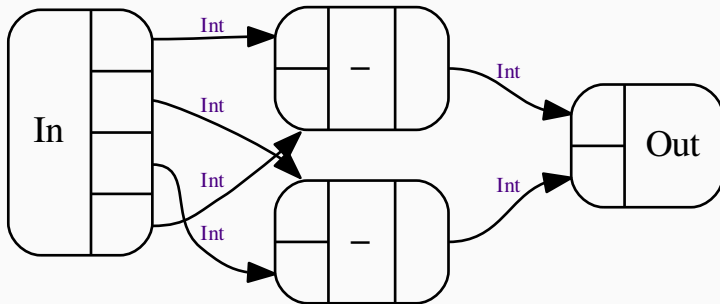
Addition

1 `runSynCirc "add" $ toCcc $ ivFun $ uncurry ((+) @Int)`



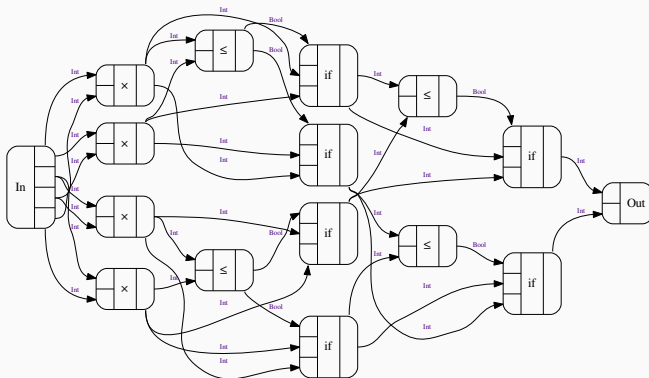
Subtraction

```
1  instance (lv a ~ (a :* a), Num a, Ord a) => NumCat IF a where  
2    ..  
3    subC = addC . second negateC  
4    ..  
5    {-# INLINE subC #-}  
6    ..
```



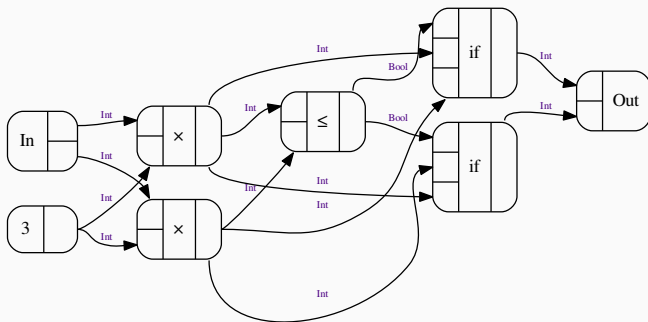
Multiplication

```
1  instance (lv a ~ (a :* a), Num a, Ord a) => NumCat IF a where
2      mulC = pack \ ((al,ah),(bl,bh)) ->
3          let cs = ((al*bl, al*bh),(ah*bl, ah*bh)) in
4              (min4 cs, max4 cs)
5      ..
6      {-# INLINE mulC #-}
```



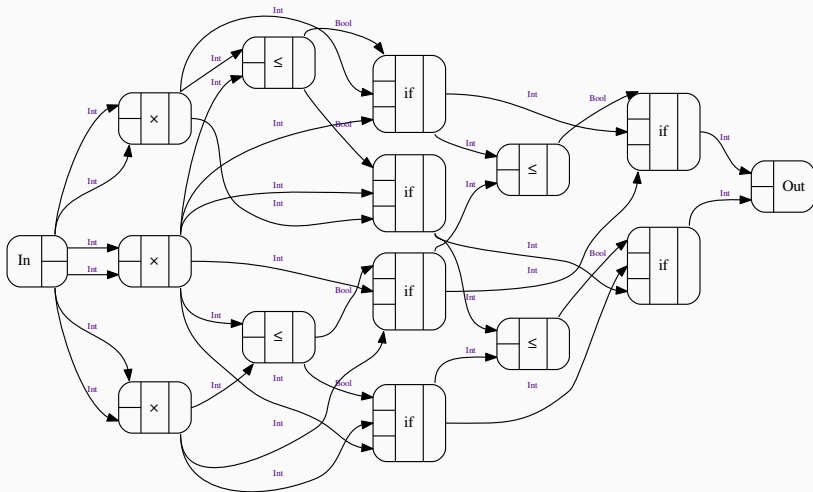
1

```
runSynCirc "thrice-iv" $ toCcc $ ivFun $ \ x -> 3 * x :: Int
```



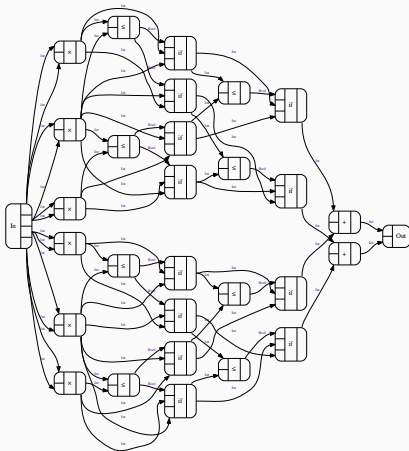
Square

1 `runSynCirc "sqr-iv" $ toCcc $ ivFun $ sqr @Int`



Magic Square

1 `runSynCirc "magSqr-iv" $ toCcc $ ivFun $ magSqr @Int`



Example: Hardware

Verilog: a language for integrated circuit design.

Example: Category Products

Example: Linear maps and automatic differentiation

Linear maps as a category

A **linear map** is a function $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$ such that $f(x + y) = f(x) + f(y)$ and $f(cx) = c f(x)$. It can also be thought of as an $n \times m$ matrix (where the columns tell you what the basis vectors of \mathbb{R}^m map to).

Linear maps form a category, because:

1. Given $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $g: \mathbb{R}^n \rightarrow \mathbb{R}^p$, we can define the composition $g \circ f: \mathbb{R}^m \rightarrow \mathbb{R}^p$, which is also a linear map.
2. Composition of linear maps (alternatively: matrix multiplication) is associative.
3. For any vector space \mathbb{R}^n , the identity function $1_{\mathbb{R}^n}: \mathbb{R}^n \rightarrow \mathbb{R}^n$ is a linear map, and has the properties we expect of an identity.

Types of differentiation

Symbolic differentiation Rule-based manipulation of algebraic expressions; cumbersome for computers.

Numeric differentiation Evaluate the function at two nearby points and compute the slope of the resulting line; easy for computers, not so useful for humans.

Automatic differentiation Tell the computer how to compute the derivatives of simple functions, and it will tell you how to compute the derivative of any composition of those functions. Easy for a computer, useful for humans. But takes more work to set up, and only works for functions that are analytically differentiable.

The chain rule

$$(g \circ f)' = (g' \circ f) \cdot (f')$$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

$$\frac{d g(f(x))}{dx} = \frac{d g(f(x))}{d f(x)} \frac{d f(x)}{dx}$$

In higher dimensions, the derivative of a function is a vector or matrix of partial derivatives, and the derivative of a composition of two functions is the product of the two matrices that give the derivatives of the individual functions.

Derivatives, linear maps, and the chain rule

The derivative of a function at a point is a **linear map** (a line, plane, linear subspace; equivalently, a matrix giving the slope(s) corresponding to a unit move in each dimension). In the category of linear maps, composition is **multiplication** (of matrices, which reduces to scalar multiplication for 1×1 matrices). Differentiation is an operation *deriv* with the property that

$$\text{deriv}(g \circ f) = (\text{deriv } g \circ f) \circ (\text{deriv } f).$$

where the second \circ on the right-hand side is in the category of linear maps (i.e. matrix multiplication). (NB: **not** a functor!)

Thus, if we know how to apply *deriv* to all our atomic functions, then we know how to apply *deriv* to all compositions of these functions.

Differentiation as a functor

But we want differentiation to be a functor! The solution is to represent every differentiable function as a *pair* (f, f') , (g, g') , etc., and define composition as

$$(g, g') \circ (f, f') = (g \circ f, (g' \circ f) \cdot f').$$

Then *deriv* is just $snd \circ andDeriv$ (more explanation to come). It is obviously a functor.

Implementation of automatic differentiation in Haskell

Instead of letting functions have types such as $a \rightarrow b$, we require them to have the type $a \rightarrow b \times (a \multimap_s b)$. In other words, take an input value, and return not only the value of the function at that point, but also another (linear) function that gives you the derivative at that point.

Chain rule:

$$Dg \circ Df = D(\lambda a \mapsto \text{let } \{(b, f') = f\ a; (c, g') = g\ b\} \text{ in } (c, g' \circ f'))$$

(The expression following D on the right-hand side defines how we compose differentiable functions.)

Possibilities

Language-to-Language Translation

CCC like the Syntax CCC, but constructing code according to the rules of language X (where X is Python, JavaScript, TypeScript, PHP, R, etc.)

Use writing type-checked code which can be used in language-specific environments (e.g. JavaScript in browser, R because you need to use an R-only library, PostScript for your printer, etc.)

Difficulties Missing abstractions in some languages (higher order functions in FORTRAN)

CCC each type a replaced by $\text{Dist } a$, the distribution over values of type a . Function f from a to b replaced by function f' from distributions over a to the resulting distribution over b under the action of f .

Use take a model mapping independent variables to dependent variables, supply distributions to the independent variables, work out expected distribution of outputs. Calculate z-scores (likelihoods) trivially from deterministic models and distributions over dependent inputs.

Difficulties The mapping of distributions may be difficult for some primitive functions.

CCC objects are predicates over a possibly-composite value,
maps are deductions from predicates over inputs to
predicates over outputs

Use proving program correctness - each computational step
maps onto the deduction about output that it corresponds
to

This approach to compilation extends the mathematical rigour of Haskell (et al) to implementation domains.

Conclusions

Further Reading

Further Reading