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**

Categories

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*.

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

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* $g \circ f\colon A \to C$ ($=$ `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* $1_A\colon A \to A$ ($=$ **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:

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

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

In **Hask**:

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

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	$\underline{\mathbf{1}}$

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

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

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^A_B such that for any C and any arrow $f\colon C \sqcap A \to B$,

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

Alternatively, the pair $\langle B^A, \mathrm{eval}^A_B \rangle$ 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 **<u>Hask</u>**, 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.

```
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

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.

CCC Constructions and the λ-Calculus

CCC Constructions in the λ-Calculus

We can give a λ-calculus expression which corresponds to each construction in the CCC.

But the reverse is also true.

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.

Category Definition

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

The Product

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

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 $\texttt{const}\, b = (\texttt{unitarrow}\, b) \circ \texttt{it}$

Exponential Objects

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

From λ-Calculus to CCCs

From λ-Calculus to CCCs

This direction is simpler.

There are only 5 main cases we need to deal with.

The mapping operation is symbolised as $\mathfrak{K}$.

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

1. Expression Body is a Single Variable

$$\mathfrak{K}(\lambda\, x \mapsto x) = \texttt{id}$$

2. Expression Body is an Application

$$\mathfrak{K}(\lambda\, x \mapsto U\, V) = \mathtt{apply} \circ (\mathfrak{K}(\lambda\, x \mapsto U) \,\Delta\, \mathfrak{K}(\lambda\, x \mapsto V))$$

3. Lambda Abstraction

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

4. Case Expressions

(more complexity than we wish to cover here)

5a. Simple Constants

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

5b. Constant Functions

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

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

From Haskell to CCC

Haskell to CCC Constructions

- `ghc` compiles Haskell code to λ-calculus
- `simplifier` reduces the λ-calculus size where possible
- `concat` intervenes in the simplifier and converts the λ-calculus to CCC constructions

Looking at GHC Intermediate Stages

Following the stackoverflow answer:
`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
```

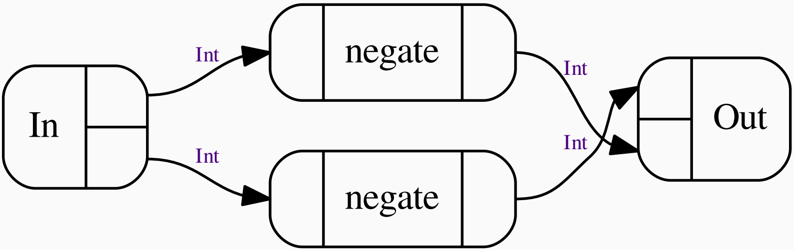
λ-Calculus to CCC Constructions

Example: Syntactic Analysis

Example: Interval Analysis

Negation

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



Addition

```
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 #-}
  ..
```

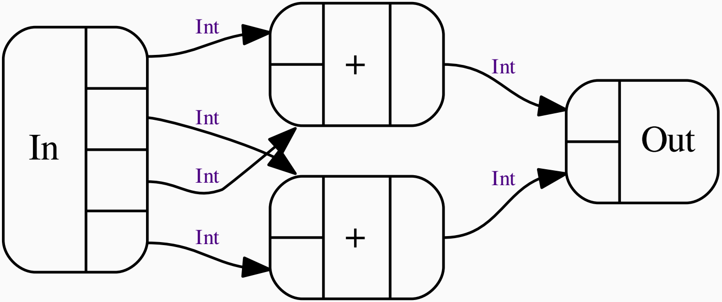
Addition

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

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

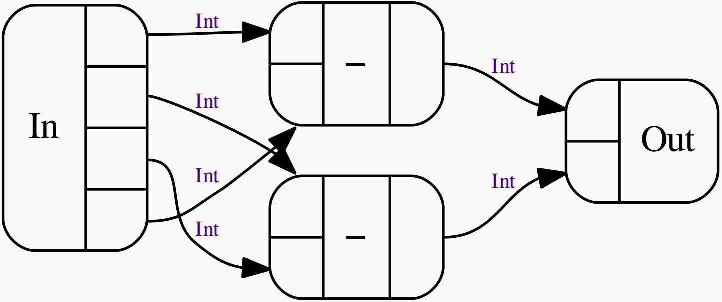
Addition

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



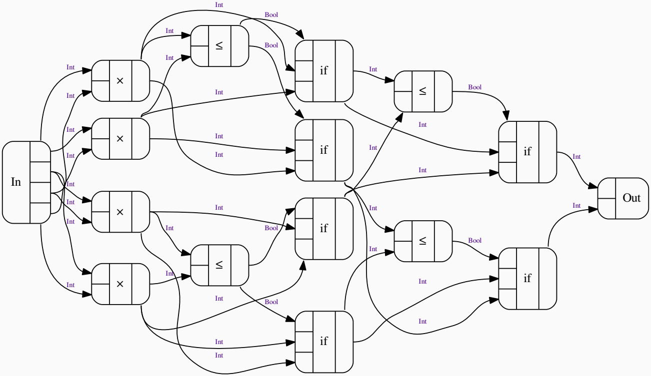
Subtraction

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



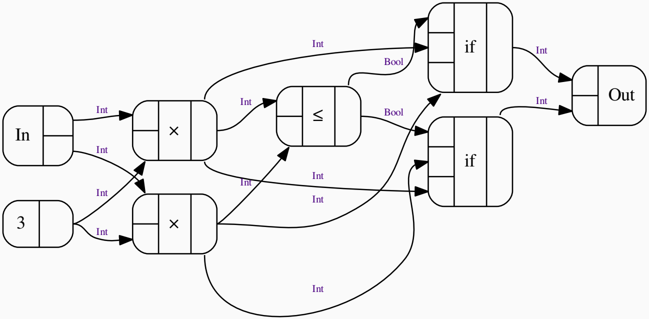
Multiplication

```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 #-}
```



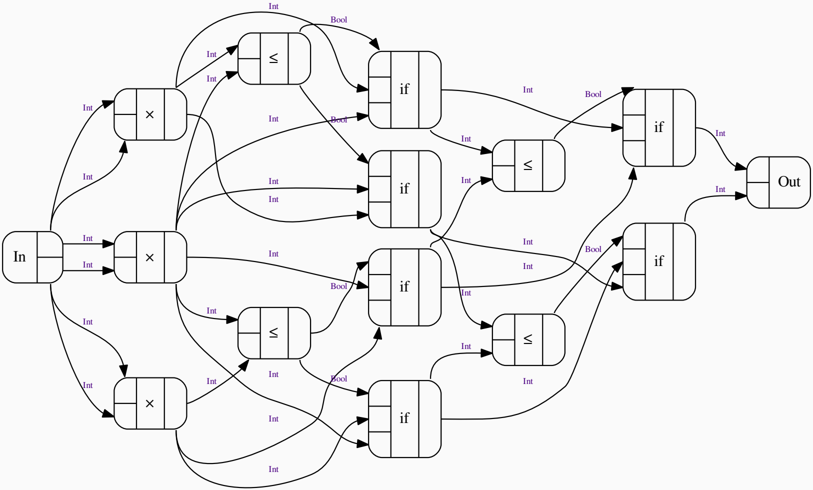
Thrice

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



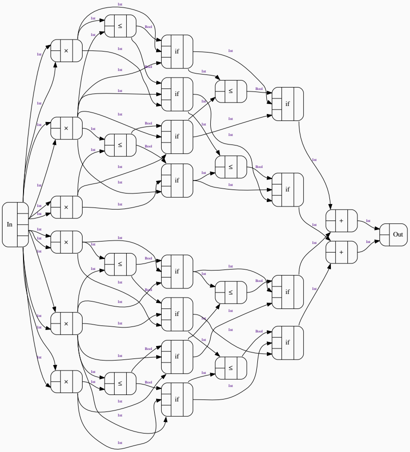
Square

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



Magic Square

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



Example: Category Products

Example: Linear maps and automatic differentiation

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!)

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 $\mathit{snd} \circ \mathit{andDeriv}$ (more explanation to come). It is obviously a functor.

Implementation of automatic differentiation in Haskell

Instead of letting functions have types such as $a \to b$, we require them to have the type $a \to b \times (a \multimap_s b)$. 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.)

Future Work

Conclusions

Further Reading

Further Reading