Compiling to Categories

mathematically-principled program transformation

T. Mark Ellison and Siva Kalyan

November 14, 2017

Australian National University

Table of contents

Haskell and Category Theory

Haskell	Category Theory
Category	**Category**
Type	**Object**
Function	**Morphism**
Hask	**Set**
...	**Terminal Objects**
Value	**Global Element**
Tuple	**Product**
Currying, Function Application	**Cartesian Closure**
Type Constructor, Functor	**Functor**
...	**Natural Transformation**
Applicative	**...**
...	**Adjoint Functor Pair**
Monad	**Monad**

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*.

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

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* $1_A\colon A \to A$ $(= \textbf{id})$. 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**:

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

Global Elements

A *global element* of an object A in category $\underline{\mathbf{C}}$ with terminal object 1 is an arrow $a : 1 \to A$.

$$1 \xrightarrow{a} A$$

In **Hask**, if we have a value v in some type a, we can upgrade it to the global element by use of **const** v.

```haskell
const :: a -> b -> a -- but for our purposes, choose b = ()
const v = \ _ -> v
```

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

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

Products in 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.

Functors

Functors

A *functor* is a mapping $F\colon \underline{\mathbf{C}} \to \underline{\mathbf{D}}$ that takes objects in $\underline{\mathbf{C}}$ to objects in $\underline{\mathbf{D}}$ and arrows in $\underline{\mathbf{C}}$ to arrows in $\underline{\mathbf{D}}$, in such a way that

1. for any $A \in \mathrm{Obj}(\underline{\mathbf{C}})$, $F(1_A) = 1_{F(A)}$:

$$\begin{tikzcd}
A \arrow[rr, "1_A"] \arrow[d, mapsto] & {} \arrow[d, mapsto] & A \arrow[d, mapsto] \\
F(A) \arrow[rr, "1_{F(A)}"'] & {} & F(A)
\end{tikzcd} ;$$

2. for any $f\colon A \to B$ and $g\colon B \to C$ in $\underline{\mathbf{C}}$, $F(g \circ f) = F(g) \circ F(f)$:

$$\begin{tikzcd}
 & B \arrow[dr, "g"] \arrow[dd, mapsto] & \\
A \arrow[ur, "f"] \arrow[rr, "g\circ f"'] \arrow[dd, mapsto] & & C \arrow[dd, mapsto] \\
 & F(B) \arrow[dr, "F(g)"] & \\
F(A) \arrow[ur, "F(f)"] \arrow[rr, "F(g)\circ F(f)"'] & & F(C)
\end{tikzcd} .$$

Functors in Haskell

In Haskell, functors are *type constructors*: they take a type (`a`) and produce another type (`F a`); and via `fmap`, they take an arrow between two types (`a -> b`) and produce an arrow between the images of those two types (`F a -> F b`).

E.g. the list constructor:

```haskell
data [] a = [] | a : [a] -- "[]" is the type constructor for lists
fmap f [] = [] -- mapping f over an empty list does nothing
fmap f (x : xs) = (f x) : (fmap f xs)
-- to turn f into a list function, apply f to the head of the list,
-- apply the list version of f to the tail of the list, and construct
```

You can verify the functor laws in **Hask**:

`fmap id (x : xs) = (id x) : (fmap id xs) = id (x : xs)`, and that

`fmap f (fmap g (x : xs)) = fmap f ((g x) : (fmap g xs))`

`= (f g x) : (fmap f (fmap g xs)) = fmap f g (x : xs)`.

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	$\underline{\mathbf{1}}$
Product	$A \times B$	`(a,b)`	$\min(a, b)$	$\underline{\mathbf{C}} \times \underline{\mathbf{D}}$
Endofunctors	functors	type const.	OPTs	nat. trans.

Cartesian-Closed Categories

Cartesian-Closed Categories (CCC)

There is a terminal object 1.

There are binary products $\sqcap$.

There is a two-argument functor taking $A \sqcap B$ onto B^A, obeying the following rules:

$$A \cong 1 \sqcap A \cong A^1$$

$$\mathrm{Hom}_{\underline{\mathbf{C}}}(A \sqcap B, C) \cong \mathrm{Hom}_{\underline{\mathbf{C}}}(A, C^B) \tag{3.1}$$

The latter relation is called the *Howard-Curry isomorphism*, or *currying*.

Cartesian-Closed Categories

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-calcululus 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 $\mathtt{id} = \lambda x \mapsto x$,
- 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 $\texttt{curry}\,f = \lambda\,ab \mapsto f\;(a, b)$
- 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 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

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

3. Expression Body is a Pair

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

4. Case Expressions

(more complexity than we wish to cover here)

5a. Simple Constants

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

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 lambda-calculus
- `simplifier` reduces the lambda-calculus size where possible
- `concat` intervenes in the simplifier and converts the lambda 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

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

```
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
```

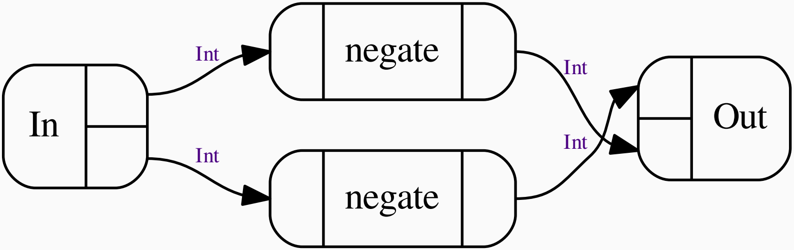
λ-Calculus to CCC Constructions

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 #-}
  ..
```

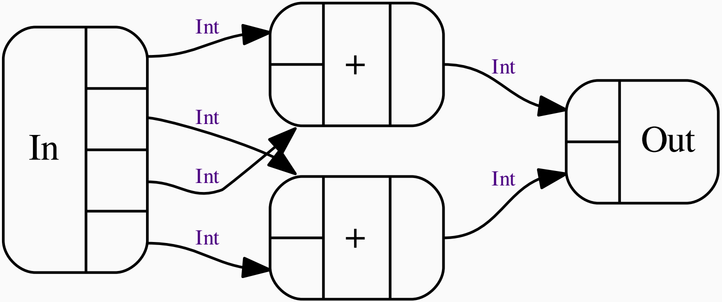
Addition

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

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

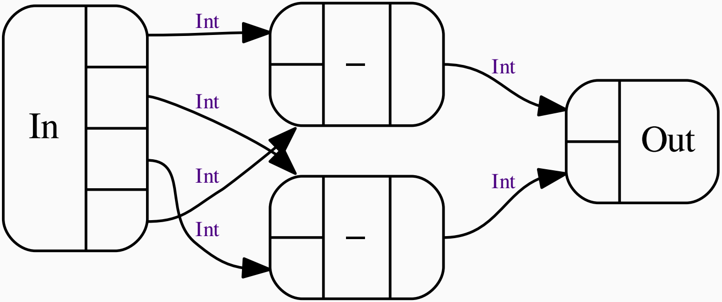
Addition

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



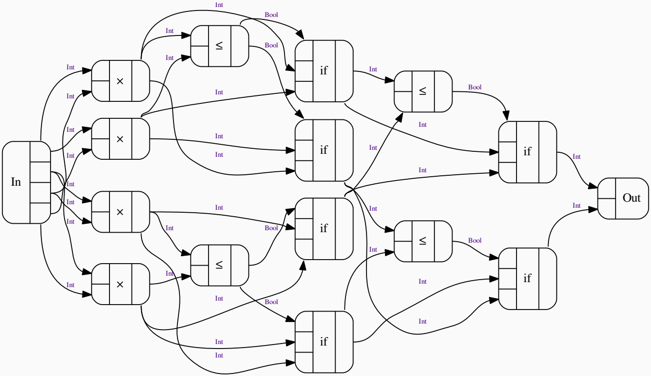
Subtraction

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



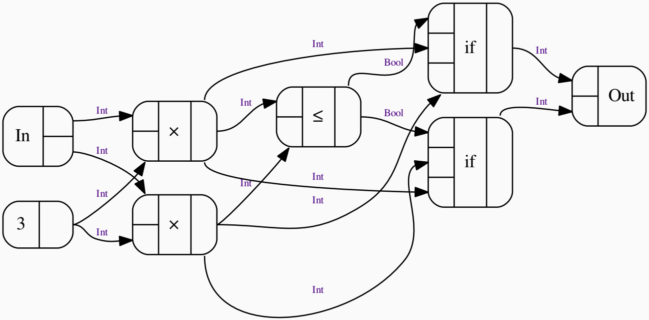
Multiplication

```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 #-}
```



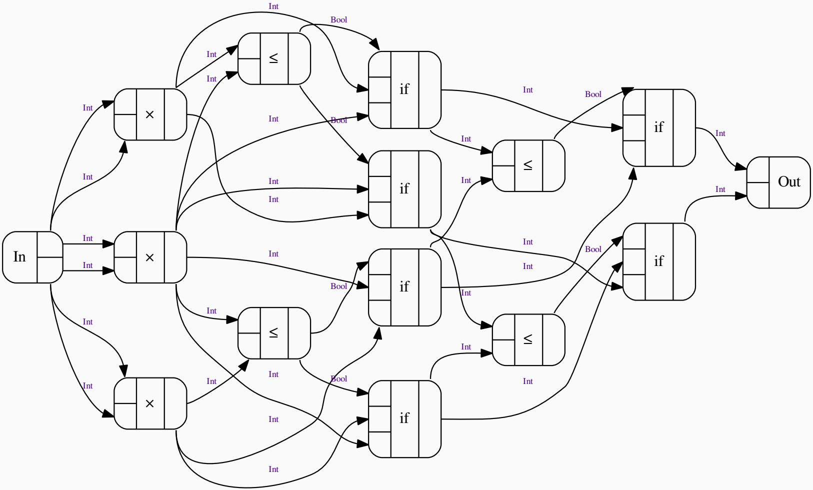
Thrice

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



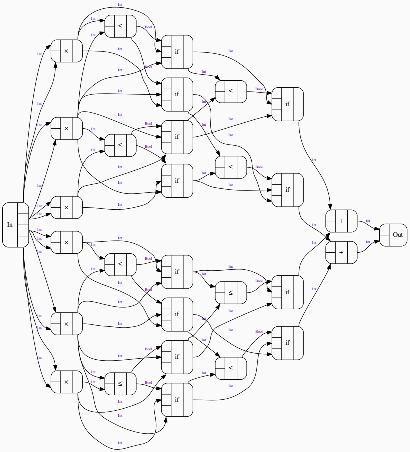
Square

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



Magic Square

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



Example: Category Products

Future Work

Conclusions

Further Reading

Further Reading