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**
<u>**Hask**</u>	<u>**Set**</u>
...	**Terminal Objects**
Tuple	**Product**
Currying, Function Application	**Cartesian Closure**

Categories

Categories

A *category* **C** consists of

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

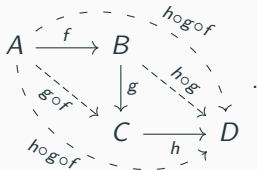$$A \underset{\Longrightarrow}{\overset{\mathrm{Hom}_{\mathbf{C}}(A,B)}{\Longrightarrow}} B$$

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**).
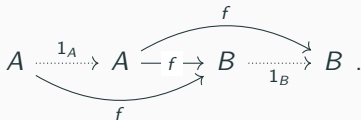
Category Laws

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

1. Given arrows $f \colon A \to B$ and $g \colon B \to C$ in $\underline{\mathbf{C}}$, the *composition* $g \circ f \colon A \to C$ ($= \mathbf{g}.\mathbf{f}$) is also in $\underline{\mathbf{C}}$.

2. Given arrows $f \colon A \to B$, $g \colon B \to C$ and $h \colon C \to D$, $(h \circ g) \circ f = h \circ (g \circ f) = h \circ g \circ f$:

$$A \xrightarrow{\ f\ } B \qquad \xrightarrow{h \circ g \circ f}$$

$$\downarrow g \qquad h \circ g$$

$$g \circ f \qquad C \xrightarrow{\ h\ } D$$

$$h \circ g \circ f$$

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

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

$$f \qquad \qquad f$$

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**	$a = a$	$1_{\underline{C}}$

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:

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

$$
\begin{array}{c}
A \\
B \;-\; \exists! \;\to\; 1 \\
C
\end{array}
$$
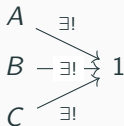
In **Hask**:

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

Products

Given objects A, B in $\underline{\mathbf{C}}$ there may be a *(pairwise) product*
$A \sqcap B \in \mathrm{Obj}(\underline{\mathbf{C}})$ 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 $x \colon X \to A \sqcap B$
such that $a = \pi_A \circ x$ and $b = \pi_B \circ x$:

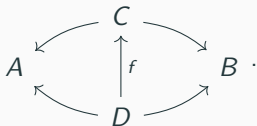$$A \xleftarrow{\ \pi_A\ } A \sqcap B \xrightarrow{\ \pi_B\ } B$$
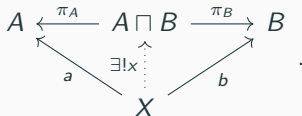
In other words: Given a particular way of mapping X to A and to B,
there's only *one* way of mapping X to $A \sqcap B$ such that everything's
consistent.

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 ⊓ B)
2    fst :: (a,b) −> a −− the projection function π_A
3    fst (x,y) = x
4    snd :: (a,b) −> b −− the projection function π_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

	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	$a = a$	$1_{\underline{C}}$
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 $[A \to B]$) is an object with an arrow eval_B^A such that for any C and any arrow $f \colon C \sqcap A \to B$,

$$
\begin{array}{ccc}
C \sqcap A & & \\
\Big\downarrow{\scriptstyle\exists!} & \searrow^{f} & \\
B^A \sqcap A & \xrightarrow[\mathrm{eval}_B^A]{} & B
\end{array}
\quad .
$$

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

$$
\begin{array}{ccc}
D \sqcap A & \searrow & \\
\Big\downarrow & & B \\
C \sqcap A & \nearrow &
\end{array}
\quad \cdot
$$

12

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 $\sqcap$ (and hence all finite products).

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

Examples:

<u>**Set**</u> the singleton set, pairs, sets of functions

<u>**Hask**</u> (), (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**.

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.

Single most exciting paper in the interpretation of programming languages.

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.

16

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

- 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 Iv a
2   type instance Iv ()     = ()
3   type instance Iv Float = Float :∗ Float
4   type instance Iv Double = Double :∗ Double
5   type instance Iv Int   = Int    :∗ Int
```

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

```
1   data IF a b = IF { unIF :: Iv a -> Iv 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.

```
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))
```

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
```

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

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

```
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

From λ-Calculus to CCCs

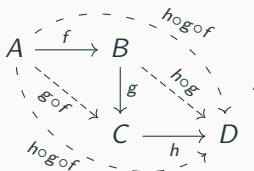The mapping operation is implemented as a pseudo-function `ccc`.

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

$$A \xrightarrow[]{\text{Hom}_{\underline{\mathbf{c}}}(A,B)} B$$

$$A \xrightarrow{\ f\ } B \xrightarrow{\ g\ } C \xrightarrow{\ h\ } D \ .$$

with dashed arrows labeled $h \circ g \circ f$, $h \circ g$, $g \circ f$, and $h \circ g \circ f$, and $\downarrow g$ from B to C.

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

with curved arrows labeled f above and f below.

Category Definition

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

Laws:

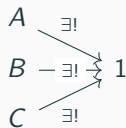- $\text{id} \circ f \equiv f \circ \text{id} \equiv f$
- $h \circ (g \circ f) \equiv (h \circ g) \circ f$

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

$$
\begin{array}{l}
A \searrow^{\exists!} \\
B - \exists! \rightrightarrows 1 \\
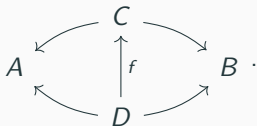C \nearrow_{\exists!}
\end{array}
$$

A Terminal Object

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

Laws:

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

Simple Constants

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

$$A \quad C \quad B \cdot$$
$$f$$
$$D$$

The Product

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

Laws:

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

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

Exponential Objects

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

Laws:

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

Expression Body is an Application

Expression body is an application

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

Lambda abstraction

$$\mathrm{ccc}(\lambda\, x \mapsto \lambda\, y \mapsto U) = \mathtt{curry}\; \mathrm{ccc}(\lambda\, (x, y) \mapsto U)$$

Constant functions

$$\mathrm{ccc}(\lambda\, x \mapsto f) = \mathtt{constFun}\; \mathrm{ccc}(f)$$

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

# Example: Syntactic Analysis

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.

Example: Interval Analysis

Negation

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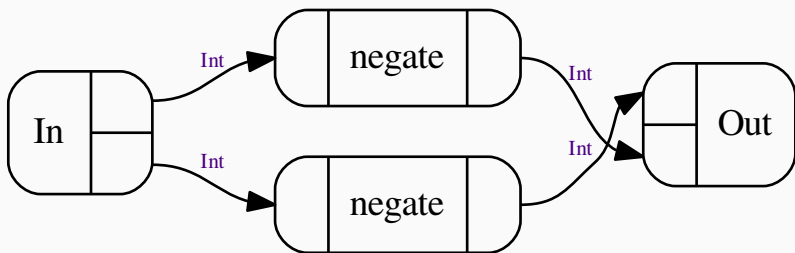
Addition

```
1   instance (Iv 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       ..
```
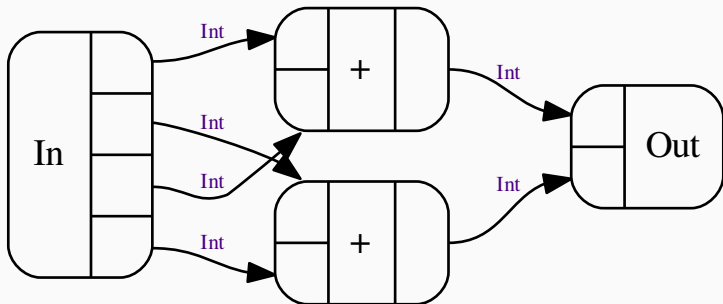
Addition

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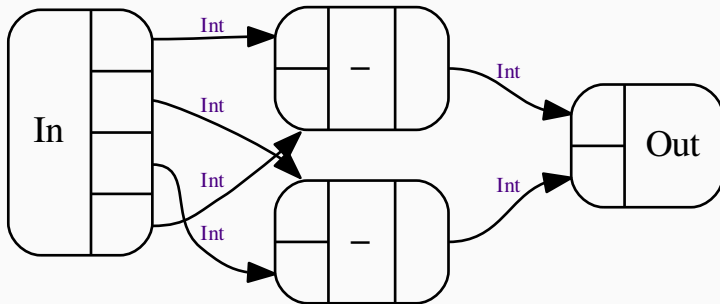
```
runSynCirc "add" $ toCcc $ ivFun $ uncurry ((+) @Int)
```
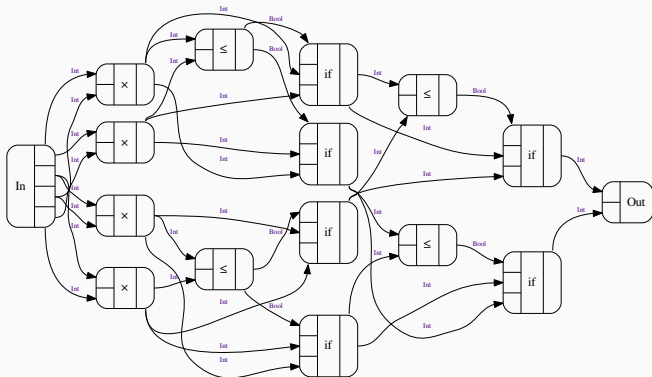
Subtraction

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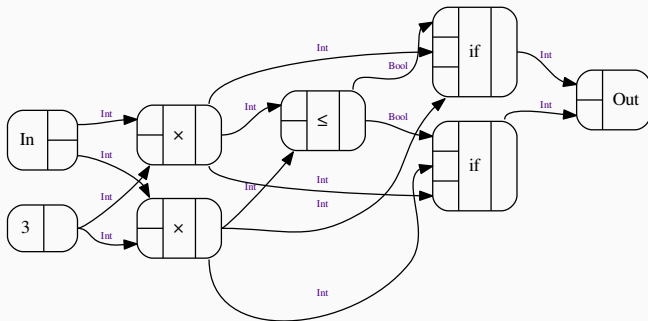
Multiplication

```
1  instance (Iv 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 #-}
```
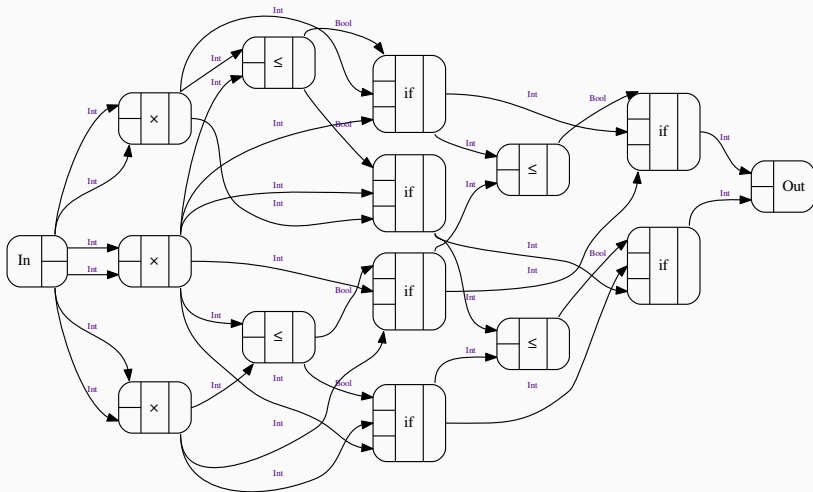
```
1    runSynCirc "thrice-iv" $ toCcc $ ivFun $ \ x -> 3 * x :: Int
```
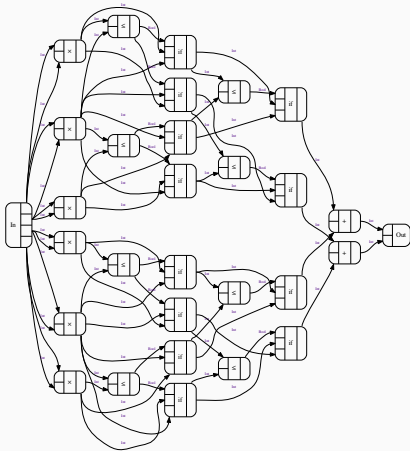
Square

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

Magic Square

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

Example: Hardware

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 $f: \mathbb{R}^m \to \mathbb{R}^n$ 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
the domain space map to).

Linear maps form a category, because:

1. Given $f: \mathbb{R}^m \to \mathbb{R}^n$ and $g: \mathbb{R}^n \to \mathbb{R}^p$, we can define the composition
   $g \circ f: \mathbb{R}^m \to \mathbb{R}^p$, 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 $1_{\mathbb{R}^n}: \mathbb{R}^n \to \mathbb{R}^n$ 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.

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 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 D with the property that

$$D(g \circ f) = (Dg \circ f) \circ (Df).$$

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 D to all our atomic functions, then we know how to apply D to all compositions of these functions.

Implementation of automatic differentiation in Haskell

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

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

(Now that we've redefined functions, it turns out D is a functor after all.)

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 `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