Compiling to Categories

Mathematically-principled program transformation

T. Mark Ellison and Siva Kalyan

November 15, 2017

Australian National University

Table of contents

Haskell and Category Theory

Haskell	Category Theory
Category	**Category**
Type	**Object**
Function	**Morphism**
Hask	**Set**
...	**Terminal Objects**
Tuple	**Product**
Currying, Function Application	**Cartesian Closure**

Closed Cartesian Categories in Category Theory

Categories

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

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

$$A \overset{\mathrm{Hom}_{\underline{\mathbf{C}}}(A,B)}{\Rrightarrow} B$$

Many familiar parts of Haskell form a category $\underline{\mathbf{Hask}}$: 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 $f\colon A \to B$ and $g\colon B \to C$ in $\underline{\mathbf{C}}$, the *composition* $g \circ f\colon A \to C$ $(= \texttt{g.f})$ is also in $\underline{\mathbf{C}}$.
2. Given arrows $f\colon A \to B$, $g\colon B \to C$ and $h\colon C \to D$, $(h \circ g) \circ f = h \circ (g \circ f) = h \circ g \circ f$:

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, dashed, "g\circ f"'] \arrow[drr, dashed, bend left=60, "h\circ g\circ f"] \arrow[drr, dashed, bend right=60, "h\circ g\circ f"'] & B \arrow[d, "g"] \arrow[dr, dashed, "h\circ g"] & \\
& C \arrow[r, "h"'] & D
\end{tikzcd}
.
$$

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

$$
\begin{tikzcd}
A \arrow[r, dotted, "1_A"] \arrow[rr, bend right, "f"'] & A \arrow[r, "f"] \arrow[rr, bend left, "f"] & B \arrow[r, dotted, "1_B"'] & B
\end{tikzcd}
.
$$

Examples

	Set	**Hask**	**POrd**	**Cat**
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**	$a = a$	$1_{\underline{\mathbf{C}}}$

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 $\mathrm{Obj}(\underline{\mathbf{C}})$, such that there is only a single mapping from any other type A onto that type:

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

$$\begin{tikzcd}
A \arrow[dr, "\exists!"] & \\
B \arrow[r, dashed, "\exists!"] & 1 \\
C \arrow[ur, "\exists!"'] &
\end{tikzcd}$$

In **Hask**:

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

Examples

	Set	**Hask**	**POrd**	**Cat**
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**	$a = a$	$1_{\underline{\mathbf{C}}}$
Terminal obj.	$\{*\}$	`()`	upper bound	$\underline{\mathbf{1}}$

Products

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

$$\begin{tikzcd}
A & A \sqcap B \arrow[l, "\pi_A"'] \arrow[r, "\pi_B"] & B \\
& X \arrow[ul, "a"] \arrow[u, dotted, "\exists! x"] \arrow[ur, "b"'] &
\end{tikzcd} .$$

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

$$\begin{tikzcd} & C \arrow[dl, bend right] \arrow[dr, bend left] & \\ A & & B \\ & D \arrow[uu, "f"'] \arrow[ul, bend left] \arrow[ur, bend right] & \end{tikzcd} \ .$$

Products in Haskell

```haskell
(a,b) -- the type containing pairs from types a and b (A ⊓ B)
fst :: (a,b) -> a -- the projection function π_A
fst (x,y) = x
snd :: (a,b) -> b -- the projection function π_B
snd (x,y) = y
factorThroughProd :: (c -> a) -> (c -> b) -> (c -> (a,b))
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

	$\underline{\mathbf{Set}}$	$\underline{\mathbf{Hask}}$	$\underline{\mathbf{POrd}}$	$\underline{\mathbf{Cat}}$
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**	$a = a$	$1_{\underline{\mathbf{C}}}$
Terminal obj.	$\{*\}$	`()`	upper bound	$\underline{\mathbf{1}}$
Product	$A \times B$	`(a,b)`	$\min(a, b)$	$\underline{\mathbf{C}} \times \underline{\mathbf{D}}$

Exponential Objects

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

$$\begin{tikzcd}
C \sqcap A \arrow[d, dotted, "\exists!"'] \arrow[dr, "f"] & \\
B^A \sqcap A \arrow[r, "\mathrm{eval}_B^A"'] & B
\end{tikzcd} .$$

Alternatively, the pair $\langle B^A, \mathrm{eval}_B^A \rangle$ 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

$$\begin{tikzcd}
D \sqcap A \arrow[d] \arrow[dr, bend left] & \\
& B \\
C \sqcap A \arrow[ur, bend right] &
\end{tikzcd} .$$

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.

```haskell
eval :: ((a -> b),a) -> b
eval (f,x) = f x
factoredArrow :: ((c,a) -> b) -> ((c,a) -> ((a -> b),a))
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 (CCC)

There is a terminal object 1.

There are binary products $\sqcap$ (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 -> 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**.

Elliott's (2017) paper *Compiling to Categories* (hereafter C2C) shows that Category Theory can not only provide semantics, but a range of compile-to domains to which *the same code* can be compiled.

Single most exciting paper in the interpretation of programming languages.

Compiling to Categories

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** → *Lambda calculus*; • **Software and its engineering** → *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

How it works (black box):

- you specify (using Haskell classes) the application category
- then Haskell code is compiled to constructions in that category
- while the constructions reflect the structure of your program, they do not simply implement it.

Compiling to Categories

Why this is exciting:

By choosing different CCCs, you can do these things (CCC names not the same as in C2C, but you can work it out):

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

How it works (under the hood):

- compile Haskell $\to$ λ-expressions (grab intermediate output from GHC)
- λ-expressions $\to$ CCC-constructions
- CCC-constructions applied in category of choice
- output result

Declaring 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.

Declaring CCCs

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

```
type family Iv a
type instance Iv ()     = ()
type instance Iv Float  = Float :* Float
type instance Iv Double = Double :* Double
type instance Iv Int    = Int    :* Int
```

Declaring CCCs

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

```haskell
data IF a b = IF { unIF :: Iv a -> Iv b }
```

Declaring CCCs

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

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

Declaring CCCs

```
instance ProductCat IF where
  exl   = pack0 exl
  exr   = pack0 exr
  (&&&) = pack2 (&&&)
```

Declaring CCCs

```
instance ClosedCat IF where
  apply   = pack0 apply
  curry   = pack1 curry
  uncurry = pack1 uncurry
```

Declaring CCCs

```
instance Iv b ~ (b :* b) => ConstCat IF b where
  const b = pack0 (const (b,b))
  unitArrow b = pack0 (unitArrow (b,b))
```

Declaring CCCs

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

```haskell
instance (Iv a ~ (a :* a), Num a, Ord a) => NumCat IF a where
  negateC = pack0 (\ (al,ah) -> (-ah, -al))
  addC = pack0 (\ ((al,ah),(bl,bh)) -> (al+bl,ah+bh))
  subC = addC . second negateC
  mulC = pack0 (\ ((al,ah),(bl,bh)) ->
                  let cs = ((al*bl, al*bh),(ah*bl,ah*bh)) in
                    (min4 cs, max4 cs))
```

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

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

Haskell to λ-Calculus

```haskell
example :: Int -> Int -> Int
example x y = x + y
```

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

Haskell to λ-Calculus

```haskell
example :: Int -> Int -> Int
example x y = x + y
```

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

From λ-Calculus to CCCs

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

$$\begin{tikzcd} A \arrow[r, shift left=2, "\mathrm{Hom}_{\underline{\mathbf{C}}}(A,B)"] \arrow[r] \arrow[r, shift right=2] & B \end{tikzcd}$$

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, dashed, "g\circ f"'] \arrow[drr, dashed, bend left=60, "h\circ g\circ f"] \arrow[drr, dashed, bend right=60, "h\circ g\circ f"'] & B \arrow[d, "g"] \arrow[dr, dashed, "h\circ g"] & \\ & C \arrow[r, "h"'] & D \end{tikzcd} .$$

$$\begin{tikzcd} A \arrow[r, dotted, "1_A"] \arrow[rr, bend right, "f"'] & A \arrow[r, "f" description] \arrow[rr, bend left, "f"] & B \arrow[r, dotted, "1_B"'] & B \end{tikzcd} .$$

Category Definition

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

Laws:

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

Expression Body is a Single Variable

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

A Terminal Object

$$\begin{tikzcd}
A \arrow[dr, "\exists!"] & \\
B \arrow[r, "\exists!"] & 1 \\
C \arrow[ur, "\exists!"'] &
\end{tikzcd}$$

A Terminal Object

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

Laws:

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

Simple Constants

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

The Product

$$
\begin{tikzcd}
 & C \arrow[dl, bend right] \arrow[dr, bend left] & \\
A & & B \; . \\
 & D \arrow[ul, bend left] \arrow[ur, bend right] \arrow[uu, "f"'] &
\end{tikzcd}
$$

The Product

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

Laws:

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

Exponential Objects

$$\begin{tikzcd}
C \sqcap A \arrow[d, dotted, "\exists!"'] \arrow[dr, "f"] & \\
B^A \sqcap A \arrow[r, "\mathrm{eval}_B^A"'] & B
\end{tikzcd} .$$

Exponential Objects

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

Laws:

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

Expression Body is an Application

Expression body is an application

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

Lambda abstraction

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

Constant functions

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

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

Examples

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 substrees from any arguments.

Syntactic Analysis

```haskell
appt :: String -> [DocTree] -> DocTree
appt = Node . const . text
-- appt s ts = Node (const (text s)) ts
```

Syntactic Analysis

```
atom :: Pretty a => a -> Syn a b
atom a = Syn (Node (ppretty a) [])

app0 :: String -> Syn a b
app0 s = Syn (appt s [])

app1 :: String -> Syn a b -> Syn c d
app1 s (Syn p) = Syn (appt s [p])

app2 :: String -> Syn a b -> Syn c d -> Syn e f
app2 s (Syn p) (Syn q) = Syn (appt s [p,q])
```

Syntactic Analysis

```
instance Category Syn where
  id  = app0 "id"
  (.) = app2 "."
```

Syntactic Analysis

```
instance ProductCat Syn where
  exl   = app0 "exl"
  exr   = app0 "exr"
  (&&&) = app2 "&&&"
  ...
```

Syntactic Analysis

```
instance TerminalCat Syn where
  it = app0 "it"
```

Syntactic Analysis

```
instance ClosedCat Syn where
  apply   = app0 "apply"
  curry   = app1 "curry"
  uncurry = app1 "uncurry"
```

Syntactic Analysis

```
instance BoolCat Syn where
  notC = app0 "not"
  andC = app0 "andC"
  orC  = app0 "orC"
  xorC = app0 "xorC"
```

Syntactic Analysis

```
instance NumCat Syn a where
  negateC = app0 "negate"
  addC    = app0 "add"
  subC    = app0 "sub"
  mulC    = app0 "mul"
  powIC   = app0 "powI"
```

Syntactic Analysis

and more code to do with pretty printing, etc.

Program Graphs

Transforms programs into data-flow graphs, which can be visualised via graphviz.

Interval Analysis

We used this example in showing the declaration of CCCs.

Here some example outputs, shown as syntax tree, and in graph form.

Interval Analysis

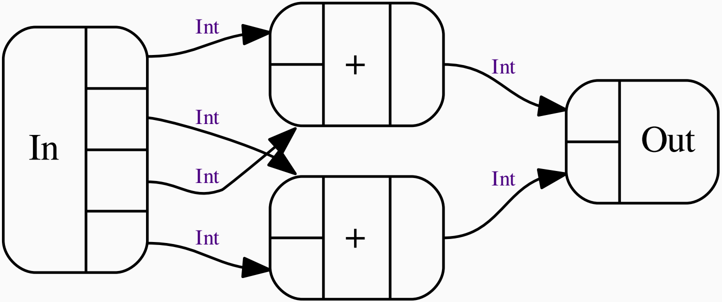
```haskell
instance (Iv a ~ (a :* a), Num a, Ord a) => NumCat IF a where
  ..
  addC = pack (\ ((al,ah),(bl,bh)) -> (al+bl,ah+bh))
  ..
  {-# INLINE addC #-}
  ..
```

Interval Analysis

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

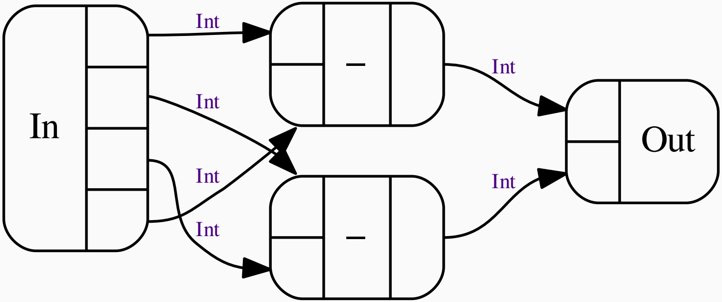
```
uncurry (curry (apply . (exl &&& exr))) .
(curry
 (
  (add . (exl  .  exl  &&& exl . exr)
   &&&
   add . (exr  .  exl  &&& exr . exr)
  )  .  exr
 ) &&& id
)
```

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



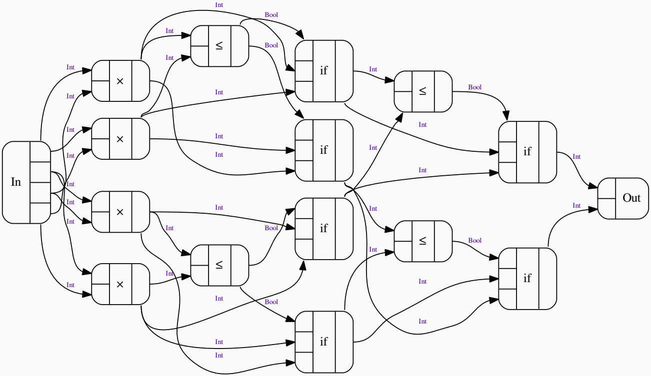
Interval Analysis

```
instance (Iv a ~ (a :* a), Num a, Ord a) => NumCat IF a where
  ..
  subC = addC . second negateC
  ..
  {-# INLINE subC #-}
  ..
```



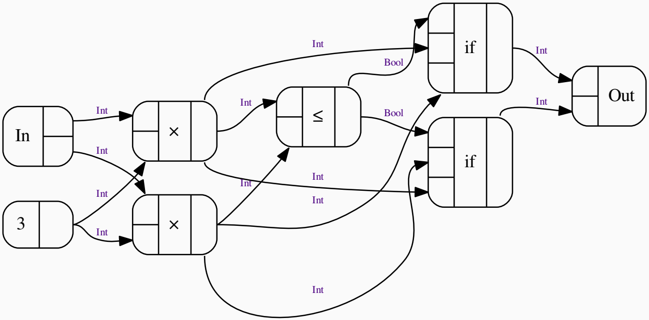
Interval Analysis

```haskell
instance (Iv a ~ (a :* a), Num a, Ord a) => NumCat IF a where
  mulC = pack (\ ((al,ah),(bl,bh)) ->
                let cs = ((al*bl, al*bh),(ah*bl,ah*bh)) in
                  (min4 cs, max4 cs))
  ..
  {-# INLINE mulC #-}
```



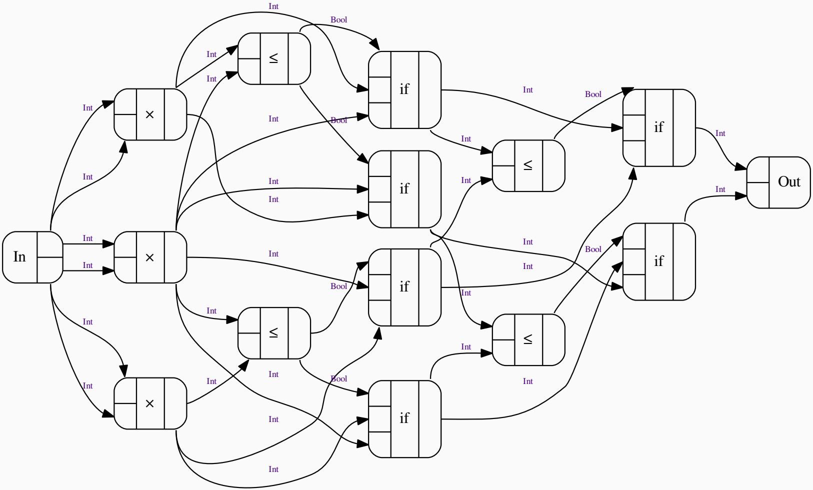
Interval Analysis

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



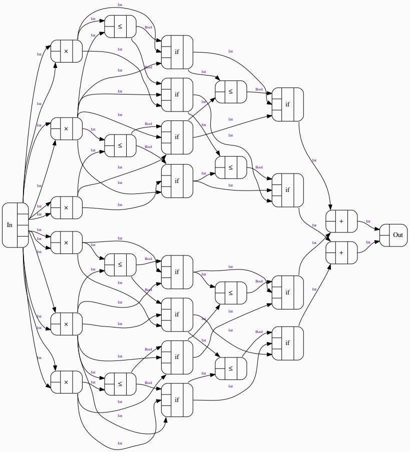
Interval Analysis

```
runSynCirc "sqr−iv"   $ toCcc $ ivFun $ sqr @Int
```



Interval Analysis

```
runSynCirc "magSqr−iv" $ toCcc $ ivFun $ magSqr @Int
```



Compiling to Hardware

Verilog: a language for integrated circuit design.

Verilog code can be realised in a CCC. So Haskell programs can be compiled to silicon.

Compiling to Hardware

```haskell
runVerilog' "adder" $ \(x :: Int, y :: Int) -> x + y
```

```verilog
module adder (clk, n0_0_d, n0_1_d, n1_0_q);
  input clk;
  input [31:0] n0_0_d;
  input [31:0] n0_1_d;
  output [31:0] n1_0_q;
  reg [31:0] n0_0;
  reg [31:0] n0_1;
  reg [31:0] n1_0_q;
  always @(posedge clk)
    begin
      n0_0 <= n0_0_d;
      n0_1 <= n0_1_d;
      n1_0_q <= n1_0;
    end
  assign n1_0 = n0_0 + n0_1;
endmodule
```

Compiling to Hardware

```
runVerilog' "cond" $ \(p :: Bool, x :: Int, y :: Int) -> if p then x else y
```

```
module cond (clk, n0_0_d, n0_1_d, n0_2_d, n1_0_q);
  input clk;
  input n0_0_d;
  input [31:0] n0_1_d;
  input [31:0] n0_2_d;
  output [31:0] n1_0_q;
  reg n0_0;
  reg [31:0] n0_1;
  reg [31:0] n0_2;
  reg [31:0] n1_0_q;
  always @(posedge clk)
    begin
      n0_0 <= n0_0_d;
      n0_1 <= n0_1_d;
      n0_2 <= n0_2_d;
      n1_0_q <= n1_0;
    end
  assign n1_0 = n0_0 ? n0_1 : n0_2;
endmodule
```

Compiling to Hardware

```
runVerilog' "odd" $ \(x :: Int) -> x `mod` 2
```

```
module odd (clk, n0_0_d, n2_0_q);
  input clk;
  input [31:0] n0_0_d;
  output [31:0] n2_0_q;
  reg [31:0] n0_0;
  reg [31:0] n2_0_q;
  wire [31:0] n1_0;
  always @(posedge clk)
    begin
      n0_0 <= n0_0_d;
      n2_0_q <= n2_0;
    end
  assign n1_0 = 32'h2;
  assign n2_0 = n0_0 % n1_0;
endmodule
```

Linear maps as a category

A **linear map** is a function $f\colon \mathbb{R}^m \to \mathbb{R}^n$ 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 $f\colon \mathbb{R}^m \to \mathbb{R}^n$ and $g\colon \mathbb{R}^n \to \mathbb{R}^p$, we can define the composition $g \circ f\colon \mathbb{R}^m \to \mathbb{R}^p$, 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 $1_{\mathbb{R}^n}\colon \mathbb{R}^n \to \mathbb{R}^n$ 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

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

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

Differentiation as a functor

But we want differentiation to be a functor! The solution is to first map every differentiable function f to a pair (f, f') (via *andDeriv*), and define composition of such pairs as

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

Then *deriv* is just *snd* $\circ$ *andDeriv*—and this is obviously a functor.

Implementation of automatic differentiation in Haskell

$$\mathit{deriv} :: (a \to b) \to (a \to (a \multimap_s b))$$

So *deriv* takes a differentiable function and returns a function that associates each input value with a linear map.

$$\mathit{andDeriv}\, f = D(f \,\Delta\, \mathit{deriv}\, f)$$

D is a type constructor for function/derivative pairs. As mentioned before, $\mathit{deriv} = \mathit{snd} \circ \mathit{andDeriv}$. NB: not quite.

Chain rule:

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

(This is exactly the expression from the previous slide. Confusingly, f and g now refer to function/derivative pairs, but f' and g' refer to the derivatives.)

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.)

Probabilistic Modelling

CCC each type a replaced by `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.

Correctness Proofs

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

Rigour

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

Summary and Conclusion

C2C offers a mathematically principled way to do program transformation by

1. defining the implementation level as a class of categories (CCCs)
2. showing how any Haskell program can be mapped onto constructions in those categories
3. offering some exciting sample translators for programs:
    - syntactic trees
    - data-flow graphs
    - bound calculation
    - hardware implementation
    - linear approximation maps
    - automatic differentiation
    - incremental adjustment

   with more possibilities to come
4. showing the way to mathematically principled remapping of code.

Links

- `http://conal.net/papers/compiling-to-categories/` is the homepage for this project. There you will find links to the paper we've discussed here, slides from Elliott's own talk on this, links to a youtube lecture, and the link to the repository which we drew code/output from.
- `https://github.com/tyrannomark/CategoryTheory4Haskellions` has the slides for this talk as `ConCat-talk-20171115.pdf`.

C. Elliott.
Compiling to categories.
Proc. ACM Program. Lang., 1(ICFP), Sept. 2017.